ADVANCED BACKEND DEVELOPMENT

Practice 2

CHAPTER 2: WORKING WITH DATABASES

Topic: LIBRARY MANAGEMENT SYSTEM

Phạm Trần Gia Hưng – 2331200153

QUESTION 1: DEVELOPMENT APPROACHES & INHERITANCE

(Development Approaches + Inheritance in Relational Databases)

```sql
-- Question1


-- TPH
CREATE TABLE Users (
    UserId INT IDENTITY(1,1) PRIMARY KEY,
    UserType NVARCHAR(50) NOT NULL,
    FullName NVARCHAR(255) NOT NULL,
    Email NVARCHAR(255) NOT NULL UNIQUE,

    -- Student
    StudentCode NVARCHAR(50) NULL,
    Major NVARCHAR(255) NULL,

    -- Lecturer
    LecturerCode NVARCHAR(50) NULL,
    Department NVARCHAR(255) NULL,

    CONSTRAINT CK_UserType CHECK (UserType IN ('Student', 'Lecturer'))
);
```

```sql
-- Books
CREATE TABLE Books (
    BookId INT IDENTITY(1,1) PRIMARY KEY,
    Title NVARCHAR(500) NOT NULL,
    Author NVARCHAR(255) NOT NULL,
    ISBN NVARCHAR(50),
    TotalCopies INT NOT NULL DEFAULT 1,
    AvailableCopies INT NOT NULL DEFAULT 1,
    CONSTRAINT CK_Copies CHECK (AvailableCopies >= 0 AND AvailableCopies <= Total
);

```

```sql
-- BorrowingTransactions
CREATE TABLE BorrowingTransactions (
    TransactionId INT IDENTITY(1,1) PRIMARY KEY,
    UserId INT NOT NULL,
    BookId INT NOT NULL,
    BorrowDate DATETIME NOT NULL DEFAULT GETDATE(),
    DueDate DATETIME NOT NULL,
    ReturnDate DATETIME NULL,
    Status NVARCHAR(50) NOT NULL DEFAULT 'Borrowed',
    CONSTRAINT FK_Borrow_User FOREIGN KEY (UserId) REFERENCES Users(UserId),
    CONSTRAINT FK_Borrow_Book FOREIGN KEY (BookId) REFERENCES Books(BookId)
);
```

Task 1:

1.1 Database-First Approach

Database-First is a development approach where the database schema is designed and created first, then the application code and models are generated from the existing database structure. Help provides full control over database design and optimization

Advantage: Easier to implement complex database features like stored procedures, triggers, and custom indexing

Disadvantage: Not versatile for fast prototyping and iterative development

1.2 Code-First Approach

Code-First is a development approach where developers define entity classes in code, and the database schema is automatically generated or updated based on these classes. Entity classes are written in programming language (C#, Java). Provides an object-oriented approach to database design

Advantage: Faster development, easier to maintain

Disadvantages: Difficult to implement complex database features without raw SQL

→ Indicate which approach is more suitable for a library management system and justify your choice

Suitable approach: Code-First Approach

Justification: The library system is data centric and requires strong consistency, integrity, transactional control. Since multiple users access the system concurrently, a stable and well defined database schema is better

- Agile: Library systems might require iterative improvements and feature. Code-First allow fast prototyping, easy schema modifications through migration.

- Version Control: Migration files provide history of changes, make it easier for dev to collab and view changes.

- Inheritance Support: Code-First framework (like Entity Framework) provide support for implementing inheritance patterns (TPH, TPT, TPC) through annotation or config.

Task 2 Design the database schema to represent **inheritance in a relational database** using **one of the following strategies**

Design with TPH

```sql
-- Question1


-- TPH
CREATE TABLE Users (
    UserId INT IDENTITY(1,1) PRIMARY KEY,
    UserType NVARCHAR(50) NOT NULL,
    FullName NVARCHAR(255) NOT NULL,
    Email NVARCHAR(255) NOT NULL UNIQUE,

    -- Student
    StudentCode NVARCHAR(50) NULL,
    Major NVARCHAR(255) NULL,

    -- Lecturer
    LecturerCode NVARCHAR(50) NULL,
    Department NVARCHAR(255) NULL,

    CONSTRAINT CK_UserType CHECK (UserType IN ('Student', 'Lecturer'))
);
```

```sql
-- Books
CREATE TABLE Books (
    BookId INT IDENTITY(1,1) PRIMARY KEY,
    Title NVARCHAR(500) NOT NULL,
    Author NVARCHAR(255) NOT NULL,
    ISBN NVARCHAR(50),
    TotalCopies INT NOT NULL DEFAULT 1,
    AvailableCopies INT NOT NULL DEFAULT 1,
    CONSTRAINT CK_Copies CHECK (AvailableCopies >= 0 AND AvailableCopies <= Total
);
```

```sql
-- BorrowingTransactions
CREATE TABLE BorrowingTransactions (
    TransactionId INT IDENTITY(1,1) PRIMARY KEY,
    UserId INT NOT NULL,
    BookId INT NOT NULL,
    BorrowDate DATETIME NOT NULL DEFAULT GETDATE(),
    DueDate DATETIME NOT NULL,
    ReturnDate DATETIME NULL,
    Status NVARCHAR(50) NOT NULL DEFAULT 'Borrowed',
    CONSTRAINT FK_Borrow_User FOREIGN KEY (UserId) REFERENCES Users(UserId),
    CONSTRAINT FK_Borrow_Book FOREIGN KEY (BookId) REFERENCES Books(BookId)
);
```

```sql
SELECT TOP (1000) [UserId]
      ,[UserType]
      ,[FullName]
      ,[Email]
      ,[StudentCode]
      ,[Major]
      ,[LecturerCode]
      ,[Department]
  FROM [CSW431_PhamTranGiaHung_2331200153_lab2].[dbo].[Users]
```

00 % 9 0 Ln: 10 Ch: 1

Results | Messages

	UserId	UserType	FullName	Email	StudentCode	Major	LecturerCode	Department
1	1	Student	Miyabi	a@student.com	SC001	CSE	NULL	NULL
2	2	Student	Fangyi	b@student.com	SC002	ECE	NULL	NULL
3	3	Lecturer	Dr.Typhon	abc@lecturer.com	NULL	NULL	LC001	Software Engineering

```sql
SELECT TOP (1000) [BookId]
      ,[Title]
      ,[Author]
      ,[ISBN]
      ,[TotalCopies]
      ,[AvailableCopies]
  FROM [CSW431_PhamTranGiaHung_2331200153_lab2].[dbo].[Books]
```

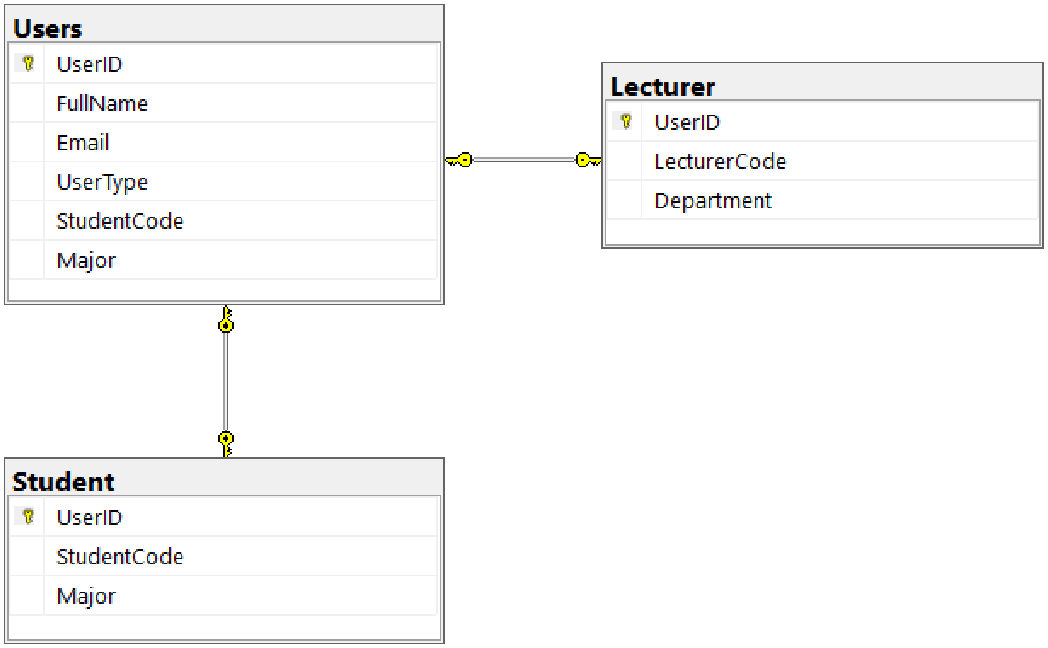
100 %   7   0

Results   Messages

	BookId	Title	Author	ISBN	TotalCopies	AvailableCopies
1	1	Backend	Hypergryph	111-001	5	5
2	2	Database	Arknight	111-002	3	3
3	3	Backend	Hypergryph	111-001	5	5
4	4	Database	Arknight	111-002	3	3

Task 3

Provide an **ERD diagram or SQL schema** for the selected design



QUESTION 2: CONCURRENCY HANDLING & TRANSACTIONS

(Concurrency Handling and Transactions)

Scenario

Two students attempt to borrow the **same book at the same time**, while only **one copy** of the book is available.

Tasks

1. Analyze potential **concurrency issues**, such as:
   - Lost Update
   - Dirty Read
   - Race Condition
2. Design a **book borrowing process** using **transactions**, including:
   - Checking the number of available copies
   - Creating a borrowing record
   - Decreasing the available book quantity

3. Provide **pseudo-code or SQL examples** illustrating:
   - BEGIN TRANSACTION
   - COMMIT

ROLLBACK

Task 1: Analyze potential concurrency issues

When two students attempt to borrow the same book simultaneously with only one copy available, several concurrency issues can occur:

- Lost Update Problem: This occurs when two transactions read the same data, modify it independently, and write it back. The last write overwrites the first, causing the first update to be lost.

Ex: Both students read AvailableCopies = 1, both check availability (pass), both decrement to 0. Result: Both successfully borrow but only 1 copy exists - data integrity violated!

- Dirty Read Problem: This occurs when a transaction reads data that has been modified by another transaction but not yet committed. If the modifying transaction rolls back, the reading transaction has used invalid data.

Ex : Student A decrements AvailableCopies to 0 (uncommitted). Student B reads 0 and is rejected. Student A rolls back. Result: Student B was incorrectly rejected based on uncommitted data!

- Race Condition: A race condition occurs when the system behavior depends on the sequence or timing of uncontrollable events. Multiple processes compete for the same resource.

Ex : Both requests arrive simultaneously. Without proper locking, both pass availability check before either decrements the counter. Result: AvailableCopies becomes negative, data integrity violated!

Task 2: Design a book borrowing process using transactions

- Process flow: Must follow ACID property

1.Begin transaction with appropriate isolation level

2.Lock the book record for update (prevent concurrent modification)

3.Check if available copies > 0

4.If available create borrowing record

5.Commit transaction, rollback if error

```sql
-- BorrowingTransactions
CREATE TABLE BorrowingTransactions (
    TransactionId INT IDENTITY(1,1) PRIMARY KEY,
    UserId INT NOT NULL,
    BookId INT NOT NULL,
    BorrowDate DATETIME NOT NULL DEFAULT GETDATE(),
    DueDate DATETIME NOT NULL,
    ReturnDate DATETIME NULL,
    Status NVARCHAR(50) NOT NULL DEFAULT 'Borrowed',
    CONSTRAINT FK_Borrow_User FOREIGN KEY (UserId) REFERENCES Users(UserId),
    CONSTRAINT FK_Borrow_Book FOREIGN KEY (BookId) REFERENCES Books(BookId)
);
```

```sql
-- question2
CREATE PROCEDURE BorrowBook
    @UserId INT,
    @BookId INT,
    @DueDate DATETIME
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        DECLARE @AvailableCopies INT;

        -- Lock availability
        SELECT @AvailableCopies = AvailableCopies
        FROM Books WITH (UPDLOCK, HOLDLOCK)
        WHERE BookId = @BookId;

        -- Check available
        IF @AvailableCopies IS NULL OR @AvailableCopies <= 0
        BEGIN
            ROLLBACK;
```

```sql
            ROLLBACK;
            PRINT 'Book not available';
            RETURN;
        END

        -- Create borrowing
        INSERT INTO BorrowingTransactions (UserId, BookId, BorrowDate, DueDate,
        VALUES (@UserId, @BookId, GETDATE(), @DueDate, 'Borrowed');

        -- Decrease copies
        UPDATE Books
        SET AvailableCopies = AvailableCopies - 1
        WHERE BookId = @BookId;

        COMMIT;
        PRINT 'Borrow successful';
    END TRY
    BEGIN CATCH
        ROLLBACK;
```

```sql
-- question2
CREATE PROCEDURE BorrowBook
    @UserId INT,
    @BookId INT,
    @DueDate DATETIME
AS
BEGIN
    BEGIN TRANSACTION;

    BEGIN TRY
        DECLARE @AvailableCopies INT;

        -- Lock availability
        SELECT @AvailableCopies = AvailableCopies
        FROM Books WITH (UPDLOCK, HOLDLOCK)
        WHERE BookId = @BookId;

        -- Check available
        IF @AvailableCopies IS NULL OR @AvailableCopies <= 0
        BEGIN
            ROLLBACK;
            PRINT 'Book not available';
            RETURN;
        END

        -- Create borrowing
        INSERT INTO BorrowingTransactions (UserId, BookId, BorrowDate, DueDate,
Status)
        VALUES (@UserId, @BookId, GETDATE(), @DueDate, 'Borrowed');

        -- Decrease copies
        UPDATE Books
        SET AvailableCopies = AvailableCopies - 1
        WHERE BookId = @BookId;

        COMMIT;
        PRINT 'Borrow successful';
```

```
    END TRY
    BEGIN CATCH
        ROLLBACK;
        PRINT 'Error: ' + ERROR_MESSAGE();
    END CATCH
END;
GO
```

Task 3: Provide pseudo-code or SQL examples illustrating

Pseudo:

```
BEGIN TRANSACTION
READ AvailableCopies FROM Books WHERE BookID = 1
IF AvailableCopies > 0 THEN
  INSERT INTO BookBorrow (UserID, BookID, BorrowDate, DueDate)
  UPDATE Books SET AvailableCopies = AvailableCopies - 1
  COMMIT
ELSE
  ROLLBACK
  RETURN "Book is not available"
END IF
```

QUESTION 3: QUERY OPTIMIZATION & NoSQL DATABASES

Requirements

The library system provides the following functionalities:

- Search books by title or author
- View a user's borrowing history
- Generate statistics for the most frequently borrowed books

Tasks

1. Propose **query optimization techniques** for:
   - Book searching
   - Borrowing statistics
     (e.g., indexing, efficient joins, avoiding SELECT *)
2. Identify:
   - Which parts should use a **Relational Database**

- Which parts could use **NoSQL databases** (MongoDB, Redis, etc.)

3. Compare **Relational Databases vs NoSQL** in the library system based on:
   - Data consistency
   - Performance

Scalability

Task 1

Book Selection: Avoid select *

```sql
-- book selection
SELECT BookId, Title, Author, AvailableCopies
FROM Books
WHERE Title LIKE 'Database%';
```

Borrowing: use index

```sql
-- index for optimization
CREATE INDEX IX_Books_Title ON Books(Title);
CREATE INDEX IX_Books_Author ON Books(Author);
CREATE INDEX IX_Borrowing_UserId ON BorrowingTransactions(UserId);
CREATE INDEX IX_Borrowing_BookId ON BorrowingTransactions(BookId);
```

```sql
-- Test

SELECT * FROM Users;
SELECT * FROM Books;

-- Test borrow
EXEC BorrowBook @UserId = 1, @BookId = 1, @DueDate = '2026-02-01';

-- View borrow
SELECT TOP 5 * FROM vw_BorrowingStats ORDER BY BorrowCount DESC;

-- View transactions
SELECT
    u.FullName,
    b.Title,
    bt.BorrowDate,
    bt.Status
FROM BorrowingTransactions bt
JOIN Users u ON bt.UserId = u.UserId
JOIN Books b ON bt.BookId = b.BookId;
GO
```

- Why use avoid * : Specifying only needed columns (SELECT BookId, Title, Author, AvailableCopies) instead of SELECT * (select all attr) reduces data transfer and memory usage, resulting in faster search responses

- Why use index: Indexes improve query performance for borrowing statistics by creating a sorted data structure that allows the database to quickly locate without scanning the entire table

Task 2

2.1 Components Using Relational Database (SQL Server)

- User Management: Requires ACID compliance and referential integrity

- Book Inventory: Structured data with complex relationships

- Borrowing Transactions: Requires transactions and atomicity

- Financial Records: Critical data requiring strict consistency

2.2 Components Using NoSQL Databases

- Search Index (Elasticsearch): Full-text search with fuzzy matching

- Session Data (Redis): In-memory storage for fast access

- Cache Layer (Redis): Reduce database load for popular queries

- Activity Logs (MongoDB): Flexible schema for varied log formats

- Analytics Data (MongoDB): Aggregation pipelines for statistics

Task 3

- Data Consistency

+ Relational (SQL): Strong consistency (ACID), full transaction support, enforced via constraints and foreign keys. Best for financial transactions and inventory management.

+ NoSQL: Eventual consistency (BASE), limited or document-level transactions, application-level validation required. Best for logs, caching, and analytics where eventual consistency is acceptable.

- Performance

+ Relational (SQL): Good read performance with proper indexing, slower for complex joins. Moderate write performance constrained by ACID compliance. Good for complex queries with JOINs and aggregations.

+ NoSQL: Good read performance especially for key-value lookups. High write throughput optimized for writes. Limited complex querying, optimized for simple lookups. Built-in memory caching.

- Scalability

+ Relational (SQL): Good vertical scaling. Difficult horizontal scaling. Best for medium-scale applications with complex relationships.

+ NoSQL: Good vertical scaling but not primary method. Good horizontal scaling, designed for distributed systems. Best for large-scale applications, big data, and high traffic