Logo, company name

Description automatically generated

Lebanese American University

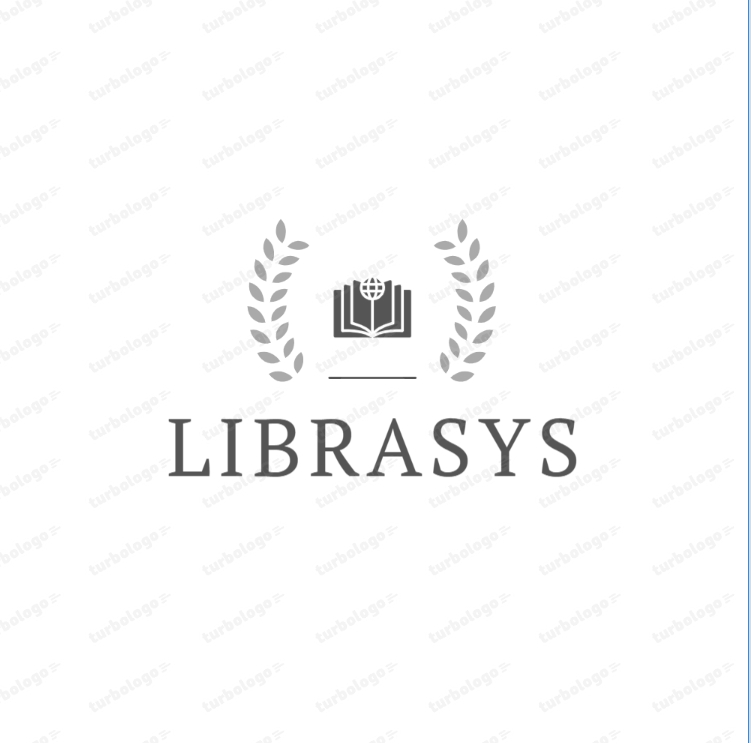
School of Arts and Sciences

Department of Computer Science and Mathematics

-

CSC 375: Database Management Systems

**LibraSys: Unified Library Management System**

****

**Students:**

Omar Al Ghadban

Makram Kordab

**Instructor:** Dr. Khaleel Mershad

December 2024

Introduction:

Libraries have long been regarded as bastions of knowledge and culture, playing a vital role in preserving history, fostering education, and promoting intellectual curiosity. Over the years, these institutions have evolved to become dynamic spaces that offer a diverse array of resources and services to cater to the growing needs of their communities. Institutions like *City Horizons Library* and *Evergreen Knowledge Hub* represent this transformation, offering not only books but also digital media, private study areas, and specialized resources. The management of these resources and the interactions between members, staff, and materials necessitate a sophisticated and efficient system to ensure that operations remain seamless and user-friendly.

In a library setting, the volume of data generated daily can be staggering. From member registrations and book reservations to loan management and supplier transactions, the need for a centralized system to manage these processes has never been greater. A robust database is not just a luxury—it is an operational necessity. It allows for the efficient storage, retrieval, and manipulation of data, ensuring accuracy and minimizing errors. By integrating various entities, such as books, members, authors, publishers, and staff, a database facilitates streamlined workflows and enhances the overall user experience. For example, tracking loans and their statuses in real time ensures that librarians can monitor overdue returns effectively and take necessary actions, such as updating member records or issuing fines.

The motivation for developing a comprehensive library database arises from the challenges posed by traditional record-keeping methods, which are often manual, fragmented, and prone to human error. Such systems struggle to keep pace with the demands of modern libraries, where members expect instant access to resources, accurate availability information, and hassle-free interactions. Moreover, libraries often deal with complex relationships between entities. For instance, a single book might be associated with multiple reviews, copies located on different floors, and various suppliers. Similarly, members may have simultaneous reservations for study rooms and books, adding another layer of complexity. A well-designed database not only simplifies these relationships but also introduces advanced features like automated notifications for due dates, statistical insights into resource utilization, and digital cataloging for quick searches.

Furthermore, the role of library staff, particularly librarians, archivists, and janitors, underscores the need for a centralized database. In libraries like *Evergreen Knowledge Hub*, librarians are responsible for monitoring loans, ensuring they are updated with accurate statuses, and resolving issues such as overdue books or lost items. Archivists oversee the management of valuable recordings or collections, while janitors ensure that the library floors remain organized and conducive to study. These staff members rely heavily on accurate and up-to-date data to perform their duties effectively. A database system ensures that all their operations are interconnected, enabling collaboration and efficient task execution.

Additionally, procurement and supplier management are critical components of library operations. Libraries frequently acquire new materials to meet the diverse demands of their members. Tracking orders, monitoring deliveries, and maintaining supplier records require meticulous attention, which can be seamlessly managed through a database. By automating these processes, libraries can ensure timely updates and maintain a consistent flow of resources.

In today’s fast-paced, information-driven world, the success of libraries hinges on their ability to adapt to technological advancements. A comprehensive database empowers libraries like *City Horizons Library* to meet these challenges head-on. By integrating all aspects of library management into a single, cohesive system, this project seeks to create a framework that supports growth, enhances user satisfaction, and upholds the library’s mission of fostering knowledge and community engagement. Through this database, the library will not only optimize its operations but also reaffirm its role as a modern hub for learning and exploration.

# Requirements:

A library requires a robust system to manage its diverse resources, user interactions, and staff operations. The system will encompass various elements, capturing detailed information about books, authors, publishers, suppliers, orders, and how these entities interrelate.

Each book will be linked to its author and publisher, allowing authors and publishers to be associated with multiple books. Book attributes will include BookID, Title, AuthorID, ISBN, PublisherID, Genre, PublicationYear, and CopiesAvailable. Authors will have attributes such as AuthorID, FirstName, LastName, BirthDate, and Nationality, while publisher details include PublisherID, Name, Address (composite: Street, City, State, ZipCode), Phone (multi-valued: HomePhone, WorkPhone, MobilePhone), and Email.

The system will track member records, including details such as MemberID, FirstName, LastName, MembershipDate, Email, Phone (multi-valued), and Address (composite). Members can make reservations for books or private study rooms, and the system will store ReservationID, MemberID, BookID or RoomID, ReservationDate, and ExpirationDate. Members can also write reviews on books they borrow, with each book linked to multiple reviews. Review attributes include ReviewID, BookID, MemberID, Rating, Comments, and ReviewDate, and the system will calculate an average rating for each book based on these reviews.

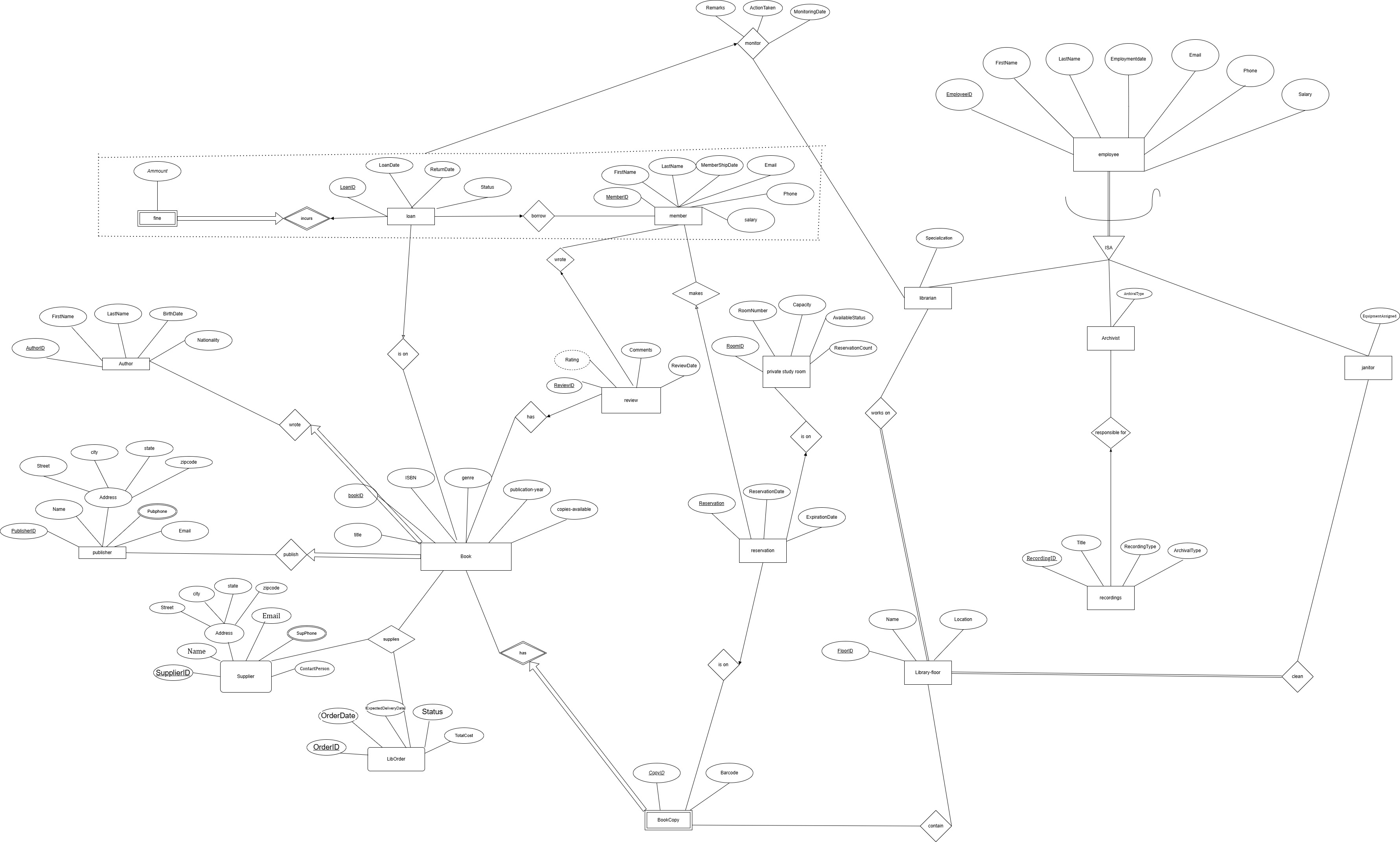
The system will also track loans made by members, where members can borrow multiple books over time. Loan attributes include LoanID, MemberID, BookID, LoanDate, ReturnDate, and Status. Fines may be associated with specific loans if books are returned late. Librarians will monitor these loans to ensure compliance with library rules and update loan statuses as needed, such as marking them as returned, overdue, or associated with a fine.

Library staff records will include employees who are librarians, archivists, and janitors. Common attributes for all employees include EmployeeID, FirstName, LastName, EmploymentDate, Email, Phone, and Salary. Librarians specifically will oversee library floors, ensuring the proper management of books located on these floors. Their primary responsibilities include monitoring loans, updating loan records after monitoring, and managing the library floors they are assigned to.

The system will support procurement operations through relationships between Book, Supplier, and Order. Suppliers provide books to the library based on orders, and these interactions will track Quantity, PricePerUnit, TotalCost, and DeliveryDate. Supplier information will include SupplierID, Name, ContactPerson, Phone, Email, and Address. Orders will track OrderID, OrderDate, ExpectedDeliveryDate, and Status, completing the library’s procurement process.

Finally, complex attributes such as multi-valued phone numbers, composite addresses, and derived attributes like book reviews (averaged ratings) will be integrated into the system. Relationships such as the one between books, suppliers, and orders will be fundamental to the system's design. Librarians will play a critical role in monitoring loans, updating records, and maintaining order on library floors.

ER DIAGRAM



# Entities

1. Book  
   The **Book** entity represents the core resource in the library, encompassing all the essential details about the books available for lending or reference. Each book is uniquely identified by a BookID and includes attributes like Title, ISBN, Genre, and PublicationYear to categorize and describe it. Relationships link books to their respective authors and publishers through AuthorID and PublisherID, ensuring that books are tied to their creators and distributors. Additionally, the CopiesAvailable attribute tracks the number of copies of the book currently present in the library, aiding in resource management.
2. Author  
   The **Author** entity captures information about the creators of books in the library. Uniquely identified by an AuthorID, this entity includes the author’s FirstName and LastName for proper identification, along with their BirthDate to provide historical context. The Nationality attribute records the author’s country of origin, which can be useful for categorizing books by cultural or regional significance. This entity establishes a one-to-many relationship with the Book entity, as an author can write multiple books.
3. Publisher  
   The **Publisher** entity represents the organizations responsible for producing and distributing books. Each publisher is uniquely identified by a PublisherID and includes details such as the Name of the publisher, Address (a composite attribute comprising Street, City, State, and ZipCode), and multiple Phone numbers for contact purposes. The Email attribute facilitates electronic communication with the publisher. By associating with multiple books, the Publisher entity ensures that records of publishing and distribution are maintained seamlessly.
4. Member  
   The **Member** entity represents library users who have registered for borrowing or reserving resources. Each member is assigned a unique MemberID and is identified by their FirstName, LastName, MembershipDate, and contact details such as Email, Phone (multi-valued), and a Address (composite). Members can reserve books or private study rooms and write reviews on the books they borrow, highlighting their integral role in the library's operations.
5. Librarian  
   The **Librarian** entity represents library staff responsible for managing book loans, ensuring proper record updates, and maintaining order on library floors. Each librarian is uniquely identified by an EmployeeID and has attributes like FirstName, LastName, EmploymentDate, Email, Phone,Specialiation and Salary. Additionally, librarians are linked to the floors they oversee, indicated by the LibraryFloor attribute. Their primary role involves monitoring loans, updating their statuses, and ensuring compliance with library rules.
6. Archivist  
   The **Archivist** entity captures details about staff members responsible for managing valuable recordings and collections. Each archivist is uniquely identified by an EmployeeID and includes attributes like FirstName, LastName, EmploymentDate, Email, Phone, and Salary. Archivists specialize in managing specific types of archival material, noted in the ArchivalType attribute, and play a crucial role in preserving the library’s historical and cultural resources.
7. Janitor  
   The **Janitor** entity represents library staff tasked with ensuring cleanliness and order across library floors. Uniquely identified by an EmployeeID, each janitor’s details include FirstName, LastName, EmploymentDate, Email, Phone, and Salary. The LibraryFloor entity indicates the floors a janitor is assigned to, while EquipmentAssigned attribute tracks tools or cleaning equipment assigned to the janitor, like vacuums, polishers, or sanitization gear. This ensures accountability and efficient resource tracking.
8. Loan  
   The **Loan** entity tracks the borrowing of books by members. Each loan is uniquely identified by a LoanID and is associated with a MemberID and BookID. Attributes like LoanDate, ReturnDate, and Status provide detailed records of the borrowing process. Fines, if applicable, are linked to a loan, ensuring proper enforcement of borrowing rules and timely returns.
9. Fine (weak entity)  
   The **Fine** entity represents monetary penalties associated with overdue book loans. Uniquely identified by the combination of LoanID and Amount, it ensures a direct connection between a specific loan and the fine imposed. This entity helps maintain discipline among borrowers and ensures the efficient management of resources.
10. Reservation  
    The **Reservation** entity records bookings made by members for either books or private study rooms. Each reservation is uniquely identified by a ReservationID and is linked to a MemberID and either a BookID or RoomID. Attributes such as ReservationDate and ExpirationDate ensure the reservation process is properly managed and time-sensitive.
11. Review  
    The **Review** entity captures feedback provided by members on books they have borrowed. Each review is uniquely identified by a ReviewID and is linked to a BookID and MemberID. Attributes like Rating, Comments, and ReviewDate provide detailed insights into user experiences. Additionally, the system derives the average rating for each book based on multiple reviews.
12. PrivateStudyRoom  
    The **PrivateStudyRoom** entity represents study spaces available for reservation. Each room is uniquely identified by a RoomID and includes details such as RoomNumber, Capacity, and AvailabilityStatus. The ReservationsCount attribute tracks the number of times a room has been reserved, ensuring efficient allocation and usage.
13. Supplier  
    The **Supplier** entity tracks external vendors who provide books to the library. Each supplier is uniquely identified by a SupplierID and includes details like Name, ContactPerson, Phone, Email, and Address (composite). Suppliers are critical for maintaining a steady flow of resources to the library.
14. Order  
    The **Order** entity records procurement transactions between the library and suppliers. Each order is uniquely identified by an OrderID and includes attributes like OrderDate, ExpectedDeliveryDate, Status, and TotalCost. This entity ensures that book acquisition is efficiently managed and documented.
15. LibraryFloor  
    The **LibraryFloor** entity represents the different sections of the library. Each floor is uniquely identified by a FloorID and includes attributes like Name, Location, and the ManagerID of the librarian responsible for overseeing it. Floors play a critical role in organizing the library’s physical layout.
16. BookCopy  
    The **BookCopy** entity represents individual copies of a book available in the library. It is a weak entity uniquely identified by a CopyID and associated with a BookID and FloorID. Attributes like Barcode help track and manage individual copies across library floors.
17. Recording  
    The **Recording** entity represents valuable audiovisual or archival materials managed by the library. Each recording is uniquely identified by a RecordingID and includes attributes like Title, Type, and ArchivalDate. Recordings are primarily managed by archivists, contributing to the library’s cultural and historical preservation efforts.

Relationships Descriptions

Author-Book (One-to-Many): “wrote”  
An **author** can write multiple books, establishing a one-to-many relationship between the Author and Book entities. This reflects the real-world scenario where authors often create multiple works over their careers. Each book, however, is attributed to one specific author in the database, ensuring clear ownership and simplifying the process of linking books to their creators.

Publisher-Book (One-to-Many): “published”  
A **publisher** can publish multiple books, forming a one-to-many relationship with the Book entity. This relationship allows the system to track the publishers responsible for producing the library's collection. Each book is linked to one publisher, ensuring accountability for its production details, such as the edition or format.

Member-Loan (One-to-Many): “borrow”  
A **member** can borrow multiple books through distinct loans, creating a one-to-many relationship between the Member and Loan entities. Each loan is associated with one member and one book, ensuring that borrowing transactions are tracked accurately. This allows the system to monitor borrowing histories and manage overdue loans or fines effectively.

Book-Loan (One-to-Many): “is on”  
A **book** can appear in multiple loans over time, establishing a one-to-many relationship between the Book and Loan entities. Each loan involves only one book, enabling the system to track how often a book is borrowed and its current status, whether available, on loan, or overdue.

Book-BookCopy (One-to-Many): “has”  
A **book** can have multiple physical copies in the library, forming a one-to-many relationship between the Book and BookCopy entities. This relationship ensures that the system can differentiate between copies of the same title, track their locations, and monitor their availability status individually.

Member-Reservation (One-to-Many): “makes”  
A **member** can make multiple reservations, creating a one-to-many relationship with the Reservation entity. Each reservation is linked to a single member, enabling the system to track which books or rooms have been reserved by specific members and manage reservation expiration dates.

BookCopy-Reservation (One-to-Many): “is on”  
A **book copy** can have multiple reservations placed on it, forming a one-to-many relationship with the Reservation entity. This allows the system to track individual copies of books that are reserved, ensuring members are notified when the reserved copy becomes available.

PrivateStudyRoom-Reservation (One-to-Many): “is on”  
A **private study room** can have multiple reservations, establishing a one-to-many relationship with the Reservation entity. Each reservation is linked to one study room, allowing the library to manage room availability and ensure fair usage among members.

Book-Review (One-to-Many): “has”  
A **book** can have multiple reviews written by members, forming a one-to-many relationship between the Book and Review entities. Each review is associated with one specific book, enabling the system to store member feedback and calculate an average rating for the book based on these reviews.

Member-Review (One-to-Many): “wrote”  
A **member** can write multiple reviews, creating a one-to-many relationship between the Member and Review entities. Each review is linked to one member, helping the system track which members have contributed feedback and their individual opinions on borrowed books.

Loan-Fine (One-to-One): “incurs”  
A **loan** can incur one fine if the book is returned late, forming a one-to-one relationship between the Loan and Fine entities. This ensures that each loan is linked to its corresponding fine, allowing the system to calculate penalties and update member records accordingly.

Librarian-LibraryFloor (Many-to-Many): “works on”  
A **librarian** can manage multiple library floors, and each floor can be managed by several librarians, creating a many-to-many relationship. This allows the system to assign specific responsibilities and track which librarians are overseeing operations on each floor.

LibraryFloor-BookCopy (One-to-Many): “contains”  
A **library floor** can house multiple book copies, establishing a one-to-many relationship with the BookCopy entity. Each book copy is assigned to one specific floor, enabling the system to track the physical location of resources within the library.

Archivist-Recording (One-to-Many): “responsible for”  
An **archivist** can oversee multiple recordings or collections, forming a one-to-many relationship with the Recording entity. This relationship ensures that the system can track which archivists are responsible for specific recordings, enabling efficient management of valuable library collections.

Janitor-LibraryFloor (Many-to-Many): “cleans ”  
A **janitor** can clean and maintain multiple library floors, and each floor can be assigned to multiple janitors, creating a many-to-many relationship. This ensures that cleaning tasks are distributed effectively, and the system can track which janitors are responsible for each floor.

Complex Attributes

The **Address** attribute is a **composite attribute** that breaks down into four components: **Street**, **City**, **State**, and **ZipCode**. This structure allows the system to capture detailed address information for entities such as members, librarians, or library branches, ensuring accurate location tracking and effective communication.

The **Phone** attribute is a **multi-valued attribute** that includes multiple types of contact numbers: **HomePhone**, **WorkPhone**, and **MobilePhone**. This flexibility ensures that the system can store various contact methods for individuals, accommodating their preferences and ensuring effective communication.

The **Review** attribute is a **derived attribute** that represents the **average rating** of a book, calculated based on multiple individual reviews. This derived value is not stored directly in the database but is computed dynamically using the ratings provided by members. It helps the library system display overall feedback on books, assisting other members in making informed choices.

# ISA Relationship

# The ISA relationship establishes that the entities Librarian, Archivist, and Janitor inherit from the more general entity Employee. This hierarchy represents a specialization where each subclass retains the common attributes of an employee—EmployeeID, FirstName, LastName, EmploymentDate, Email, Phone, and Salary—while also having its own specific attributes and responsibilities. For instance, librarians manage loans and library floors, archivists oversee recordings and collections, and janitors ensure cleanliness and maintenance. This structure allows the system to efficiently manage both shared and unique characteristics of staff, enhancing data organization and supporting role-specific functionality.

Ternary Relationship

The ternary relationship between **Book**, **Supplier**, and **Order** captures the intricate interactions involved in library procurement. This relationship represents the process where a **Supplier** provides a specific **Book** to the library through a particular **Order**. Each entity plays a crucial role: the **Supplier** ensures availability, the **Book** represents the resource being acquired, and the **Order** tracks the procurement details such as quantity, price, and delivery timelines.

This ternary design is essential for managing the library's supply chain efficiently, as it links suppliers to the exact books they provide and the orders they fulfill. By incorporating attributes like **Quantity**, **PricePerUnit**, and **TotalCost**, this relationship enables detailed tracking of procurement transactions, ensuring accuracy in inventory management and budgeting. It also supports processes such as identifying reliable suppliers, analyzing order histories, and maintaining a steady flow of resources to meet the library's demands.

Aggregation Relationship (Monitors)

Loan, Member: Aggregation between member and loan, which is to be monitored by a librarian.

The **Monitors** relationship connects librarians to the loans they oversee, emphasizing their responsibility for managing and ensuring compliance with library policies. This relationship includes the attributes **LibrarianID** and **LoanID** as foreign keys, establishing links to the **Librarian** and **Loan** entities, respectively. Additional attributes like **MonitoringDate** capture the specific date on which the librarian reviewed or acted upon the loan, while **Remarks** provide space for recording observations or comments about the loan status. The **ActionTaken** attribute documents any steps the librarian takes, such as marking a book as returned, issuing a fine, or sending a notification to a member. This relationship supports transparency and accountability in loan management, ensuring that all actions are systematically recorded and traceable.

Relational model

**Legends:**

Primary key

#Foriegn key

Not NULL

Book (BookID, Title, ISBN, Genre, PublicationYear, CopiesAvailable, #AuthorID, #PublisherID)

Author (AuthorID, FirstName, LastName, BirthDate, Nationality)

Publisher (PublisherID, Name, street, city, state, zipcode, Phone, Email)

Member (MemberID, FirstName, LastName, MembershipDate, Email, Phone, Address)

Employee (EmployeeID, FirstName, LastName, EmploymentDate, Email, Phone, Salary)

Librarian (#EmployeeID, specialization)

Archivist (#EmployeeID, ArchivalType)

Janitor (#EmployeeID, EquipmentAssigned)

Loan (LoanID, LoanDate, ReturnDate, Status, #BookID, #MemberID)

Fine (#LoanID, Amount)

Reservation (ReservationID, ReservationDate, ExpirationDate, #MemberID, #BookID, #CopyID, #RoomID)

Review (ReviewID, Rating, Comments, ReviewDate, #BookID, #MemberID)

Recording (RecordingID , Title, ArchivalType, RecordingType)

PrivateStudyRoom (RoomID, RoomNumber, Capacity, AvailabilityStatus, ReservationsCount)

Supplier (SupplierID, Name, ContactPerson, Phone, Email, street, city, state, zipcode)

LibOrder (OrderID, OrderDate, ExpectedDeliveryDate, Status, TotalCost)

LibraryFloor (FloorID, Name, Location)

BookCopy (CopyID, Barcode, #BookID)

Cleans (#Janitor.EmployeeID, #FloorID)

WorksOn (#Librarian.EmployeeID, #FloorID)

Contain (#FloorID, #BookID, #CopyID)

Supplies (#SupplierID, #OrderID, #BookID)

Monitors (#LoanID, #MemberID, #EmployeeID, MonitoringDate, ActionTaken, Remarks)

Database Implementation

Author Table

CREATE TABLE Author (

AuthorID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

BirthDate DATE,

Nationality VARCHAR(50)

);

**Referencing:** This table has no foreign key references, as it's a standalone table used to store author details.

Constraints:

* + PRIMARY KEY on AuthorID: Ensures each author has a unique identifier.
  + NOT NULL on AuthorID: Ensures AuthorID is provided for every record, guaranteeing uniqueness.

DML :

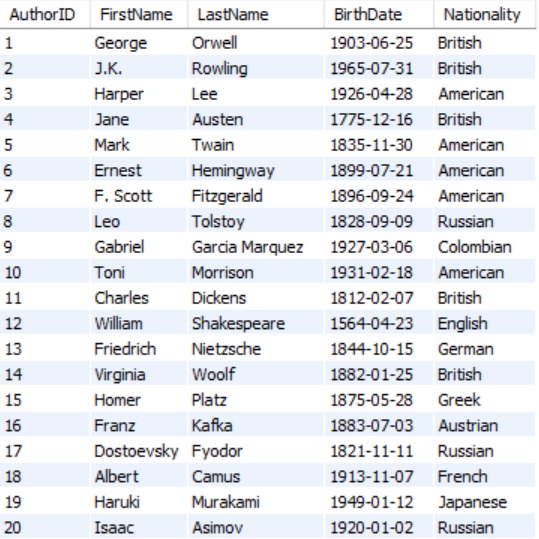
INSERT INTO Author (AuthorID, FirstName, LastName, BirthDate, Nationality)

VALUES

(1, 'George', 'Orwell', '1903-06-25', 'British'),

...

**Explanation**:

* Inserts data into the **Author** table.
* **AuthorID**: Unique identifier for each author.
* **FirstName** & **LastName**: Name of the author.
* **BirthDate**: Author's date of birth.
* **Nationality**: Country of origin or citizenship.
* 

Publisher Table

CREATE TABLE Publisher (

PublisherID INT PRIMARY KEY,

PublisherName VARCHAR(100) ,

Address\_Street VARCHAR(100),

Address\_City VARCHAR(50),

Address\_State VARCHAR(50),

Address\_ZipCode VARCHAR(10),

Phone VARCHAR(15),

Email VARCHAR(100)

);

Referencing**:** This table has no foreign key references, as it's a standalone table to store publisher information.

Constraints**:**

* PRIMARY KEY on PublisherID: Ensures each publisher has a unique identifier.
* NOT NULL on PublisherID: Ensures that a publisher's ID must be provided, guaranteeing uniqueness.

DML

INSERT INTO Publisher (PublisherID, PublisherName, Address\_Street, Address\_City, Address\_State, Address\_ZipCode, Email, Phone)

VALUES

(1, 'Penguin Books', '123 Main St', 'New York', 'NY', '10001', 'contact@penguin.com', '555-1234'),

...

**Explanation**:

* Inserts data into the **Publisher** table.
* **PublisherID**: Unique ID for each publisher.
* **PublisherName**: Name of the publishing house.
* **Address\_Street, Address\_City, Address\_State, Address\_ZipCode**: Contact address details.
* **Email** & **Phone**: Publisher's contact information.
* A screenshot of a computer

  Description automatically generated

Book Table

CREATE TABLE Book (

BookID INT PRIMARY KEY,

Title VARCHAR(200) ,

ISBN VARCHAR(20) UNIQUE ,

Genre VARCHAR(50),

PublicationYear INT,

CopiesAvailable INT DEFAULT 0,

AuthorID INT NOT NULL,

PublisherID INT NOT NULL,

FOREIGN KEY (AuthorID) REFERENCES Author(AuthorID),

FOREIGN KEY (PublisherID) REFERENCES Publisher(PublisherID)

);

Referencing**:**

* AuthorID is a foreign key that references Author(AuthorID).
* PublisherID is a foreign key that references Publisher(PublisherID).

Constraints**:**

* PRIMARY KEY on BookID: Ensures that each book has a unique identifier.
* NOT NULL on AuthorID and PublisherID: Ensures that both the author and publisher for a book must be specified.
* DEFAULT on CopiesAvailable: Ensures a default value of 0 is set if no value is specified for the number of available copies.

DML

INSERT INTO Book (BookID, Title, ISBN, Genre, PublicationYear, CopiesAvailable, AuthorID, PublisherID)

VALUES

(1, '1984', '9780451524935', 'Dystopian', 1949, 10, 1, 1),

...

**Explanation**:

* Inserts data into the **Book** table.
* **BookID**: Unique ID for each book.
* **Title**: Title of the book.
* **ISBN**: International Standard Book Number for the book.
* **Genre**: Genre or category of the book.
* **PublicationYear**: The year the book was published.
* **CopiesAvailable**: Number of copies of the book available in the library.
* **AuthorID**: Links the book to an author in the **Author** table (foreign key).
* **PublisherID**: Links the book to a publisher in the **Publisher** table (foreign key).
* A screenshot of a computer

  Description automatically generated

Member Table

CREATE TABLE Member (

MemberID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

MembershipDate DATE ,

Email VARCHAR(100),

Phone VARCHAR(15),

Address\_Street VARCHAR(100),

Address\_City VARCHAR(50),

Address\_State VARCHAR(50),

Address\_ZipCode VARCHAR(10)

);

Referencing**:** This table does not reference any other tables.

Constraints**:**

* PRIMARY KEY on MemberID: Ensures each member has a unique identifier.
* No NOT NULL constraints are defined, but you can add them as needed.

DML

INSERT INTO Member (MemberID, FirstName, LastName, MembershipDate, Email, Phone, Address\_Street, Address\_City, Address\_State, Address\_ZipCode)

VALUES

(1, 'Alice', 'Johnson', '2023-01-01', 'alice.johnson@example.com', '555-1234', '123 Oak St', 'Springfield', 'IL', '62701'),

...

**Explanation**:

* Inserts data into the **Member** table.
* **MemberID**: Unique ID for each library member.
* **FirstName** & **LastName**: Member's name.
* **MembershipDate**: Date when the person became a library member.
* **Email** & **Phone**: Member's contact information.
* **Address\_Street, Address\_City, Address\_State, Address\_ZipCode**: Member's residence address.
* 

Employee Table (Parent Table for ISA Hierarchy)

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50) ,

LastName VARCHAR(50) ,

EmploymentDate DATE ,

Email VARCHAR(100),

Phone VARCHAR(15),

Salary DECIMAL(10, 2) NOT NULL

);

Referencing: This table does not have any foreign key references. It serves as the parent table in an inheritance structure.

Constraints:

* PRIMARY KEY on EmployeeID: Ensures each employee has a unique identifier.
* NOT NULL on Salary: Ensures the salary is provided and cannot be NULL.

DML

INSERT INTO Employee (EmployeeID, FirstName, LastName, EmploymentDate, Email, Phone, Salary)

VALUES

(1, 'John', 'Doe', '2015-06-15', 'jdoe@company.com', '555-1010', 50000.00),

...

**Explanation**:

* Inserts data into the **Employee** table.
* **EmployeeID**: Unique ID for each library employee.
* **FirstName** & **LastName**: Employee's name.
* **EmploymentDate**: Date when the employee joined the library.
* **Email** & **Phone**: Employee's contact information.
* **Salary**: Annual salary of the employee.

Librarian Table (Specialized from Employee)

CREATE TABLE Librarian (

EmployeeID INT PRIMARY KEY,

Specialization VARCHAR(100),

FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)

);

Referencing:

* + EmployeeID is a foreign key that references Employee(EmployeeID).

Constraints:

* + PRIMARY KEY on EmployeeID: Ensures that each librarian is uniquely identified by their EmployeeID.
  + NOT NULL on Specialization: Ensures a librarian has a specialization defined.

DML

INSERT INTO Librarian (EmployeeID, FirstName, LastName, EmploymentDate, Email, Phone, Salary)

VALUES

(1, 'Sarah', 'Williams', '2018-03-10', 'sarah.williams@library.com', '555-2020', 45000.00),

...

**Explanation**:

* Inserts data into the Librarian table.
* EmployeeID: Unique identifier for each librarian (foreign key from the Employee table).
* FirstName & LastName: Librarian's name.
* EmploymentDate: Date when the librarian started working.
* Email & Phone: Contact details of the librarian.
* Salary: Annual salary of the librarian.
* 

Archivist Table (Specialized from Employee)

CREATE TABLE Archivist (

EmployeeID INT PRIMARY KEY,

ArchivalType VARCHAR(50),

FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)

);

Referencing**:**

* EmployeeID is a foreign key that references Employee(EmployeeID).

Constraints**:**

* PRIMARY KEY on EmployeeID: Ensures that each archivist is uniquely identified by their EmployeeID.
* NOT NULL on ArchivalType: Ensures that an archival type is specified for the archivist.

DML

INSERT INTO Archivist (EmployeeID, FirstName, LastName, EmploymentDate, Email, Phone, Salary)

VALUES

(1, 'John', 'Smith', '2017-08-05', 'john.smith@library.com', '555-3030', 40000.00),

...

**Explanation**:

* Inserts data into the Archivist table.
* EmployeeID: Unique identifier for each archivist (foreign key from the Employee table).
* FirstName & LastName: Archivist's name.
* EmploymentDate: Date when the archivist started working.
* Email & Phone: Contact details of the archivist.
* Salary: Annual salary of the archivist.
* A list of information on a computer

  Description automatically generated

Janitor Table (Specialized from Employee)

CREATE TABLE Janitor (

EmployeeID INT PRIMARY KEY,

EquipmentAssigned VARCHAR(100),

FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)

);

Referencing**:**

* EmployeeID is a foreign key that references Employee(EmployeeID).

Constraints**:**

* PRIMARY KEY on EmployeeID: Ensures that each janitor is uniquely identified by their EmployeeID.
* NOT NULL on EquipmentAssigned: Ensures that the equipment assigned to a janitor is specified.

DML Query

INSERT INTO Janitor (EmployeeID, FirstName, LastName, EmploymentDate, Email, Phone, Salary)

VALUES

(1, 'Emily', 'Davis', '2016-11-01', 'emily.davis@library.com', '555-4040', 35000.00),

...

**Explanation**:

* Inserts data into the Janitor table.
* EmployeeID: Unique identifier for each janitor (foreign key from the Employee table).
* FirstName & LastName: Janitor's name.
* EmploymentDate: Date when the janitor started working.
* Email & Phone: Contact details of the janitor.
* Salary: Annual salary of the janitor.
* A list of cleaning supplies

  Description automatically generated

Loan Table

CREATE TABLE Loan (

LoanID INT PRIMARY KEY,

LoanDate DATE NOT NULL,

ReturnDate DATE,

LoanStatus VARCHAR(20) CHECK (LoanStatus IN ('Active', 'Returned', 'Overdue')),

BookID INT ,

MemberID INT ,

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (MemberID) REFERENCES Member(MemberID)

);

Referencing**:**

* BookID is a foreign key that references Book(BookID).
* MemberID is a foreign key that references Member(MemberID).

Constraints**:**

* PRIMARY KEY on LoanID: Ensures each loan record has a unique identifier.
* NOT NULL on LoanDate: Ensures a loan date is provided.
* CHECK on LoanStatus: Restricts the possible values of loan status to 'Active', 'Returned', or 'Overdue'.
* NOT NULL on BookID and MemberID: Ensures that both the book and member related to the loan are specified.

DML

INSERT INTO Loan (LoanID, LoanDate, DueDate, ReturnDate, MemberID, BookID)

VALUES

(1, '2023-05-01', '2023-05-15', '2023-05-10', 1, 1),

...

**Explanation**:

* Inserts data into the Loan table.
* LoanID: Unique identifier for each loan transaction.
* LoanDate: The date when the book was borrowed.
* DueDate: The date by which the book should be returned.
* ReturnDate: The date the book was actually returned (if applicable).
* MemberID: Links the loan to a member (foreign key from the Member table).
* BookID: Links the loan to a book (foreign key from the Book table).
* A screenshot of a table

  Description automatically generated

Fine Table (Weak Entity)

CREATE TABLE Fine (

LoanID INT PRIMARY KEY,

Amount DECIMAL(10, 2) CHECK (Amount >= 0),

FOREIGN KEY (LoanID) REFERENCES Loan(LoanID)

);

Referencing**:**

* LoanID is a foreign key that references Loan(LoanID).

Constraints**:**

* PRIMARY KEY on LoanID: Ensures each fine is uniquely tied to a specific loan.
* CHECK on Amount: Ensures the fine amount is non-negative.

DML Query

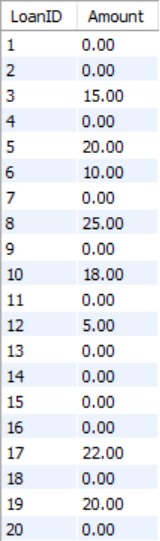
INSERT INTO Fine (FineID, Amount, DateIssued, MemberID, LoanID)

VALUES

(1, 5.00, '2023-05-20', 1, 1),

...

**Explanation**:

* Inserts data into the Fine table.
* FineID: Unique identifier for each fine.
* Amount: The amount of the fine (typically based on overdue days).
* DateIssued: The date the fine was issued.
* MemberID: Links the fine to a member (foreign key from the Member table).
* LoanID: Links the fine to a specific loan (foreign key from the Loan table).
* 

Library Floor Table

CREATE TABLE LibraryFloor (

FloorID INT PRIMARY KEY,

FloorName VARCHAR(50),

Location VARCHAR(100)

);

Referencing**:**

This table does not have any foreign key references.

Constraints**:**

* + PRIMARY KEY on FloorID: Ensures each floor has a unique identifier.

DML Query

INSERT INTO LibraryFloor (FloorID, FloorName, FloorLevel)

VALUES

(1, 'Main Floor', 1),

...

**Explanation**:

* Inserts data into the LibraryFloor table.
* FloorID: Unique identifier for each floor.
* FloorName: Name or description of the floor (e.g., Main Floor, Fiction, etc.).
* FloorLevel: The level number of the floor in the library (e.g., 1 for first floor).
* A table with text on it

  Description automatically generated

Book Copy Table (Weak Entity)

CREATE TABLE BookCopy (

CopyID INT PRIMARY KEY,

Barcode VARCHAR(50) UNIQUE ,

BookID INT NOT NULL,

FOREIGN KEY (BookID) REFERENCES Book(BookID)

);

Referencing**:**

* BookID is a foreign key that references Book(BookID).

Constraints**:**

* PRIMARY KEY on CopyID: Ensures each book copy has a unique identifier.
* NOT NULL on BookID: Ensures that each book copy is linked to a valid book.
* UNIQUE on Barcode: Ensures each book copy has a unique barcode.

DML

INSERT INTO BookCopy (CopyID, BookID, Condition, AvailabilityStatus)

VALUES

(1, 1, 'Good', 'Available'),

...

**Explanation**:

* Inserts data into the BookCopy table.
* CopyID: Unique identifier for each physical copy of a book.
* BookID: Links the copy to a specific book (foreign key from the Book table).
* Condition: Describes the physical condition of the book (e.g., Good, Fair, Damaged).
* AvailabilityStatus: Indicates if the book is available for loan (e.g., Available, Checked Out).
* A table of numbers and letters

  Description automatically generated

Private Study Room Table

CREATE TABLE PrivateStudyRoom (

RoomID INT PRIMARY KEY,

RoomNumber VARCHAR(10),

Capacity INT CHECK (Capacity > 0),

AvailabilityStatus VARCHAR(20) CHECK (AvailabilityStatus IN ('Available', 'Occupied', 'Reserved')),

ReservationsCount INT DEFAULT 0

);

Referencing**:**

This table does not have any foreign key references.

Constraints**:**

* PRIMARY KEY on RoomID: Ensures each private study room has a unique identifier.
* CHECK on Capacity: Ensures that the capacity is positive.
* CHECK on AvailabilityStatus: Ensures the availability status is one of 'Available', 'Occupied', or 'Reserved'.
* DEFAULT on ReservationsCount: Ensures that if no reservations are made, the count starts at 0.

DML

INSERT INTO PrivateStudyRoom (RoomID, RoomNumber, Capacity, Status, ReservedBy)

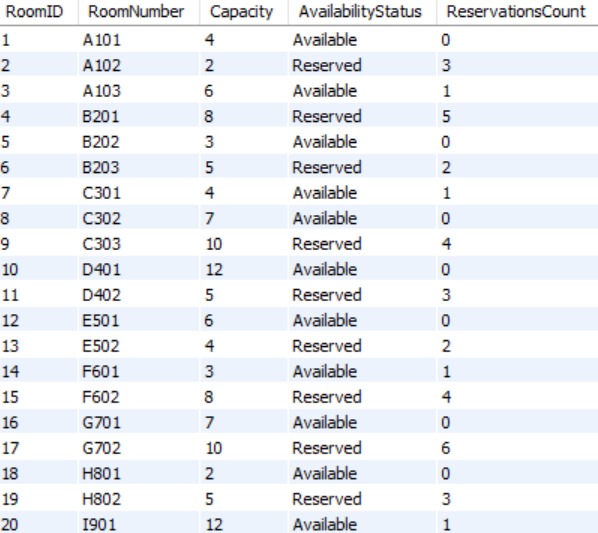
VALUES

(1, 'PSR101', 4, 'Available', NULL),

(2, 'PSR102', 2, 'Reserved', 1),

...

**Explanation**:

* Inserts data into the PrivateStudyRoom table.
* RoomID: Unique identifier for each study room.
* RoomNumber: Designation or number of the room.
* Capacity: Maximum number of people allowed in the room.
* Status: Current status of the room (e.g., Available, Reserved).
* ReservedBy: The member who has reserved the room, if applicable (foreign key from the Member table).
* 

Reservation Table

CREATE TABLE Reservation (

ReservationID INT PRIMARY KEY,

ReservationDate DATE ,

ExpirationDate DATE ,

MemberID INT ,

BookID INT,

CopyID INT,

RoomID INT,

FOREIGN KEY (MemberID) REFERENCES Member(MemberID),

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (CopyID) REFERENCES BookCopy(CopyID),

FOREIGN KEY (RoomID) REFERENCES PrivateStudyRoom(RoomID)

);

Referencing**:**

* MemberID is a foreign key that references Member(MemberID).
* BookID is a foreign key that references Book(BookID).
* CopyID is a foreign key that references BookCopy(CopyID).
* RoomID is a foreign key that references PrivateStudyRoom(RoomID).

Constraints**:**

* PRIMARY KEY on ReservationID: Ensures each reservation has a unique identifier.
* NOT NULL on MemberID, BookID, CopyID, and RoomID: Ensures that each reservation is tied to a valid member, book, copy, and room.

DML

INSERT INTO Reservation (ReservationID, MemberID, BookID, ReservationDate, Status)

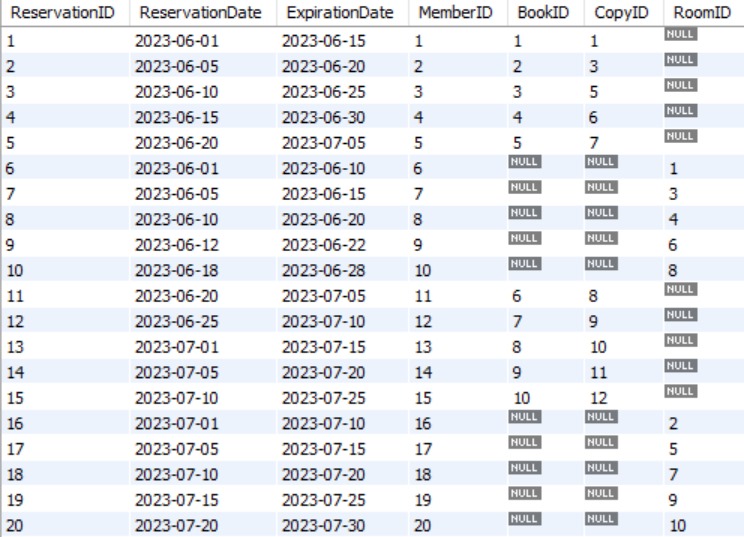
VALUES

(1, 1, 1, '2023-06-15', 'Reserved'),

(2, 2, 2, '2023-07-01', 'Completed'),

...

**Explanation**:

* Inserts data into the Reservation table.
* ReservationID: Unique identifier for each reservation.
* MemberID: Links the reservation to a member (foreign key from the Member table).
* BookID: Links the reservation to a specific book (foreign key from the Book table).
* ReservationDate: Date when the reservation was made.
* Status: Status of the reservation (e.g., Reserved, Completed, Cancelled).
* 

Review Table

CREATE TABLE Review (

ReviewID INT PRIMARY KEY,

Rating INT CHECK (Rating BETWEEN 1 AND 5),

Comments TEXT,

ReviewDate DATE ,

BookID INT NOT NULL,

MemberID INT NOT NULL,

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (MemberID) REFERENCES Member(MemberID)

);

Referencing**:**

* BookID is a foreign key that references Book(BookID).
* MemberID is a foreign key that references Member(MemberID).

Constraints**:**

* PRIMARY KEY on ReviewID: Ensures each review has a unique identifier.
* CHECK on Rating: Ensures the rating is between 1 and 5.
* NOT NULL on BookID and MemberID: Ensures that each review is tied to a valid book and member.

DML

INSERT INTO Review (ReviewID, MemberID, BookID, ReviewDate, Rating, Comments)

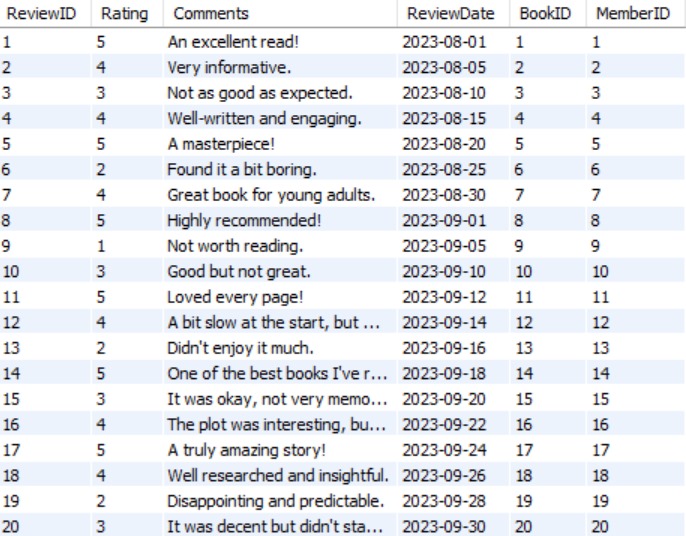
VALUES

(1, 1, 1, '2023-07-01', 5, 'An amazing dystopian novel!'),

(2, 2, 3, '2023-08-15', 4, 'Great read, but a bit slow in the middle.'),

...

**Explanation**:

* Inserts data into the Review table.
* ReviewID: Unique identifier for each review.
* MemberID: Links the review to a member (foreign key from the Member table).
* BookID: Links the review to a specific book (foreign key from the Book table).
* ReviewDate: Date the review was written.
* Rating: Rating given by the member (e.g., 1 to 5 stars).
* Comments: Textual feedback or review provided by the member.
* 

Supplier Table

CREATE TABLE Supplier (

SupplierID INT PRIMARY KEY,

SupplierName VARCHAR(100) ,

ContactPerson VARCHAR(100),

Phone VARCHAR(15),

Email VARCHAR(100),

Address\_Street VARCHAR(100),

Address\_City VARCHAR(50),

Address\_State VARCHAR(50),

Address\_ZipCode VARCHAR(10)

);

Referencing**:**

This table does not have any foreign key references.

Constraints**:**

* PRIMARY KEY on SupplierID: Ensures each supplier has a unique identifier.

DML

INSERT INTO Supplier (SupplierID, SupplierName, ContactName, Address\_Street, Address\_City, Address\_State, Address\_ZipCode, Email, Phone)

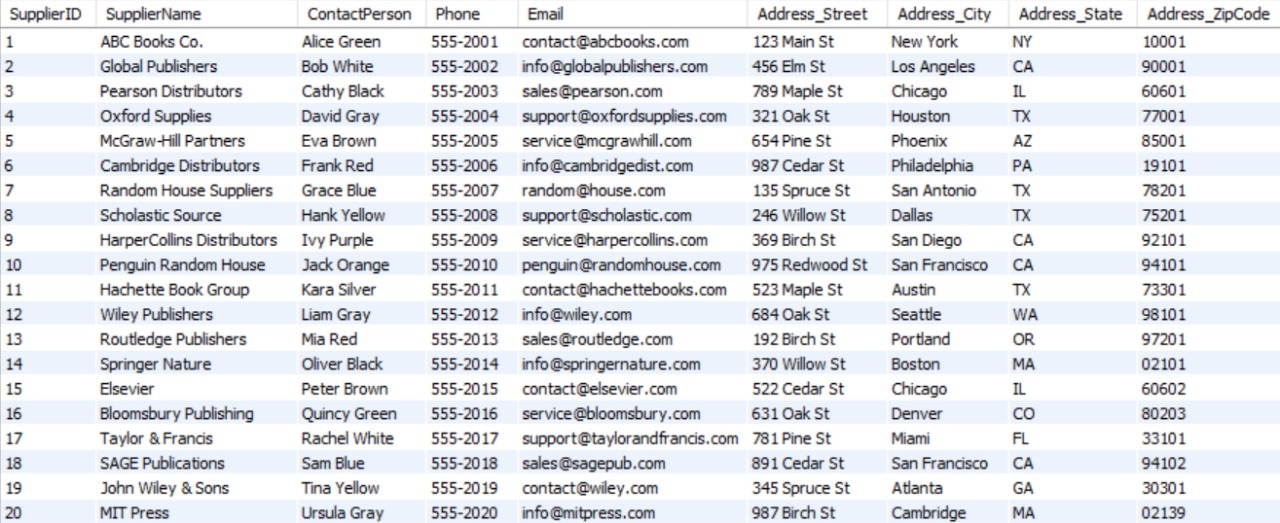
VALUES

(1, 'Book Supply Co.', 'James White', '456 Supply Rd', 'Chicago', 'IL', '60601', 'contact@booksupply.com', '555-5050'),

(2, 'Library Goods Ltd.', 'Anna Green', '789 Commerce Blvd', 'Los Angeles', 'CA', '90001', 'contact@librarygoods.com', '555-6060'),

...

**Explanation**:

* Inserts data into the Supplier table.
* SupplierID: Unique identifier for each supplier.
* SupplierName: Name of the supplier company.
* ContactName: Name of the primary contact person at the supplier.
* Address\_Street, Address\_City, Address\_State, Address\_ZipCode: Supplier's address.
* Email & Phone: Supplier's contact details.
* 

libOrder Table

CREATE TABLE libOrder (

libOrderID INT PRIMARY KEY,

libOrderDate DATE ,

ExpectedDeliveryDate DATE,

OrderStatus VARCHAR(20) CHECK (OrderStatus IN ('Pending', 'Delivered', 'Cancelled')),

TotalCost DECIMAL(10, 2) CHECK (TotalCost >= 0)

);

Referencing**:**

This table does not have any foreign key references.

Constraints**:**

* PRIMARY KEY on libOrderID: Ensures each order has a unique identifier.
* CHECK on OrderStatus: Restricts the possible values of order status to 'Pending', 'Delivered', or 'Cancelled'.
* CHECK on TotalCost: Ensures the total cost is non-negative.

DML

INSERT INTO libOrder (OrderID, OrderDate, SupplierID, TotalAmount)

VALUES

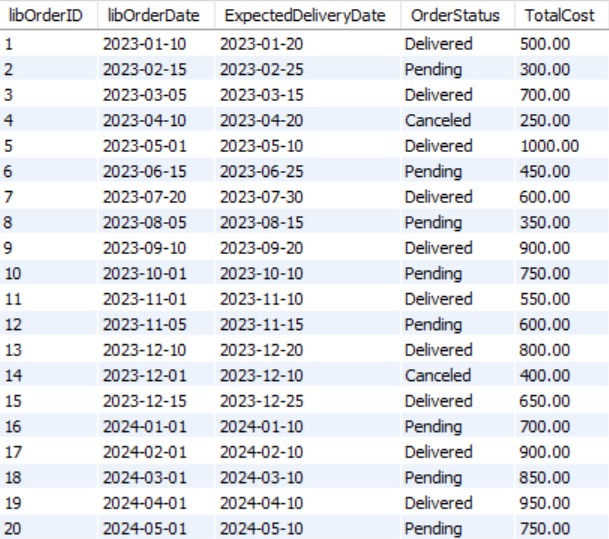
(1, '2023-06-10', 1, 1000.00),

(2, '2023-07-12', 2, 1200.00),

...

**Explanation**:

* Inserts data into the libOrder table.
* OrderID: Unique identifier for each order.
* OrderDate: Date when the order was placed.
* SupplierID: Links the order to a supplier (foreign key from the Supplier table).
* TotalAmount: The total value of the order.



Recording Table

CREATE TABLE Recording (

RecordingID INT PRIMARY KEY,

Title VARCHAR(200) NOT NULL,

RecodingType VARCHAR(50) ,

ArchivalDate DATE ,

);

Referencing**:**

This table does not reference any other tables.

Constraints**:**

* PRIMARY KEY on RecordingID: Ensures each recording has a unique identifier.
* NOT NULL on Title: Ensures a title is provided for each recording.

DML

INSERT INTO Recording (RecordingID, MemberID, BookID, DateRecorded)

VALUES

(1, 1, 1, '2023-06-12'),

(2, 2, 3, '2023-07-14'),

...

**Explanation**:

* Inserts data into the Recording table.
* RecordingID: Unique identifier for each recording.
* MemberID: Links the recording to a member (foreign key from the Member table).
* BookID: Links the recording to a book (foreign key from the Book table).
* DateRecorded: Date the recording was made
* 

Cleans Relationship Table

CREATE TABLE Cleans (

EmployeeID INT,

FloorID INT,

PRIMARY KEY (EmployeeID, FloorID),

FOREIGN KEY (EmployeeID) REFERENCES Janitor(EmployeeID),

FOREIGN KEY (FloorID) REFERENCES LibraryFloor(FloorID)

);

Referencing**:**

* EmployeeID is a foreign key that references Janitor(EmployeeID).
* FloorID is a foreign key that references LibraryFloor(FloorID).

Constraints**:**

* PRIMARY KEY on EmployeeID and FloorID: Ensures that each employee is assigned to clean a specific floor uniquely.

DML

INSERT INTO Cleans (EmployeeID, FloorID)

VALUES

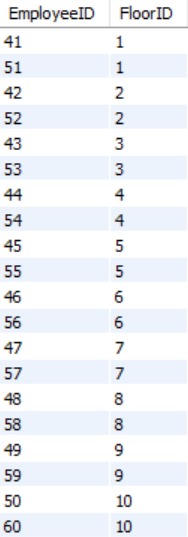
(41, 1), -- Janitor 41 cleans Floor 1

(42, 2), -- Janitor 42 cleans Floor 2

...

(60, 10); -- Janitor 60 cleans Floor 10

Purpose: Inserts data into the Cleans table.  
**Column Explanations**:

* **EmployeeID**: Refers to the ID of the janitor (foreign key from Employee table).
* **FloorID**: Refers to the ID of the floor being cleaned (foreign key from Library Floor table).
* 

Works On Relationship Table (Librarian-Library Floor)

CREATE TABLE WorksOn (

EmployeeID INT,

FloorID INT,

PRIMARY KEY (EmployeeID, FloorID),

FOREIGN KEY (EmployeeID) REFERENCES Librarian(EmployeeID),

FOREIGN KEY (FloorID) REFERENCES LibraryFloor(FloorID)

);

Referencing**:**

* EmployeeID is a foreign key that references Librarian(EmployeeID).
* FloorID is a foreign key that references LibraryFloor(FloorID).

Constraints**:**

* PRIMARY KEY on EmployeeID and FloorID: Ensures that each librarian works on a specific floor uniquely.

DML

INSERT INTO WorksOn (EmployeeID, FloorID)

VALUES

(1, 1), -- Librarian 1 works on Floor 1

(2, 2), -- Librarian 2 works on Floor 2

...

(20, 10); -- Librarian 20 works on Floor 10

**Purpose**: Inserts data into the WorksOn table.  
**Column Explanations**:

* **EmployeeID**: Refers to the ID of the librarian (foreign key from Employee table).
* **FloorID**: Refers to the ID of the floor where the librarian works (foreign key from Library Floor table).
* A white rectangular table with black numbers

  Description automatically generated

Contains Relationship Table (Library Floor - Book Copy)

CREATE TABLE Contain (

FloorID INT,

BookID INT,

CopyID INT,

PRIMARY KEY (FloorID, BookID, CopyID),

FOREIGN KEY (FloorID) REFERENCES LibraryFloor(FloorID),

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (CopyID) REFERENCES BookCopy(CopyID)

);

Referencing**:**

* FloorID is a foreign key that references LibraryFloor(FloorID).
* BookID is a foreign key that references Book(BookID).
* CopyID is a foreign key that references BookCopy(CopyID).

Constraints**:**

* PRIMARY KEY on FloorID, BookID, and CopyID: Ensures each floor contains a specific book copy uniquely.

DML

INSERT INTO Contain (FloorID, BookID, CopyID)

VALUES

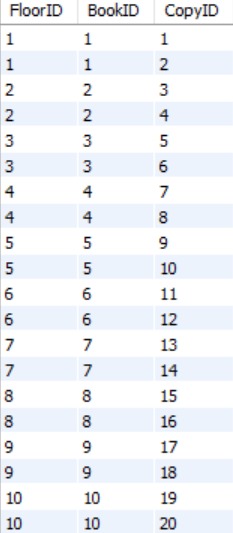
(1, 1, 1), -- Book '1984' (ID 1) Copy 1 on Floor 1

(1, 1, 2), -- Book '1984' (ID 1) Copy 2 on Floor 1

...

(10, 10, 20); -- Book 'Beloved' (ID 10) Copy 20 on Floor 10

**Purpose**: Inserts data into the Contain table.  
**Column Explanations**:

* **FloorID**: Refers to the floor where the book copy is stored (foreign key from Library Floor table).
* **BookID**: Refers to the book being stored (foreign key from Book table).
* **CopyID**: Refers to the specific copy of the book (foreign key from Book Copy table).
* 

Supplies Relationship Table (Supplier-libOrder-Book)

CREATE TABLE Supplies (

SupplierID INT,

libOrderID INT,

BookID INT,

PRIMARY KEY (SupplierID, libOrderID, BookID),

FOREIGN KEY (SupplierID) REFERENCES Supplier(SupplierID),

FOREIGN KEY (libOrderID) REFERENCES libOrder(libOrderID),

FOREIGN KEY (BookID) REFERENCES Book(BookID)

);

Referencing**:**

* SupplierID is a foreign key that references Supplier(SupplierID).
* libOrderID is a foreign key that references libOrder(libOrderID).
* BookID is a foreign key that references Book(BookID).

Constraints**:**

* PRIMARY KEY on the combination of SupplierID, libOrderID, and BookID: This ensures that each entry in the table is unique for a particular supplier, order, and book combination.
* There are no additional NOT NULL constraints since all foreign key references are required by their respective tables, making the relationship itself mandatory.

DML

INSERT INTO Supplies (SupplierID, libOrderID, BookID)

VALUES

(1, 1, 1), -- Supplier 1 provides Book 1 in libOrder 1

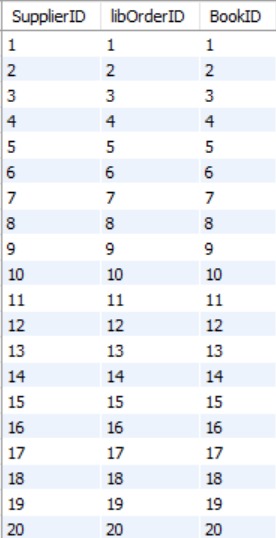
(2, 2, 2), -- Supplier 2 provides Book 2 in libOrder 2

...

(20, 20, 20); -- Supplier 20 provides Book 20 in libOrder 20

**Purpose**: Inserts data into the Supplies table.  
**Column Explanations**:

* **SupplierID**: Refers to the supplier providing the book (foreign key from Supplier table).
* **libOrderID**: Refers to the order placed for the book (foreign key from libOrder table).
* **BookID**: Refers to the book being supplied (foreign key from Book table).



Monitors Aggregation Relationship Table

CREATE TABLE Monitors (

LoanID INT,

MemberID INT,

EmployeeID INT,

MonitoringDate DATE ,

ActionTaken VARCHAR(100),

Remarks TEXT,

PRIMARY KEY (LoanID, MemberID, EmployeeID),

FOREIGN KEY (LoanID) REFERENCES Loan(LoanID),

FOREIGN KEY (MemberID) REFERENCES Member(MemberID),

FOREIGN KEY (EmployeeID) REFERENCES Librarian(EmployeeID)

);

Referencing**:**

* LoanID is a foreign key that references Loan(LoanID).
* MemberID is a foreign key that references Member(MemberID).
* EmployeeID is a foreign key that references Librarian(EmployeeID).

Constraints**:**

* PRIMARY KEY on the combination of LoanID, MemberID, and EmployeeID: This ensures that the combination of loan, member, and employee is unique. It effectively tracks which librarian is monitoring a specific loan transaction involving a particular member.
* NOT NULL constraints ensure that the references to LoanID, MemberID, and EmployeeID are mandatory, linking each monitoring record to valid loan, member, and librarian entries.
* Remarks and ActionTaken can be NULL since they represent optional commentary on the actions taken during monitoring.

DML

INSERT INTO Monitors (LoanID, MemberID, EmployeeID, MonitoringDate, ActionTaken, Remarks)

VALUES

(1, 1, 1, '2024-11-01', 'Returned', 'Loan returned on time by Alice'),

(2, 2, 2, '2024-11-02', 'Active', 'Loan is still active, Bob has the book'),

...

(20, 20, 20, '2024-11-20', 'Active', 'Tina has the book, loan status active');

**Purpose**: Inserts data into the Monitors table.  
**Column Explanations**:

* **LoanID**: Refers to the loan being monitored (foreign key from Loan table).
* **MemberID**: Refers to the ID of the member who borrowed the book (foreign key from Member table).
* **EmployeeID**: Refers to the employee monitoring the loan (foreign key from Employee table).
* **MonitoringDate**: Date when the loan was monitored.
* **ActionTaken**: Status or action taken during monitoring (e.g., Returned, Overdue, Active).
* **Remarks**: Additional information about the monitoring status.

Let me know if you want any specific details or formatting adjustments!

A screenshot of a computer

Description automatically generated

Basic SQL Queries

**Query 1:**

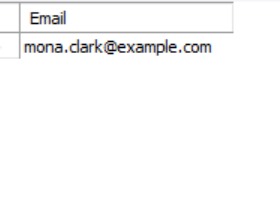
SELECT M.Email

FROM Reservation R, Member M

WHERE M.MemberID=R.MemberID AND R.CopyID=10;

* **Purpose**: Retrieves the email of members who have reserved a specific book copy (CopyID = 10).
* **Usefulness**: Allows library administrators to contact members about specific reservations, such as availability or overdue notices.

**Result:**



**Query 2:**

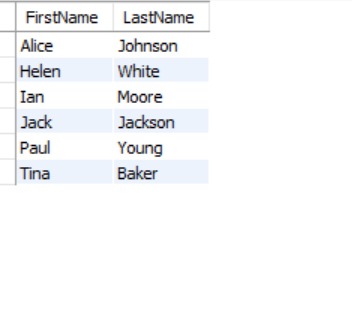
SELECT FirstName, LastName

FROM Reservation NATURAL JOIN Member

WHERE RoomID%2=0;

* **Purpose**: Finds members who reserved rooms with even-numbered IDs.
* **Usefulness**: Useful for analytics on how certain room assignments are distributed and potentially for scheduling based on patterns.

**Result:**



**Query 3:**

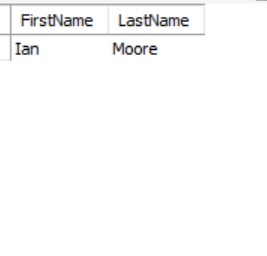
SELECT M.FirstName, M.LastName

FROM Reservation R JOIN Member M USING (MemberID)

WHERE R.RoomID=6;

* **Purpose**: Retrieves the names of members who reserved Room 6.
* **Usefulness**: Helps in managing private study room reservations or resolving conflicts regarding Room 6 usage.

**Result:**



Query 4:

SELECT E.EmployeeID, E.Salary

FROM Cleans C JOIN Employee E ON (C.EmployeeID=E.EmployeeID);

* **Purpose**: Displays employee IDs and their salaries for janitors responsible for cleaning library floors.
* **Usefulness**: Enables payroll management for janitorial staff and cross-verification of cleaning responsibilities.

**Result:**

A screenshot of a spreadsheet

Description automatically generated

**Query 5:**

SELECT E.EmployeeID

FROM Employee E1, Employee E2

WHERE E1.Salary>E2.Salary AND E2.EmployeeID=1;

* **Purpose**: Finds the employees earning more than Employee 1.
* **Usefulness**: Useful for payroll analysis or identifying high earners compared to a specific benchmark employee.

**Result:**

A screenshot of a calendar

Description automatically generated

**Query 6:**

SELECT DISTINCT R.MemberID

FROM Review R

WHERE R.Rating=5;

* **Purpose**: Identifies members who gave a perfect (5-star) rating in reviews.
* **Usefulness**: Helps in marketing efforts or personalized recommendations to members with highly positive feedback.

**Result:**

**A screenshot of a computer

Description automatically generated**

**Query 7:**

SELECT R.RecordingID, R.Title

FROM Recording R

WHERE R.Title LIKE '%Quantum %';

* **Purpose**: Searches for recordings with "Quantum" in their titles.
* **Usefulness**: Useful for members or staff looking for recordings on specific topics or titles containing targeted keywords.

**Result:**

**A screenshot of a computer

Description automatically generated**

**Query 8:**

SELECT R.RecordingID, R.Title

FROM Recording R

ORDER BY R.Title ASC;

* **Purpose**: Lists all recordings, sorted alphabetically by title.
* **Usefulness**: Helps users or staff to browse recordings in an organized, easy-to-search manner.

**Result:**

A screenshot of a computer

Description automatically generated

**Query 9:**

SELECT C.EmployeeID

FROM Cleans C

WHERE FloorID=10

UNION

SELECT W.EmployeeID

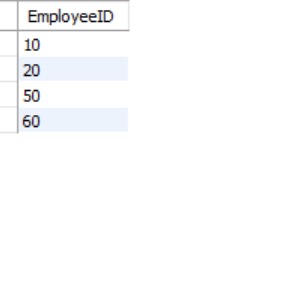
FROM WorksOn W

WHERE FloorID=10

ORDER BY EmployeeID;

* **Purpose**: Combines janitors and librarians who work or clean Floor 10 into a single list.
* **Usefulness**: Provides a unified view of all staff associated with a specific floor for floor-specific management or coordination.

**Result:**



**Query 10 (INTERSECT)**

-- SELECT R.MemberID

-- FROM Reservation R

-- WHERE R.RoomID IS NULL

-- INTERSECT

-- SELECT R.MemberID

-- FROM Reservation R

-- WHERE R.BookID IS NULL AND R.CopyID IS NULL;

* **Purpose**: Intended to find members who reserved a room and a book (intersection of conditions).
* **MySQL Limitation**: **INTERSECT** is not supported in MySQL. Equivalent functionality can be achieved using INNER JOIN or WHERE EXISTS.

***SELECT DISTINCT R1.MemberID***

***FROM Reservation R1***

***INNER JOIN Reservation R2***

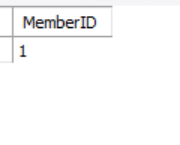
***ON R1.MemberID = R2.MemberID***

***WHERE R1.RoomID IS NULL***

***AND R2.BookID IS NULL***

***AND R2.CopyID IS NULL;***

**Result:**

****

**Query 11 (EXCEPT)**

-- SELECT E.EmployeeID

-- FROM Employee E

-- EXCEPT

-- SELECT E1.EmployeeID

-- FROM Employee E1, Employee E2

-- WHERE E1.EmployeeID < E2.EmployeeID;

* **Purpose**: Intended to find the id of the highest paid employee.
* **MySQL Limitation**: **EXCEPT** is not supported in MySQL. Equivalent functionality can be achieved using NOT IN or LEFT JOIN with NULL checking.

SELECT E.EmployeeID

FROM Employee E

WHERE E.EmployeeID NOT IN (

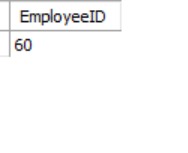
SELECT E1.EmployeeID

FROM Employee E1, Employee E2

WHERE E1.EmployeeID < E2.EmployeeID

);

**Result:**



**Query 12:**

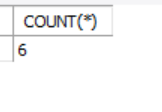
*SELECT COUNT(\*)*

*FROM libOrder L*

*WHERE L.OrderStatus='Delivered' AND L.TotalCost>=700.0;*

* **Purpose**: Groups all reviews by their rating and counts how many reviews each rating received.
* **Usefulness**: Provides insights into user feedback distribution, helping the library gauge overall user satisfaction and identify areas for improvement.

**Result:**

****

**Query 13:**

*SELECT R.Rating, COUNT(\*)*

*FROM Review R*

*GROUP BY R.Rating;*

* **Purpose**: Groups all reviews by their rating and counts how many reviews each rating received.
* **Usefulness**: Provides insights into user feedback distribution, helping the library gauge overall user satisfaction and identify areas for improvement.

**Result**:

**A screenshot of a computer

Description automatically generated**

**Query 14:**

*SELECT B.Genre, COUNT(B.BookID) AS BookCount*

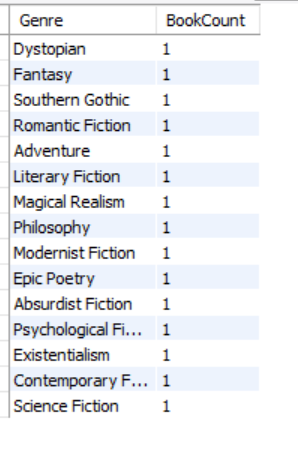
*FROM Book B*

*GROUP BY B.Genre*

*HAVING COUNT(B.BookID) < 4;*

* **Purpose**: Give us all genras of books we have and how many book we have in each genra, such that we have less than 4 books in.
* **Usefulness**: Provides insights about the number of books we have on each genra, so that we know that we should supply more books on these genras.

**Result**:



Advanced SQL Qeueries

**Query 1:**

*SELECT Title*

*FROM Book*

*WHERE BookID IN (*

*SELECT Loan.BookID*

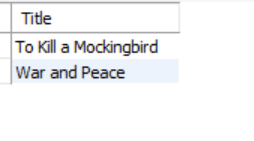
*FROM Loan*

*WHERE LoanStatus = 'Overdue'*

*);*

* **Purpose**: Retrieve the titles of books that are overdue for return by checking the Loan table for overdue loans.
* **Detailed Explanation**:
  + Inner Query: This query scans the Loan table and finds all BookID values where the LoanStatus is 'Overdue'. The result is a list of BookIDs associated with overdue loans.
  + Outer Query: The outer query matches the BookID values returned by the inner query with the Book table. Using the IN clause ensures that only books present in the inner query results are retrieved.

**Result**:



**Query 2:**

*SELECT MemberID, FirstName, LastName*

*FROM Member*

*WHERE NOT EXISTS (*

*SELECT BookID*

*FROM Book*

*WHERE AuthorID = 1*

*EXCEPT (*

*SELECT Loan.BookID*

*FROM Loan*

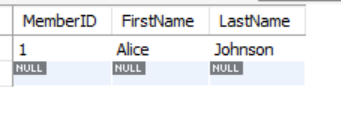
*WHERE Loan.MemberID = Member.MemberID*

*)*

*);*

* **Purpose**: Identify members who have borrowed all the books written by a specific author (AuthorID = 1).
* **Detailed Explanation:**
  + **Inner 1:** This query retrieves the BookIDs for all books written by the author with AuthorID = 1
  + **Inner 2:** This query finds all the books loaned by a specific member (MemberID) from the Loan table
  + Set Comparison Using EXCEPT:EXCEPT subtracts the set of books borrowed by the member (Inner Query 2) from the set of books written by the author (Inner Query 1).If the result of the subtraction is empty, it means the member has borrowed all books written by the author
  + **NOT EXISTS:** NOT EXISTS (...) ensures that only members for whom the subtraction result is empty (i.e., they have borrowed all books by the author) are included in the result.
  + **Outer Query:** The outer query retrieves the MemberID, FirstName, and LastName of such members from the Member table

**Result**:



**Query 3 with Set Cardinality (Nested Query):**

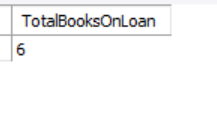
SELECT COUNT(DISTINCT BookID) AS TotalBooksOnLoan

FROM Loan

WHERE LoanStatus = 'Active';

* **Purpose**: Count the total number of unique books currently on loan.
* **Explanation:** Filter the Loan table for records where loans are still ongoing, include only those where LoanStatus = 'Active', and count unique books to ensure only unique books are considered, even if the same book has been loaned multiple times.

**Results**:



**-Query 4 with Two or More Nesting Levels:**

SELECT DISTINCT MemberID, FirstName, LastName

FROM Member

WHERE MemberID IN (

SELECT Review.MemberID

FROM Review

WHERE BookID IN (

SELECT Loan.BookID

FROM Loan

WHERE LoanStatus = 'Overdue'

)

);

Purpose:  
Find members who have written reviews for books that are currently overdue.

Detailed Explanation:

Innermost Query (Books on Overdue Loan):

SELECT Loan.BookID FROM Loan WHERE LoanStatus = 'Overdue':

Finds all BookIDs associated with loans that have a LoanStatus of 'Overdue'.

Second-Level Query (Members Who Reviewed Overdue Books):

SELECT Review.MemberID FROM Review WHERE BookID IN (...):

Matches the BookID values returned by the innermost query to the BookID column in the Review table.

Retrieves the MemberIDs of members who reviewed these books.

Outer Query:

SELECT DISTINCT MemberID, FirstName, LastName FROM Member WHERE MemberID IN (...):

Uses the IN clause to check which members from the Member table match the MemberIDs returned by the second-level query.

DISTINCT ensures duplicate rows are eliminated.

Result:

Outputs the MemberID, FirstName, and LastName of members who have reviewed overdue books.

**A screenshot of a computer

Description automatically generated**

**Query 5 with SQL Division (NOT EXISTS … EXCEPT)**

SELECT SupplierID, SupplierName

FROM Supplier

WHERE NOT EXISTS (

SELECT BookID

FROM Book

EXCEPT (

SELECT Supplies.BookID

FROM Supplies

WHERE Supplies.SupplierID = Supplier.SupplierID

)

);

Purpose:  
Find suppliers that supply all the books available in the Book table.

Detailed Explanation:

Inner Query 1 (All Books):

SELECT BookID FROM Book:

Retrieves the BookID values for all books in the Book table.

Inner Query 2 (Books Supplied by a Supplier):

SELECT Supplies.BookID FROM Supplies WHERE Supplies.SupplierID = Supplier.SupplierID:

Finds all BookIDs supplied by a particular supplier.

Set Comparison Using EXCEPT:

EXCEPT subtracts the books supplied by the supplier (Inner Query 2) from all books in the Book table (Inner Query 1).

If the subtraction result is empty, it means the supplier supplies all books in the Book table.

NOT EXISTS Clause:

Ensures that only suppliers meeting the "supplies all books" condition are included.

Outer Query:

Retrieves the SupplierID and SupplierName of such suppliers from the Supplier table.

**Results:**

**A screenshot of a computer

Description automatically generated**

**Query 6 with Nested Query in the FROM Clause**

SELECT AVG(LateDays) AS AvgLateDays

FROM (

SELECT LoanID, DATEDIFF(ReturnDate, LoanDate) AS LateDays

FROM Loan

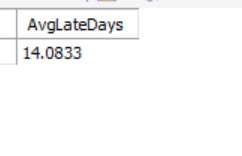
WHERE LoanStatus = 'Returned'

) AS LoanDurations;

**Purpose**:  
Calculate the average number of days it takes to return books after borrowing, for loans with a status of "Returned".

**Detailed Explanation**:

1. **Inner Query (Loan Durations)**:
   * SELECT LoanID, DATEDIFF(ReturnDate, LoanDate) AS LateDays FROM Loan WHERE LoanStatus = 'Returned':
     + Filters the Loan table to include only loans that have been returned (LoanStatus = 'Returned').
     + Calculates the number of days between the LoanDate and ReturnDate for each loan using DATEDIFF, and labels this difference as LateDays.
2. **Outer Query (Average Calculation)**:
   * SELECT AVG(LateDays) AS AvgLateDays:
     + Computes the average of the LateDays values returned by the inner query.
   * The FROM clause treats the inner query result as a temporary table named LoanDurations.
3. **Result**:
   * Outputs a single value (AvgLateDays), representing the average number of days it takes to return books.

****

**Query 7 with Nested Query in the SELECT Clause**

SELECT MemberID, FirstName, LastName,

(SELECT COUNT(LoanID)

FROM Loan

WHERE Loan.MemberID = Member.MemberID) AS TotalLoans

FROM Member;

**Purpose**:  
Retrieve the details of all members along with the total number of loans each member has made.

**Detailed Explanation**:

1. **Nested Query in SELECT Clause**:
   * (SELECT COUNT(LoanID) FROM Loan WHERE Loan.MemberID = Member.MemberID):
     + Counts the number of LoanIDs in the Loan table where the MemberID matches the MemberID from the outer query.
     + This calculates the total number of loans made by each member.
2. **Outer Query**:
   * Retrieves the MemberID, FirstName, and LastName of each member from the Member table.
   * The nested query adds a new column (TotalLoans) that shows the total number of loans for each member.
3. **Result**:
   * Returns all members with their personal details and total loan counts.

A table of names

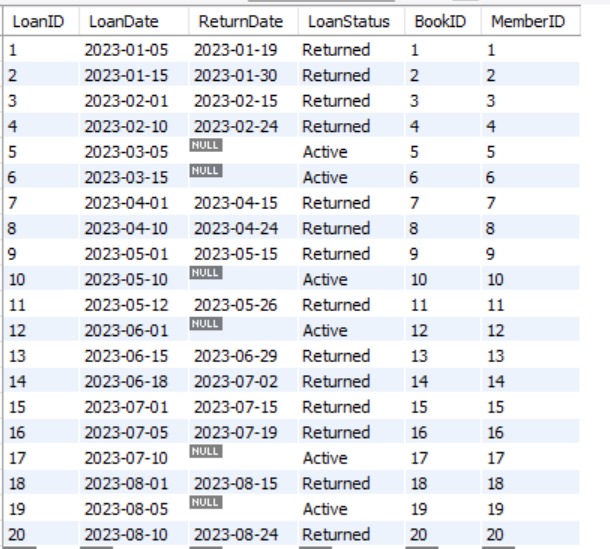
Description automatically generated

**Purpose**

**This query updates the LoanStatus column in the Loan table based on the current date (CURDATE()) and the value of the ReturnDate. It ensures that the status of each loan dynamically reflects its current state: active, returned, or overdue.**

**Detailed Explanation**

1. **UPDATE Loan**
   * **Specifies the table to update: the Loan table.**
2. **SET LoanStatus = CASE ...**
   * **This dynamically evaluates and sets the LoanStatus column using a CASE statement. The CASE logic applies the following conditions:**
   * **Condition 1: WHEN ReturnDate IS NULL THEN 'Active'**
     + **If there is no return date (ReturnDate IS NULL), the book is still loaned out and is considered 'Active'.  
       Example: LoanID 5 (ReturnDate = NULL) → Status is 'Active'.**
   * **Condition 2: WHEN ReturnDate <= CURDATE() THEN 'Returned'**
     + **If the ReturnDate is on or before today's date (CURDATE()), the loan is marked as 'Returned' because the book has been returned.  
       Example: LoanID 1 (ReturnDate = 2023-01-19) → Status is 'Returned'.**
   * **Condition 3: WHEN ReturnDate > CURDATE() THEN 'Overdue'**
     + **If the ReturnDate is later than today’s date (CURDATE()), the book is overdue and the status is set to 'Overdue'.  
       Example: If today's date is 2023-08-01 and ReturnDate is 2023-08-15, the status will be 'Overdue'.**
3. **WHERE LoanID IS NOT NULL**
   * **Ensures that all valid rows in the table are updated (excluding rows with missing LoanID values, if any).**



**Query 9 with OUTER JOIN**

SELECT Member.FirstName, Member.LastName, Loan.LoanID, Loan.LoanStatus

FROM Member

LEFT OUTER JOIN Loan

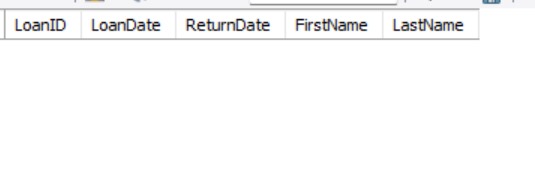
ON Member.MemberID = Loan.MemberID

WHERE Loan.LoanID IS NULL OR LoanStatus = 'Overdue';

**Purpose**:  
Find all members who either have no loans or have loans that are overdue.

**Detailed Explanation**:

1. **LEFT OUTER JOIN**:
   * FROM Member LEFT OUTER JOIN Loan ON Member.MemberID = Loan.MemberID:
     + Combines the Member table with the Loan table by matching MemberID.
     + Ensures all rows from the Member table are included, even if there is no corresponding record in the Loan table.
2. **WHERE Clause**:
   * WHERE Loan.LoanID IS NULL OR LoanStatus = 'Overdue':
     + Filters the result to include only:
       - Members who have no loans (Loan.LoanID IS NULL).
       - Members with overdue loans (LoanStatus = 'Overdue').
3. **Columns in SELECT**:
   * Displays the FirstName and LastName of members, along with their loan information (LoanID and LoanStatus), if applicable.
4. **Result**:
   * Outputs a list of members who either have overdue loans or no loans at all.



**Query 10 Assertion from Original Queries**

CREATE ASSERTION EnsureMinimumBooksAndMembers

CHECK (

(SELECT COUNT(\*) FROM Book) >= 100 AND

(SELECT COUNT(\*) FROM Member) >= 50

);

**Purpose**:  
Ensure that the library system always has at least 100 books and 50 members.

**Detailed Explanation**:

1. **COUNT Subqueries**:
   * (SELECT COUNT(\*) FROM Book) checks the total number of books in the Book table.
   * (SELECT COUNT(\*) FROM Member) checks the total number of members in the Member table.
2. **Condition**:
   * (COUNT(Book) >= 100 AND COUNT(Member) >= 50):
     + Ensures both counts meet or exceed the minimum thresholds.
3. **Assertion**:
   * Prevents any operation that would violate this condition, such as deleting too many books or members.
4. **Result**:
   * Maintains the integrity of the library database by enforcing a minimum baseline of books and members.

Result no supported by MYSQL

**Query 11 View Creation**

CREATE VIEW OverdueLoansView AS

SELECT Loan.LoanID, Loan.LoanDate, Loan.ReturnDate, Member.FirstName, Member.LastName

FROM Loan

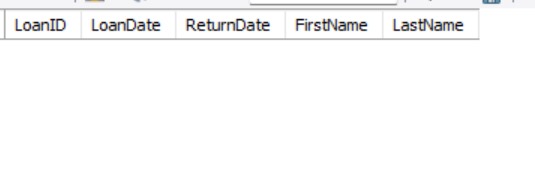
JOIN Member ON Loan.MemberID = Member.MemberID

WHERE Loan.LoanStatus = 'Overdue';

**Purpose**:  
Create a simplified and reusable view that shows information about overdue loans along with the borrowing members' names.

**Detailed Explanation**:

1. **JOIN Clause**:
   * Combines the Loan table and Member table using the condition Loan.MemberID = Member.MemberID to retrieve the details of members associated with each loan.
2. **WHERE Clause**:
   * Filters the results to include only loans with a LoanStatus of 'Overdue'. This ensures the view focuses exclusively on overdue loans.
3. **Columns Selected**:
   * Includes relevant details:
     + LoanID: Unique identifier for each overdue loan.
     + LoanDate and ReturnDate: Loan's start and expected return dates.
     + FirstName and LastName: Member details for better traceability.
4. **View Functionality**:
   * This view can be queried like a table to quickly access overdue loan data without writing the query repeatedly.
5. **Result**:
   * A permanent logical table (OverdueLoansView) that provides an easy reference for overdue loans and the members responsible.



**Query 12 Trigger**

CREATE TRIGGER UpdateBookCopiesOnLoan

AFTER INSERT ON Loan

FOR EACH ROW

BEGIN

UPDATE Book

SET CopiesAvailable = CopiesAvailable - 1

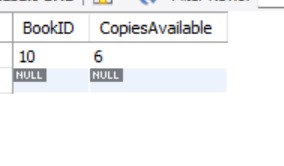
WHERE BookID = NEW.BookID;

END;

**Purpose**:  
Automatically decrement the CopiesAvailable value in the Book table whenever a new loan is created.

**Detailed Explanation**:

1. **Trigger Timing**:
   * AFTER INSERT: The trigger executes after a new record is inserted into the Loan table.
2. **Trigger Event**:
   * Watches the Loan table for new insertions to identify when a book is loaned out.
3. **Trigger Logic**:
   * Uses an UPDATE query to adjust the CopiesAvailable column of the corresponding book.
   * The NEW.BookID refers to the BookID from the newly inserted row in the Loan table.
4. **Result**:
   * Ensures the availability of book copies is always updated whenever a loan is created, maintaining data accuracy.



**Query 13 Stored Procedure**

DELIMITER //

CREATE PROCEDURE RecordLoanAndUpdateCopies(

IN p\_LoanDate DATE,

IN p\_ReturnDate DATE,

IN p\_LoanStatus VARCHAR(20),

IN p\_BookID INT,

IN p\_MemberID INT

)

BEGIN

-- Insert a new loan record

INSERT INTO Loan (LoanDate, ReturnDate, LoanStatus, BookID, MemberID)

VALUES (p\_LoanDate, p\_ReturnDate, p\_LoanStatus, p\_BookID, p\_MemberID);

-- Decrement the available copies for the book

UPDATE Book

SET CopiesAvailable = CopiesAvailable - 1

WHERE BookID = p\_BookID;

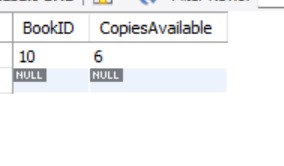
END //

DELIMITER ;

**Purpose**:  
Simplify the process of recording a new loan and updating the number of available book copies in a single operation.

**Detailed Explanation**:

1. **Procedure Input Parameters**:
   * p\_LoanDate, p\_ReturnDate, p\_LoanStatus, p\_BookID, and p\_MemberID: Inputs required to create a new loan and identify the associated book and member.
2. **Insertion Logic**:
   * The INSERT statement adds a new row to the Loan table using the provided parameters.
3. **Update Logic**:
   * The UPDATE statement adjusts the CopiesAvailable column of the Book table for the specified BookID.
4. **Result**:
   * Automates two essential tasks (creating a loan and updating book availability) into a single stored procedure.
   * Reduces redundancy and ensures the database remains consistent.

****

**Query 14 Assertion**

CREATE ASSERTION EnsureNoNegativeCopies

CHECK (

(SELECT MIN(CopiesAvailable) FROM Book) >= 0

);

**Purpose**:  
Prevent any book in the database from having a negative value for CopiesAvailable.

**Detailed Explanation**:

1. **Subquery Logic**:
   * (SELECT MIN(CopiesAvailable) FROM Book) retrieves the smallest CopiesAvailable value across all rows in the Book table.
2. **Assertion Condition**:
   * MIN(CopiesAvailable) >= 0: Ensures that the minimum value of CopiesAvailable is never less than zero.
3. **Database Integrity**:
   * Guarantees that operations affecting book availability, such as loans or returns, cannot reduce the CopiesAvailable column to a negative number.
4. **Result**:
   * Prevents logical inconsistencies in the database, ensuring accurate stock management.

Not Supported by MYSQL

**Query 15 Advanced View**

*CREATE VIEW ActiveLoansSummary AS*

*SELECT Member.FirstName, Member.LastName, COUNT(Loan.LoanID) AS TotalActiveLoans*

*FROM Member*

*LEFT JOIN Loan ON Member.MemberID = Loan.MemberID AND Loan.LoanStatus = 'Active'*

*GROUP BY Member.MemberID;*

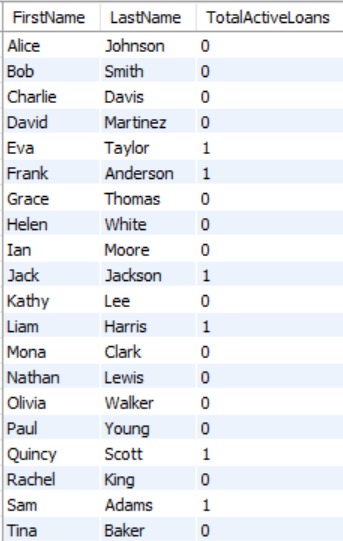
**Purpose**

The purpose of this query is to **create a view** that summarizes the total number of **active loans** for each library member. It provides a quick, reusable summary that displays the first name, last name, and count of active loans for each member.

**Detailed Explanation**

1. **CREATE VIEW ActiveLoansSummary AS**
   * Creates a **view** named ActiveLoansSummary to store the results of this query.
   * Views are virtual tables that allow users to retrieve precomputed data without rewriting the query each time.
2. **SELECT Member.FirstName, Member.LastName, COUNT(Loan.LoanID) AS TotalActiveLoans**
   * **Member.FirstName and Member.LastName**: Retrieves the member's first and last names for easy identification.
   * **COUNT(Loan.LoanID)**:
     + Counts the total number of loans where the LoanStatus is 'Active'.
     + If a member has no active loans, the count will return 0 due to the LEFT JOIN.
3. **LEFT JOIN Loan ON Member.MemberID = Loan.MemberID AND Loan.LoanStatus = 'Active'**
   * A **LEFT JOIN** ensures that all records from the Member table are included, even if there are no active loans associated with a member.
   * The condition Loan.LoanStatus = 'Active' ensures only loans with the status 'Active' are considered in the count.
   * **Key Benefit of LEFT JOIN**: Members without any active loans will still appear in the result, with a loan count of 0.
4. **GROUP BY Member.MemberID**
   * Groups the results by each unique member, identified by their MemberID.
   * This allows the COUNT function to calculate the total number of active loans for each member.
5. **View Output**:
   * The resulting view ActiveLoansSummary contains the following columns:  
     | FirstName | LastName | TotalActiveLoans |
     + **FirstName and LastName**: Member's names.
     + **TotalActiveLoans**: Total count of loans that are active for each member.

**Result**:



Conclusion

The **LibraSys: Unified Library Management System** represents a cornerstone in the modernization of library operations by introducing a comprehensive, scalable, and efficient database system. Libraries, as evolving hubs of knowledge and culture, face increasing complexities in managing diverse resources, member activities, and staff workflows. This project highlights the critical role of an integrated database in addressing these challenges, ensuring smooth operations while meeting the expectations of modern users for instant access to resources and accurate information.

**Key Accomplishments**

The database design incorporates 20 interrelated tables that encapsulate all key entities within the library, such as books, members, employees, suppliers, loans, and reviews. The relationships between these entities were meticulously crafted to reflect real-world interactions, ensuring that every data point serves a meaningful role. Some key highlights of the database are:

* **Entity Relationships**: The ER diagram and relational schema provided a solid foundation for modeling complex interactions. Relationships such as *Author-Book*, *Loan-Fine*, *Book-Reservation*, and many others ensure that the database accurately reflects real-world processes.
* **Data Integrity**: The use of constraints (e.g., primary keys, foreign keys, and check constraints) ensures consistency and prevents invalid data entries. For instance, *LoanStatus* is restricted to specific valid values, and negative values for *CopiesAvailable* or *Fine Amounts* are prohibited.
* **Complex Attributes**: Features such as composite attributes (e.g., addresses split into street, city, state, and zip code), multi-valued attributes (e.g., phone numbers), and derived attributes (e.g., average book ratings) were implemented to enrich data representation and facilitate better reporting.

**SQL Queries and Their Impact**

Throughout the project, a wide variety of SQL queries were developed to meet operational, analytical, and reporting needs. These queries ensure the efficient retrieval, modification, and monitoring of data, enhancing the library's ability to make informed decisions and provide seamless services.

**Basic Queries:**

1. Retrieve the email addresses of members who reserved a specific book copy.
2. List members who reserved rooms with even-numbered IDs.
3. Identify employees responsible for cleaning or working on specific library floors.
4. Count the number of reviews for each rating category to analyze feedback.

**Advanced Queries:**

Advanced SQL queries were created to tackle more complex use cases, highlighting the versatility and power of the database system:

* **Nested Queries and Aggregations**: Queries such as retrieving books currently overdue or identifying members who have borrowed all books by a specific author leverage multi-level nesting and set operations.
* **Views**: Views like *OverdueLoansView* and *ActiveLoansSummary* provide simplified yet powerful abstractions for regularly accessed data, allowing librarians to focus on key metrics such as overdue loans or active loan counts per member.
* **Triggers and Stored Procedures**: The system includes a trigger to automatically decrement the number of available copies when a loan is created, as well as a stored procedure to simplify the simultaneous recording of a loan and updating of book availability.
* **Assertions and Integrity Checks**: Although not fully supported in MySQL, theoretical assertions were designed to maintain rules like a minimum of 100 books or prohibiting negative values for available book copies.

**Key Examples of Advanced SQL:**

* **Finding Members Who Reviewed Overdue Books**: A multi-level nested query identifies members who provided feedback on books still overdue, combining the *Loan*, *Review*, and *Member* tables.
* **Supplier Completeness**: An advanced SQL division query determines suppliers who provide every book available in the library, ensuring comprehensive supplier performance analysis.
* **Average Loan Durations**: A nested query calculates the average number of days members take to return books, providing insights into borrowing behavior.

These queries not only enable operational efficiency but also empower library administrators with actionable insights, such as improving resource allocation, identifying popular books, and targeting high-value members for personalized services.

**Future Enhancements**

While the current system addresses the critical needs of library management, future developments can unlock even greater potential:

1. **Enhanced User Interfaces**: Developing mobile and web applications with seamless integration into the database will improve usability for both members and staff.
2. **Digital Resource Management**: Extending the system to manage e-books, online journals, and multimedia archives will cater to the growing demand for digital resources.
3. **Predictive Analytics**: Leveraging advanced analytics tools to predict borrowing trends, recommend books, and optimize resource procurement based on historical data.
4. **Advanced Security**: Implementing role-based access controls, encryption, and auditing mechanisms to safeguard sensitive data and ensure compliance with privacy regulations.
5. **Interoperability with External Systems**: API integrations for third-party services like e-book platforms or supplier management systems to automate procurement and expand offerings.

**Final Remarks**

In conclusion, the **LibraSys** database system is a testament to the transformative power of database technologies in modernizing library management. By centralizing data, automating workflows, and enabling deep insights through advanced queries, the system empowers libraries to remain relevant in the digital age. Whether it is tracking overdue loans, managing procurement, or analyzing user behavior, the database lays the groundwork for operational excellence and user satisfaction. With future enhancements, the system will continue to adapt to the evolving needs of libraries, ensuring they remain indispensable hubs of learning, culture, and community engagement.