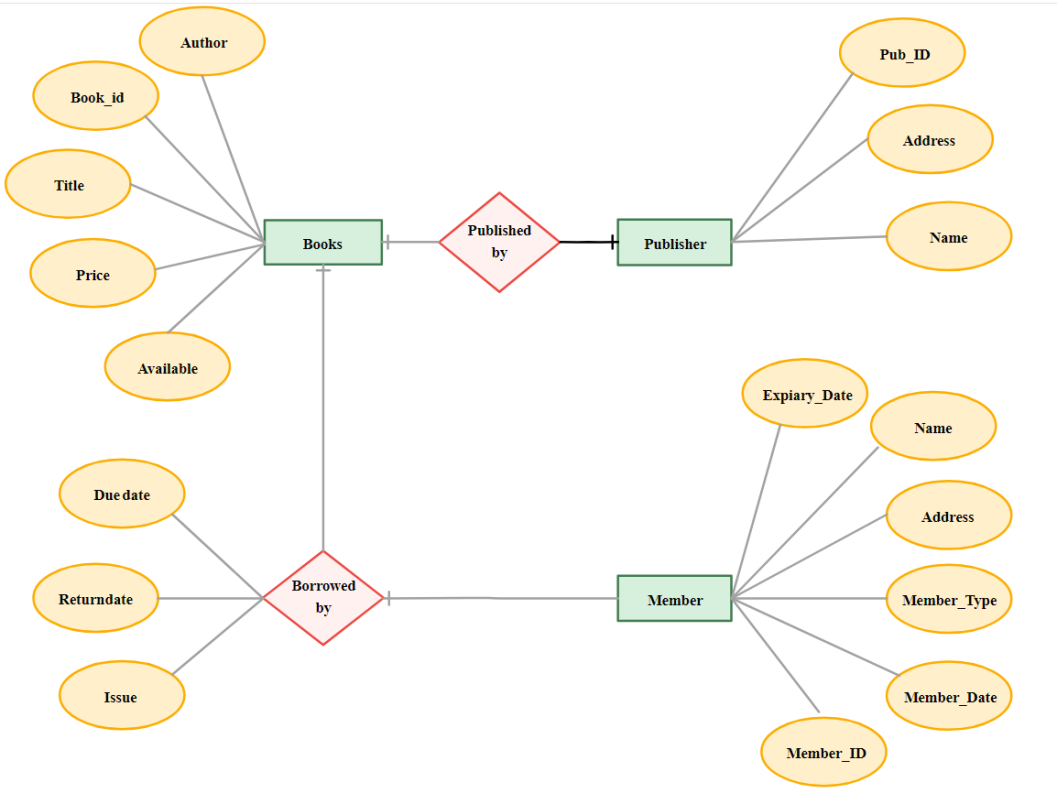
Assignment 1: Analyze a given business scenario and create an ER diagram that includes entities, relationships, attributes, and cardinality. Ensure that the diagram reflects proper normalization up to the third normal form.

Steps to Create an ER Diagram:

1. **Identify Entities:** Start by identifying the main entities involved in the business scenario. These entities represent real-world objects or concepts with which the system deals (e.g., Customer, Order, Product).
2. **Define Attributes:** For each entity, determine the relevant attributes that describe its characteristics (e.g., Customer: customerID, name, address; Order: orderID, date, customerID).
3. **Identify Relationships:** Establish the relationships between the entities. These connections represent how entities interact with each other (e.g., a Customer places an Order).
4. **Cardinality:** Specify the cardinality of each relationship, indicating how many instances of one entity can relate to how many instances of another entity (e.g., One Customer can place Many Orders; One Order is placed by One Customer).
5. **Normalization:** Aim for a normalized ER diagram to minimize data redundancy and improve data integrity. Strive for at least Third Normal Form (3NF), which eliminates transitive dependencies.



Example: Library Management System

Here's a sample ER diagram for a library management system to illustrate the concepts:

- **Entities:**
  - Book (bookID, title, author, ISBN)
  - Author (authorID, name)
  - Member (memberID, name, address)
  - Borrowing (borrowingID, memberID, bookID, borrowDate, returnDate)
- **Relationships:**
  - Book written by Author (One Author can write Many Books; One Book is written by One Author)
  - Member borrows Book (One Member can borrow Many Books; One Book can be borrowed by Many Members)
- **Cardinality:**
  - Book:Author (One-to-Many)
  - Member:Book (Many-to-Many) - This is resolved using an associative entity "Borrowing"
- **Normalization:**
  - This example is already in 3NF as it eliminates transitive dependencies (no attribute depends on another non-key attribute).

Assignment 2: Design a database schema for a library system, including tables, fields, and constraints like NOT NULL, UNIQUE, and CHECK. Include primary and foreign keys to establish relationships between tables.

Library Management System Database Schema

This schema represents a relational database design for a library system:

Tables:

1. **Authors:**
   - **author_id (INT PRIMARY KEY):** Unique identifier for the author (auto-increment)
   - **name (VARCHAR(255) NOT NULL):** Author's full name
   - **biography (TEXT):** Optional biography of the author
2. **Books:**
   - **book_id (INT PRIMARY KEY):** Unique identifier for the book (auto-increment)
   - **title (VARCHAR(255) NOT NULL):** Title of the book
   - **isbn (VARCHAR(13) UNIQUE):** International Standard Book Number (unique identifier for the book edition)
   - **publication_year (INT):** Year the book was published

    - **author_id (INT FOREIGN KEY REFERENCES Authors(author_id)):** Foreign key referencing the author's ID in the Authors table (establishes relationship)
3. **Categories:**
    - **category_id (INT PRIMARY KEY):** Unique identifier for the category (auto-increment)
    - **name (VARCHAR(50) NOT NULL UNIQUE):** Category name (e.g., Fiction, Non-Fiction, Biography)
4. **Book_Categories:**
    - **book_id (INT FOREIGN KEY REFERENCES Books(book_id)):** Foreign key referencing the book's ID in the Books table (establishes many-to-many relationship)
    - **category_id (INT FOREIGN KEY REFERENCES Categories(category_id)):** Foreign key referencing the category's ID in the Categories table (establishes many-to-many relationship)
    - **PRIMARY KEY (book_id, category_id):** Composite primary key to uniquely identify a book-category association
5. **Members:**
    - **member_id (INT PRIMARY KEY):** Unique identifier for the member (auto-increment)
    - **name (VARCHAR(255) NOT NULL):** Member's full name
    - **email (VARCHAR(255) UNIQUE):** Member's email address (unique identifier)
    - **member_type (ENUM('Student', 'Faculty', 'Staff'))**: Type of membership (e.g., Student, Faculty, Staff)
6. **Loans:**
    - **loan_id (INT PRIMARY KEY):** Unique identifier for the loan (auto-increment)
    - **book_id (INT FOREIGN KEY REFERENCES Books(book_id)):** Foreign key referencing the book's ID in the Books table (establishes relationship)
    - **member_id (INT FOREIGN KEY REFERENCES Members(member_id)):** Foreign key referencing the member's ID in the Members table (establishes relationship)
    - **loan_date (DATE NOT NULL):** Date the book was loaned
    - **return_date (DATE):** Optional return date (NULL if book is not returned yet)
    - **due_date (DATE):** Calculated due date for the loan (based on loan date and library policy)

Constraints:

- **NOT NULL:** Ensures specific columns must have a value and cannot be empty.
- **UNIQUE:** Guarantees unique values within a column, preventing duplicates (e.g., ISBN, email).
- **PRIMARY KEY:** Defines a unique identifier for each table row.
- **FOREIGN KEY:** Creates a link between tables, ensuring data integrity and referencing existing data.
- **CHECK:** (Can be added) This constraint allows for more specific data validation. For example, a CHECK constraint on due_date could ensure it's always a future date relative to the loan_date.

Relationships:

- One author can write many books (one-to-many between Authors and Books tables).
- One book can have many categories, and one category can belong to many books (many-to-many relationship established by the Book_Categories table).
- One member can have many loans, and one loan belongs to one member (one-to-many between Members and Loans tables).
- One book can have many loans, and one loan can be for one book (one-to-many between Books and Loans tables).

Assignment 3: Explain the ACID properties of a transaction in your own words. Write SQL statements to simulate a transaction that includes locking and demonstrate different isolation levels to show concurrency control.

ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure the integrity and reliability of data manipulation within a database transaction.

- **Atomicity:** Imagine a transaction as a single unit of work. Either all the operations within the transaction succeed, or none of them do. It's like flipping a coin - it lands heads or tails, but it can't be both. This ensures data consistency and prevents partial updates.
- **Consistency:** A transaction takes the database from one valid state to another. It enforces the defined business rules and constraints on the data. Think of it like following a recipe - the ingredients and steps lead to a complete and consistent dish.
- **Isolation:** Transactions are isolated from each other, meaning changes made by one transaction aren't visible to other concurrent transactions until the first transaction is committed. This prevents conflicts and ensures data integrity in a multi-user environment. Imagine multiple chefs working in a kitchen, but each has their own ingredients and workspace until they finish their dish.
- **Durability:** Once a transaction is committed, the changes are permanent and persist even in case of system failures like crashes. It's like putting your finished dish in the oven - it's baked and saved, even if the power goes out.

Transaction with Locking and Isolation Levels

Here's an example transaction simulating a bank account withdrawal with locking and demonstrating different isolation levels:

Scenario:

- Account ID: 100
- Current Balance: $1000

Transaction: Withdraw $200

BEGIN TRANSACTION;

SELECT * FROM accounts WHERE account_id = 100 FOR UPDATE;

SELECT SLEEP(1);

UPDATE accounts SET balance = balance - 200 WHERE account_id = 100; COMMIT;

This transaction uses a SELECT FOR UPDATE statement to acquire a WRITE lock on the account row. This prevents other transactions from modifying the same account data concurrently, ensuring data consistency.

Isolation Levels:

- **READ UNCOMMITTED:** This allows a transaction to see uncommitted changes from other transactions. This can lead to dirty reads (seeing incomplete data).
- **READ COMMITTED:** A transaction only sees changes committed by other transactions before it started. This avoids dirty reads but allows non-repeatable reads (seeing the same data twice and getting different results due to concurrent modifications).
- **REPEATABLE READ:** Guarantees consistent reads within a transaction. It prevents non-repeatable reads but can lead to phantom reads (seeing new rows inserted by other transactions during its execution).
- **SERIALIZABLE:** Provides the strongest isolation level, ensuring serial execution as if transactions were run one after another. This avoids all concurrency anomalies but can significantly impact performance due to frequent locking.

Assignment 4: Write SQL statements to CREATE a new database and tables that reflect the library schema you designed earlier. Use ALTER statements to modify the table structures and DROP statements to remove a redundant table.

Creating the Library Management System Database Schema

1. CREATE DATABASE:

CREATE DATABASE library_system;

2. Use the newly created database:

USE library_system;

3. CREATE Tables:

Authors Table:

CREATE TABLE Authors (

```
  author_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  biography TEXT
);
```

Books Table:

```
CREATE TABLE Books (
  book_id INT PRIMARY KEY AUTO_INCREMENT,
  title VARCHAR(255) NOT NULL,
  isbn VARCHAR(13) UNIQUE,
  publication_year INT,
  author_id INT FOREIGN KEY REFERENCES Authors(author_id)
);
```

Categories Table:

```
CREATE TABLE Categories (
  category_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(50) NOT NULL UNIQUE
);
```

Book_Categories Table (Many-to-Many Relationship):

```
CREATE TABLE Book_Categories (
  book_id INT FOREIGN KEY REFERENCES Books(book_id),
  category_id INT FOREIGN KEY REFERENCES Categories(category_id),
  PRIMARY KEY (book_id, category_id)
);
```

Members Table:

```
CREATE TABLE Members (
  member_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) UNIQUE,
  member_type ENUM('Student', 'Faculty', 'Staff')
);
```

Loans Table:

```
CREATE TABLE Loans (
  loan_id INT PRIMARY KEY AUTO_INCREMENT,
  book_id INT FOREIGN KEY REFERENCES Books(book_id),
  member_id INT FOREIGN KEY REFERENCES Members(member_id),
  loan_date DATE NOT NULL,
  return_date DATE,
  due_date DATE
```

);

Modifying Table Structures (ALTER):

1. Add a new column 'genre' (VARCHAR(50)) to the Books table:

ALTER TABLE Books
ADD COLUMN genre VARCHAR(50);

2. Make the 'due_date' column in the Loans table NOT NULL:

ALTER TABLE Loans
MODIFY due_date DATE NOT NULL;

Removing a Redundant Table (DROP):

Assuming the 'Book_Categories' table serves the purpose of associating books with categories, the 'Categories' table might be redundant.

DROP TABLE Categories;

Assignment 5: Demonstrate the creation of an index on a table and discuss how it improves query performance. Use a DROP INDEX statement to remove the index and analyze the impact on query execution.

Indexing for Speed: Optimizing Library System Queries

Scenario:

Consider the Books table in our library system schema. Frequently used queries might search for books by title. Here's how indexing can improve performance:

1. CREATE INDEX:

CREATE INDEX idx_title ON Books(title);

2. How Indexing Improves Performance:

- **Faster Searches:** When a query searches for books by title, the database engine can leverage the index to quickly locate relevant entries instead of scanning the entire table. This significantly reduces query execution time, especially for large datasets.
- **Optimized Data Structures:** Indexes use optimized data structures like B-trees for faster searching and sorting.

3. DROP INDEX:

DROP INDEX idx_title ON Books;

4. Impact of Removing the Index:

Without the index, searching by title becomes less efficient. The database engine needs to scan the entire Books table for each search, potentially leading to slower query execution times.

Assignment 6: Create a new database user with specific privileges using the CREATE USER and GRANT commands. Then, write a script to REVOKE certain privileges and DROP the user.

Creating and Managing Database Users

1. CREATE USER:

CREATE USER 'new_user' IDENTIFIED BY 'secure_password';

2. GRANT Privileges:

GRANT SELECT, INSERT ON library_system.* TO 'new_user';

3. REVOKE Privileges:

REVOKE INSERT ON library_system.Books TO 'new_user';

4. DROP USER:

DROP USER 'new_user';

Assignment 7: Prepare a series of SQL statements to INSERT new records into the library tables, UPDATE existing records with new information, and DELETE records based on specific criteria. Include BULK INSERT operations to load data from an external source.

1. INSERT Statements:

- **Insert a new Author:**

INSERT INTO Authors (name, biography)
VALUES ('William Shakespeare', 'Renowned English playwright, poet, and actor');

- **Insert a new Book:**

INSERT INTO Books (title, isbn, publication_year, author_id)
VALUES ('Hamlet', '978-1509888339', 1603, 1); -- Author ID references existing author

- **Insert a new Member:**

```
INSERT INTO Members (name, email, member_type)
VALUES ('John Doe', 'johndoe@example.com', 'Student');
```

- **Insert a new Loan (assuming book and member exist):**

```
INSERT INTO Loans (book_id, member_id, loan_date, due_date)
VALUES (1, 1, CURRENT_TIMESTAMP, DATE_ADD(CURRENT_TIMESTAMP, INTERVAL 14 DAY));  -- Due date 14 days from now
```

2. UPDATE Statements:

- **Update Book Title:**

```
UPDATE Books
SET title = 'The Great Gatsby'
WHERE book_id = 2;  -- Assuming book ID 2 exists
```

- **Update Member Email:**

```
UPDATE Members
SET email = 'janedoe@example.com'
WHERE member_id = 3;  -- Assuming member ID 3 exists
```

3. DELETE Statements:

- **Delete a Loan Record (assuming proper checks):**

```
DELETE FROM Loans
WHERE loan_id = 1;  -- Assuming loan ID 1 exists
```

4. BULK INSERT for Efficiency:

Imagine you have a CSV file containing book data. You can use a BULK INSERT operation to efficiently load this data into the Books table:

```
BULK INSERT Books (title, isbn, publication_year, author_id)
FROM 'C:\path\to\books.csv'
WITH (FIELDTERMINATOR = ',', ROWTERMINATOR = '\n');
```