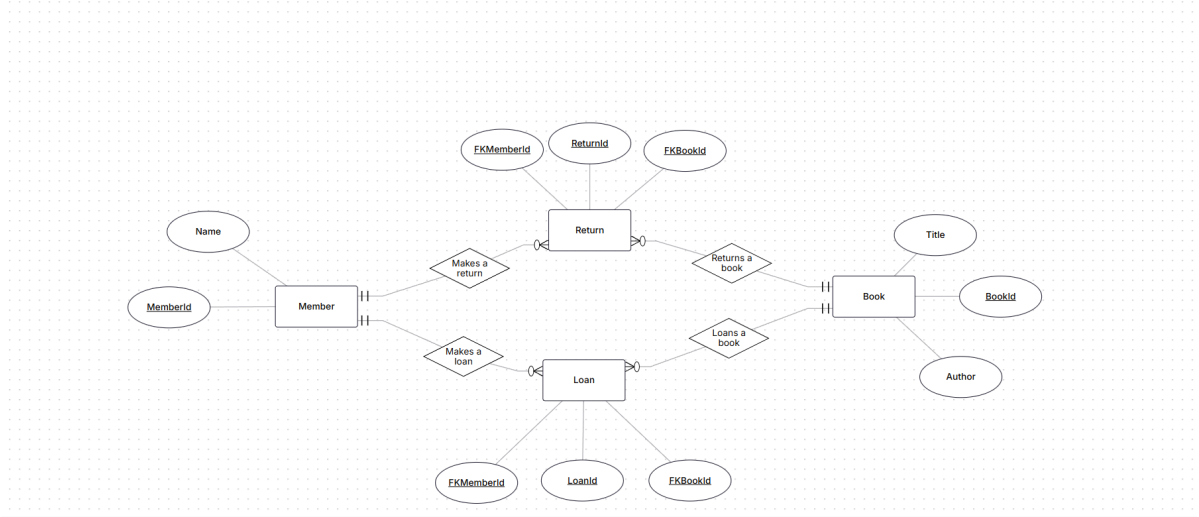
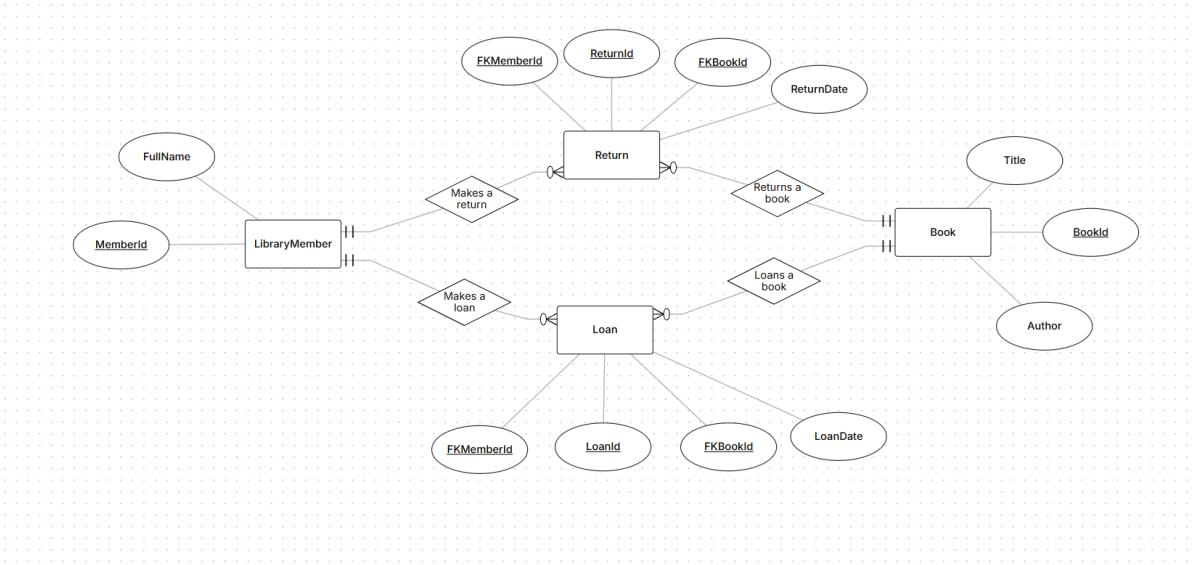
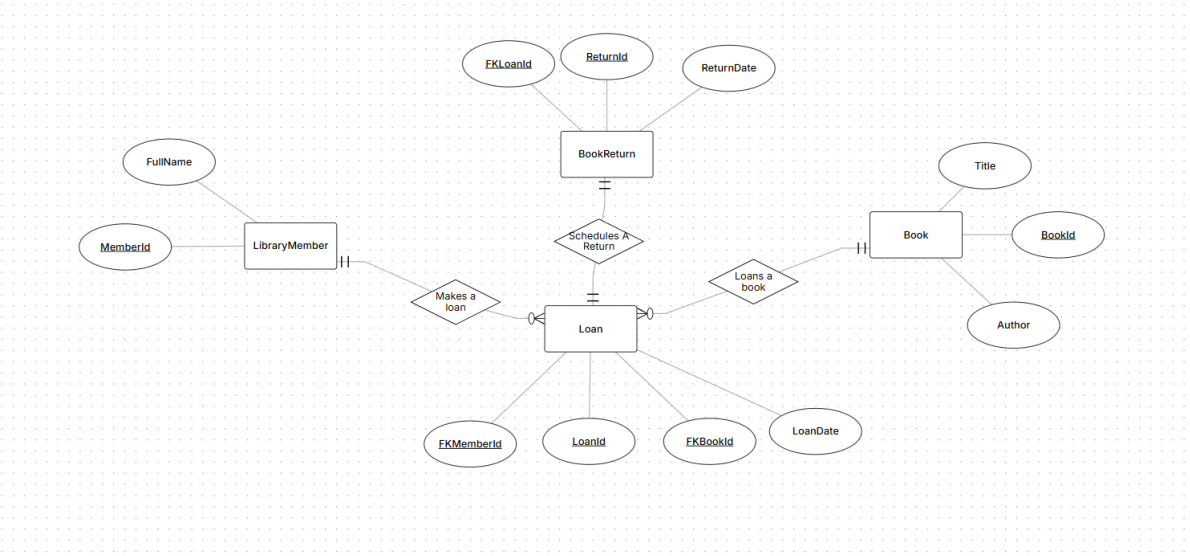
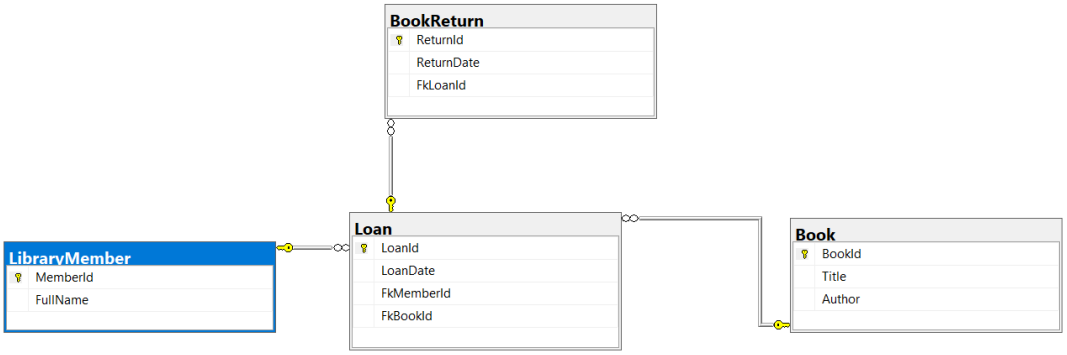
Dokumentation För K2U2

Jag började genom att skapa ett ER-Diagram med ERDPlus. Det tog några försök









Tillslut fick jag något som jag tyckte blev smidigt och bra.

Jag använde mig av den gyllene regeln och av en tabell för att lättare veta var jag borde placera index. (Jag skriver mer om mina index strategier längre ner)

“The Golden Rule: Index columns that you use frequently in `WHERE` clauses or `JOIN` statements, but avoid indexing columns that change constantly or are rarely searched.”

Feature	Without Index	With Index
Search Speed	Slow (Reads every row)	**Very Fast** (Jumps to data)
Insert/Update Speed	**Fast**	Slower (SQL must update the index too)
Disk Space	Minimal	Uses extra space to store the index

```sql
-- Insert Loans
INSERT INTO Loan (LoanDate, FkMemberId, FkBookId)
VALUES
('2023-10-01', 1, 1), -- Alice borrows Gatsby
('2023-10-05', 2, 2), -- Bob borrows 1984
('2023-10-10', 3, 4), -- Charlie borrows The Hobbit
('2023-10-12', 1, 3); -- Alice borrows Mockingbird

-- Insert Returns
INSERT INTO BookReturn (ReturnDate, FkLoanId)
VALUES
('2023-10-15', 1), -- Alice returns Gatsby
('2023-10-20', 2); -- Bob returns 1984
```

Dokumentation av implementering av transaktioner

Konkurrenshantering

```sql
CREATE OR ALTER PROCEDURE [dbo].[sp_RegisterNewLoan]
    @MemberId INT,
    @BookId INT
AS
BEGIN
    IF EXISTS (SELECT 1 FROM View_ActiveLoans WHERE BookTitle = (SELECT Title FROM Book WHERE BookId = @BookId))
    BEGIN
        PRINT 'Boken är redan utlånad.';
        RETURN;
    END

    INSERT INTO Loan (LoanDate, FkMemberId, FkBookId)
    VALUES (GETDATE(), @MemberId, @BookId);

    PRINT 'Lånet har registrerats.';
END
GO
```

Mitt första försök på konkurrenshantering var genom att ha en typ av “if sats” i min “RegisterNewLoan” procedure som utför en “kontroll” ifall boken som försöker lånas redan har lagts in i “ActiveLoans” view:en – alltså att någon redan har lånat den.

Problemet som uppstod när jag simulerade samtidiga lån var att om två användare kör min procedur exakt samtidigt händer detta:
Användare A kollar IF EXISTS. Svaret är "Nej, boken är ledig".
Användare B kollar IF EXISTS en millisekund senare. Svaret är fortfarande "Nej, boken är ledig" (eftersom Användare A inte hunnit skriva in sitt lån än).
Användare A kör INSERT.
Användare B kör INSERT.
Resultat: Samma bok har lånats ut till två personer samtidigt.

Jag löste detta genom att använda mig av två saker, transactions och isolation. Min nya kod såg ut såhär:

```sql
CREATE OR ALTER PROCEDURE [dbo].[sp_RegisterNewLoan]
    @MemberId INT,
    @BookId INT
AS
BEGIN
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

    BEGIN TRANSACTION;

    BEGIN TRY
        IF EXISTS (SELECT 1 FROM View_ActiveLoans WHERE BookTitle = (SELECT Title FROM Book WHERE BookId = @BookId))
        BEGIN
            PRINT 'Boken är redan utlånad.';
            ROLLBACK;
            RETURN;
        END

        INSERT INTO Loan (LoanDate, FkMemberId, FkBookId)
        VALUES (GETDATE(), @MemberId, @BookId);

        COMMIT;
        PRINT 'Lånet har registrerats.';
    END TRY
    BEGIN CATCH
        ROLLBACK;
        PRINT 'Ett fel uppstod vid registrering av lånet.';
    END CATCH
END
GO
```

Jag använde en transaktion för att göra systemet säkrare och mer tillförlitligt, särskilt i större skalor. Transaktionen säkerställer att processen utförs korrekt: antingen uppfylls villkoren och ett lån skapas, eller så avbryts hela processen genom en rollback och databasen återgår till sitt tidigare tillstånd.

Jag använder isolation level SERIALIZABLE för att förhindra att två användare samtidigt skapar en transaktion på samma bok. Genom att använda den högsta isolationsnivån säkerställs att ingen annan transaktion kan påbörjas på en bok som redan är involverad i en pågående transaktion.

När jag simulerar konkurrenshantering med denna stored procedure så uppstår inte problemen som jag råkade ut för tidigare!

Analys av execution plans och indexstrategier

Jag började utan några egna indexstrategier och använde bara tabellernas grundstruktur. När jag analyserade execution plans i detta läge såg jag att SQL Server ofta använde table scans, särskilt vid sökningar på böcker och vid joins mellan Book, Loan och LibraryMember. Detta fungerade men är ineffektivt när datamängden växer. Efter detta införde jag en tydligare indexstrategi genom att skapa index på kolumner som används ofta i WHERE-villkor och relationer, till exempel Title och Author i Book-tabellen samt foreign keys i Loan och BookReturn. Som jag nämnt tidigare gjorde detta att execution plans istället använde index seeks.

```
-- Creates index for book title
CREATE INDEX IX_Book_Title ON Book(Title);

-- Creates index for book author
CREATE INDEX IX_Book_Author ON Book(Author);

-- Creates index for member foreign key
CREATE INDEX IX_Loan_MemberId ON Loan(FkMemberId);

-- Creates index for book foreign key
CREATE INDEX IX_Loan_BookId ON Loan(FkBookId);

-- Creates index for loan foreign key
CREATE INDEX IX_BookReturn_LoanId ON BookReturn(FkLoanId);
```

Skillnaden mellan dessa två strategier är att table scans innebär att hela tabeller läses igenom varje gång, medan index gör att databasen snabbt kan hitta rätt rader. Efter att index lades till blev frågorna snabbare och mer stabila, samtidigt som belastningen på databasen minskade.

Dokumentering av användning av isolation levels

Som jag skrev tidigare började jag med att använda SQL Servers default isolation level (READ COMMITTED). Med den nivån uppstod problemet vid samtidiga lån, där två användare kunde kontrollera att samma bok var ledig och sedan båda registrera ett lån. Default isolation level tillät alltså att dessa operationer överlappade varandra. Efter detta ändrade jag till isolation level SERIALIZABLE i min lagrade procedur. Till skillnad från default-nivån gör SERIALIZABLE att transaktioner hanteras som om de körs en i taget. När en transaktion kontrollerar om en bok är utlånad blockeras andra transaktioner från att göra samma sak tills den första är klar. Skillnaden mellan dessa två isolation levels är alltså att READ COMMITTED ger bättre prestanda men sämre skydd mot samtidighetsproblem, medan SERIALIZABLE ger starkare dataintegritet på bekostnad av att fler transaktioner kan behöva vänta.