

Laboration 2

Sam Sahbaeirazavi

Version 1.0

Innehåll

1 Inledning	3
2 Problembeskrivning	3
3 Systembeskrivning	3
4 Implementation	5
5 Gränssnitt	7
6 Lösningens begränsningar	19
7 Problem och reflektioner	20
8 Referenser	21
9 Bilagor	21

1 Inledning

Syftet med laboration 2 är att ge grundläggande kunskap och praktisk erfarenhet av relationsdatabaser, databasdesign och frågespråket SQL.

För denna uppgift syftar jag på G-nivå, vilket innebär:

- En databas med minst två tabeller och en en-till-många-relation.
- Design beskriven med ER-diagram.
- Implementation i SQL Server (create, insert, select, update, delete).
- Referensintegritet demonstrerad.
- Övningsuppgifter (12, 16, 19) från Northwind.
- Lagrade procedurer, vyer och transaktioner (uppgifter 21, 23, 25) också gjort på NorthWind.

2 Problembeskrivning

Problemet är att designa och implementera en enkel relationsdatabas som kan användas för att lagra information om filmer och genrer.

Databasen ska visa hur tabeller kopplas samman med primärnycklar, främmande nycklar och referensintegritet. Dessutom ska SQL-frågor kunna användas för att hämta, uppdatera och radera data.

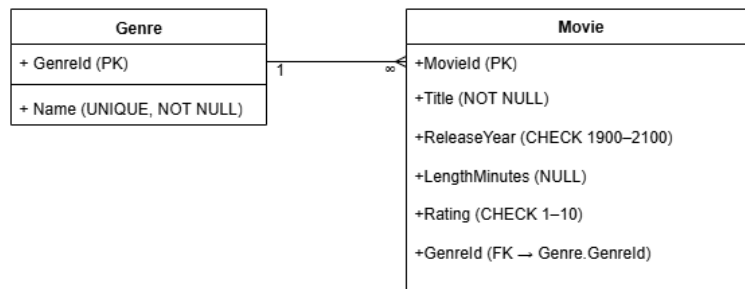
3 Systembeskrivning

Systemet har två tabeller:

- **Genre** (GenreId, Name)
- **Movie** (MovieId, Title, ReleaseYear, LengthMinutes, Rating, GenreId)

Relationen är "en-till-många": varje genre kan ha flera filmer, men varje film tillhör en genre.

ER-diagram



beskrivning av tabeller

- GenreId: primärnyckel, automatisk ID.
- Name: unikt och obligatoriskt.
- MovieId: primärnyckel, automatisk ID.
- Title: obligatoriskt.
- ReleaseYear: måste vara mellan 1900–2100.
- Rating: måste vara mellan 1–10.
- GenreId: främmande nyckel mot Genre.

4 Implementation

- **Språk/verktyg:** SQL Server (LocalDB) via Visual Studio.
- **Kodfiler:**
 - create_tables.sql
 - insert_data.sql
 - SQLQuery1.sql
 - queries_select_update_delete.sql

 - uppgift2.sql (Northwind 12, 16, 19)
 - uppgift.sql (21, 23, 25)

Exempel på kod

1. För att skapa en databas:

```
-- Skapa en databas med namnet MoviesLab2  
CREATE DATABASE MoviesLab2;  
GO
```

2. Sedan skapar vi tabeller för vår data

```
--we create the tables for the genres
CREATE TABLE dbo.Genre (
    GenreId INT IDENTITY(1,1) PRIMARY KEY,
    Name     NVARCHAR(50) NOT NULL UNIQUE
);
GO

-- we create another table for movies
CREATE TABLE dbo.Movie (
    MovieId      INT IDENTITY(1,1) PRIMARY KEY,
    Title        NVARCHAR(200) NOT NULL,
    ReleaseYear  INT NULL,
    LengthMinutes INT NULL,
    Rating       INT NULL,
    GenreId      INT NOT NULL,

    CONSTRAINT FK_Movie_Genre
        FOREIGN KEY (GenreId) REFERENCES dbo.Genre(GenreId)
        ON DELETE NO ACTION
        ON UPDATE NO ACTION,

    CONSTRAINT CK_Movie_Year
        CHECK (ReleaseYear IS NULL OR (ReleaseYear BETWEEN 1900 AND
2100)),
    CONSTRAINT CK_Movie_Rating
        CHECK (Rating IS NULL OR (Rating BETWEEN 1 AND 10))
);
GO

CREATE INDEX IX_Movie_Title    ON dbo.Movie(Title);
CREATE INDEX IX_Movie_GenreId ON dbo.Movie(GenreId);
GO
```

5 Gränssnitt

Bild 1: SELECT som visar alla filmer och deras genre via en JOIN.

```
USE MoviesLab2;
GO

SELECT
    m.MovieId,
    m.Title,
    m.ReleaseYear,
    m.Rating,
    g.Name AS Genre
FROM dbo.Movie AS m
JOIN dbo.Genre AS g ON m.GenreId = g.GenreId
ORDER BY m.MovieId;
```

	MovieId	Title	ReleaseYear	Rating	Genre
1	1	The Matrix	1999	10	Sci-Fi
2	2	The Godfather	1972	10	Drama
3	4	Toy Story	1995	8	Comedy
4	5	The Conjuring	2013	7	Horror
5	8	Die Hard	1988	8	Action

- Förklaring: Visar att tabellerna Movie och Genre är korrekt kopplade med en främmande nyckel.

Bild 2: UPDATE av "The Matrix" där betyget ändrades från 9 till 10.

```
USE MoviesLab2;
GO

-- Before
SELECT Title, Rating FROM dbo.Movie WHERE Title = 'The Matrix';

-- Update
UPDATE dbo.Movie
SET Rating = 10
WHERE Title = 'The Matrix';

-- After
SELECT Title, Rating FROM dbo.Movie WHERE Title = 'The Matrix';
```

	Title	Rating
1	The Matrix	9

	Title	Rating
1	The Matrix	10

- Förklaring: Demonstrerar att data kan uppdateras.

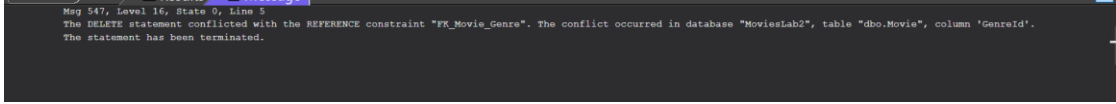
Bild 3: DELETE

Med FK:

```
USE MoviesLab2;  
GO
```

```
-- We try to delete a genre that still has movies assigned to it  
DELETE FROM dbo.Genre WHERE Name = 'Drama';
```

Vi får detta felmeddelande:



Msg 547, Level 16, State 0, Line 5
The DELETE statement conflicted with the REFERENCE constraint "FK_Movie_Genre". The conflict occurred in database "MoviesLab2", table "dbo.Movie", column 'GenreId'.
The statement has been terminated.

Msg 547, Level 16, State 0, Line 5
The DELETE statement conflicted with the REFERENCE constraint
"FK_Movie_Genre". The conflict occurred in database "MoviesLab2", table
"dbo.Movie", column 'GenreId'.
The statement has been terminated.

Utan FK:

Vi skapar en kopia och gör testet för att inte påverka originaltabellerna:

```
USE MoviesLab2;  
GO
```

```
-- 1) we create some temporary copies of Genre and Movie (without FK) so  
we can test it
```

```
IF OBJECT_ID('dbo.GenreTemp', 'U') IS NOT NULL DROP TABLE dbo.GenreTemp;  
IF OBJECT_ID('dbo.MovieTemp', 'U') IS NOT NULL DROP TABLE dbo.MovieTemp;  
GO
```

```
SELECT * INTO dbo.GenreTemp FROM dbo.Genre;  
SELECT * INTO dbo.MovieTemp FROM dbo.Movie;  
GO
```

```
-- 2) Confirm their counts
SELECT 'Genres before delete' AS Info, COUNT(*) AS CountRows FROM
dbo.GenreTemp;
SELECT 'Movies before delete' AS Info, COUNT(*) AS CountRows FROM
dbo.MovieTemp;

-- 3) Delete 'Drama' from GenreTemp this will succeed as we dont have any
FK here
DELETE FROM dbo.GenreTemp WHERE Name = 'Drama';

-- 4) Show Movies that now point to a missing GenreId
SELECT m.MovieId, m.Title, m.GenreId
FROM dbo.MovieTemp m
LEFT JOIN dbo.GenreTemp g ON m.GenreId = g.GenreId
WHERE g.GenreId IS NULL;
```

	Info	CountRows
1	Genres before delete	5

	Info	CountRows
1	Movies before delete	5

	MovieId	Title	GenreId
1	2	The Godfather	3

Vi kollar nu om genren "drama" fortfarande finns.

```
-- Vi kollar om GenreTemp innehåller genren 'Drama'
SELECT * FROM dbo.GenreTemp WHERE Name = 'Drama';

-- Listar alla rader i GenreTemp
SELECT GenreId, Name FROM dbo.GenreTemp ORDER BY GenreId;
```

Vi får:

	GenreId	Name
1	1	Action
2	2	Comedy
3	4	Sci-Fi
4	5	Horror

Förklaring: Referensintegritet – med och utan FK

Först försökte jag radera genren "Drama" i originaltabellen `Genre`. SQL Server stoppade detta med ett felmeddelande eftersom det fanns filmer som fortfarande pekade på genren via främmande nyckeln `FK_Movie_Genre`.

Därefter skapade jag kopior (`GenreTemp`, `MovieTemp`) utan främmande nyckel och raderade Drama där. Raderingen lyckades, men en kontroll med `LEFT JOIN` visade att t.ex. *The Godfather* hade `GenreId = 3` som inte längre fanns i `GenreTemp`. Det gav en föräldralös (**orphan**) post.

Slutsats: **Med FK** förhindras inkonsistens men **Utan FK** tillåts raderingen och lämnar orphan-poster, vilket leder till felaktig data.

- **Bild 4** — Northwind: uppgift 12 (Lakkalikööri → category)

```
USE Northwind;  
GO
```

```
SELECT p.ProductName, c.CategoryName, c.CategoryID  
FROM Products p  
JOIN Categories c ON p.CategoryID = c.CategoryID  
WHERE p.ProductName = 'Lakkalikööri';
```

	ProductName	CategoryName	CategoryID
1	Lakkalikööri	Beverages	1

- Bild 5** — Northwind: uppgift 16 (total order sum 1996-12-12, with discounts)

```
USE Northwind;  
GO
```

```
SELECT SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS  
TotalOrderSum  
FROM [Order Details] od  
JOIN Orders o ON od.OrderID = o.OrderID  
WHERE o.OrderDate = '1996-12-12';
```

	TotalOrderSum
1	1425.81996154785

Bild 6 — Northwind: Uppgift 19 (före/efter +20% för CategoryID = 4)

```

USE Northwind;
GO

-- Before
SELECT ProductName, UnitPrice AS OldPrice
FROM Products
WHERE CategoryID = 4
ORDER BY ProductName;

-- After (calculated only, not persisted)
SELECT ProductName,
       UnitPrice AS CurrentPrice,
       UnitPrice * 1.2 AS NewPrice
FROM Products
WHERE CategoryID = 4
ORDER BY ProductName;

```

	ProductName	OldPrice
1	Camembert Pierrot	34.00
2	Flotemysost	21.50
3	Geitost	2.50
4	Gorgonzola Telino	12.50
5	Gudbrandsdalsost	36.00
6	Mascarpone Fabioli	32.00
7	Mozzarella di Giov...	34.80
8	Queso Cabrales	21.00
9	Queso Manchego...	38.00
10	Raclette Courdava...	55.00

	ProductName	CurrentPrice	NewPrice
1	Camembert Pierrot	34.00	40.80000
2	Flotemysost	21.50	25.80000
3	Geitost	2.50	3.00000
4	Gorgonzola Telino	12.50	15.00000
5	Gudbrandsdalsost	36.00	43.20000
6	Mascarpone Fab...	32.00	38.40000
7	Mozzarella di Gio...	34.80	41.76000
8	Queso Cabrales	21.00	25.20000
9	Queso Mancheg...	38.00	45.60000
10	Raclette Courdav...	55.00	66.00000

Bild 7 — Northwind: Uppgift 21 lagrad procedure (+5%) med ROLLBACK

```
USE Northwind;
GO

-- Create or replace the procedure
IF OBJECT_ID('dbo.spRaisePrices', 'P') IS NOT NULL
    DROP PROCEDURE dbo.spRaisePrices;
GO
CREATE PROCEDURE dbo.spRaisePrices
AS
BEGIN
    UPDATE Products
    SET UnitPrice = UnitPrice * 1.05; -- +5%
END
GO

-- SAFE DEMO: run in a transaction and roll it back
BEGIN TRANSACTION;

SELECT TOP 5 ProductID, ProductName, UnitPrice AS BeforePrice
FROM Products
ORDER BY ProductID;

EXEC dbo.spRaisePrices;

SELECT TOP 5 ProductID, ProductName, UnitPrice AS AfterPrice
FROM Products
ORDER BY ProductID;

ROLLBACK TRANSACTION;

-- Verify rollback
SELECT TOP 5 ProductID, ProductName, UnitPrice AS PriceAfterRollback
FROM Products
ORDER BY ProductID;
```

	ProductID	ProductName	BeforePrice
1	1	Chai	18.00
2	2	Chang	19.00
3	3	Aniseed Syrup	10.00
4	4	Chef Anton's Cajun Seasoning	22.00
5	5	Chef Anton's Gumbo Mix	21.35

	ProductID	ProductName	AfterPrice
1	1	Chai	18.90
2	2	Chang	19.95
3	3	Aniseed Syr...	10.50
4	4	Chef Anton'...	23.10
5	5	Chef Anton'...	22.4175

	ProductID	ProductName	PriceAfterRollback
1	1	Chai	18.00
2	2	Chang	19.00
3	3	Aniseed Syr...	10.00
4	4	Chef Anton'...	22.00
5	5	Chef Anton'...	21.35

Bild 8 — Northwind: Uppgift 23 vy (Product, Category, Price) + SELECT

```

USE Northwind;
GO

-- Create or replace the view
IF OBJECT_ID('dbo.vProductCategoryPrice', 'V') IS NOT NULL
    DROP VIEW dbo.vProductCategoryPrice;
GO
CREATE VIEW dbo.vProductCategoryPrice
AS
SELECT
    p.ProductID,
    p.ProductName,
    c.CategoryName,
    p.UnitPrice AS Price
FROM dbo.Products AS p
JOIN dbo.Categories AS c ON p.CategoryID = c.CategoryID;
GO

-- Read from the view
SELECT TOP 10 *
FROM dbo.vProductCategoryPrice
ORDER BY ProductID;

```

	ProductID	ProductName	CategoryName	Price
1	1	Chai	Beverages	18.00
2	2	Chang	Beverages	19.00
3	3	Aniseed Syrup	Condiments	10.00
4	4	Chef Anton's Cajun Seasoning	Condiments	22.00
5	5	Chef Anton's Gumbo Mix	Condiments	21.35
6	6	Grandma's Boysenberry Spread	Condiments	25.00
7	7	Uncle Bob's Organic Dried Pears	Produce	30.00
8	8	Northwoods Cranberry Sauce	Condiments	40.00
9	9	Mishi Kobe Niku	Meat/Poultry	97.00
10	10	Ikura	Seafood	31.00

Bild 9 — Northwind: Uppgift 25 transaktion (2 uppdateringar) + forced error →
ROLLBACK

```
USE Northwind;
GO

-- före
SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID IN (1, 2)
ORDER BY ProductID;

BEGIN TRY
    BEGIN TRANSACTION;

    -- 2 uppdateringar
    UPDATE Products SET UnitPrice = UnitPrice + 1.00 WHERE ProductID = 1;
-- Chai
    UPDATE Products SET UnitPrice = UnitPrice + 2.00 WHERE ProductID = 2;
-- Chang

    -- Force error
    SELECT 1 / 0;

    COMMIT TRANSACTION; -- kommer inte hit pga error
END TRY
BEGIN CATCH
    IF XACT_STATE() <> 0
        ROLLBACK TRANSACTION;

    -- errorn visas
    SELECT ERROR_NUMBER() AS ErrorNumber, ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO

-- Efter matchar före pga rollback
SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID IN (1, 2)
ORDER BY ProductID;
```

	ProductID	ProductName	UnitPrice
1	1	Chai	18.00
2	2	Chang	19.00

(No column name)

	ErrorNumber	ErrorMessage
1	8134	Divide by zero error encountered.

	ProductID	ProductName	UnitPrice
1	1	Chai	18.00
2	2	Chang	19.00

6 Lösningens begränsningar

- **Litet omfång:** Databasen innehåller bara två tabeller (Genre och Movie) med ett litet antal attribut. Ett verkligt filmdatabassystem skulle behöva fler tabeller (t.ex. aktörer, regissörer, användarrecensioner).
- **Enkel validering:** Databasen har endast grundläggande begränsningar (NOT NULL, CHECK, FOREIGN KEY) används. Mer avancerad datavalidering och affärsregler saknas.
- **Begränsad datamängd:** Innehållet består av ett litet antal manuellt inlagda testposter. Databasen är inte skalbar för stora datamängder.
- **Ingen användargränssnitt (UI):** Systemet interageras enbart via SQL-frågor i Visual Studio. En riktig applikation skulle behöva ett grafiskt gränssnitt eller webbtjänst så personer kan använda databasen.
- **Ej optimerad för prestanda:** Ingen indexering, normalisering utöver 1NF/2NF, eller andra optimeringsmetoder har implementerats.

7 Problem och reflektioner

Under laborationen uppstod vissa svårigheter men också viktiga lärdomar:

- **Referensintegritet:**
Det var i början svårt att förstå varför man inte kunde ta bort en rad i en tabell som fortfarande hade kopplade poster i en annan tabell. Genom testet med att försöka ta bort genren *Drama* insåg jag att främmande nycklar skyddar databasen mot inkonsekventa data. Utan referensintegritet skulle “föräldrar” kunna raderas och “barnposter” lämnas kvar utan giltig koppling (s.k. orphan rows).
- **Risker utan rensning av relaterade poster:**
Om man tar bort en rad utan att först ta bort eller flytta relaterade poster kan databasen hamna i ett felaktigt tillstånd. Exempel: filmer som pekar på en genre som inte längre finns. Detta gör rapporter och analyser opålitliga.
- **Lagrade procedurer (fördelar/nackdelar):**
Fördelar är att procedurer är återanvändbara, kan optimeras av databashanteraren och ökar säkerheten (man kan ge rättighet att köra proceduren utan att ge direkt åtkomst till tabellen). Nackdelar är att de kan bli svåra att underhålla om många procedurer finns, och de är ofta specifika för en databas (mindre portabla).
- **Skillnaden mellan vy och procedur:**
En vy är en sparad SELECT som används för att förenkla återkommande frågor och för att ge användare begränsad åtkomst. En procedur kan däremot innehålla logik, ta parametrar och även ändra data (INSERT, UPDATE, DELETE).
- **TRY...CATCH i transaktioner:**
Jag insåg varför det är bättre än IF-satser. IF kan bara kontrollera villkor man själv tänker på, men oväntade fel (t.ex. division med noll) måste fångas automatiskt. TRY...CATCH garanterar att en rollback sker så att inga halva uppdateringar lämnas kvar. Detta är avgörande för databasens konsistens.
- **Git och versionshantering:**
Det tog lite tid att förstå arbetsflödet, men att lägga till alla .sql-filer i ett Git-repo gjorde det lättare att hålla ordning och att visa läraren en tydlig historik över arbetet.

8 Referenser

- Microsoft. (n.d.). *SQL Server documentation*. Hämtad från <https://learn.microsoft.com/en-us/sql/sql-server/>
- W3Schools. (n.d.). *SQL Tutorial*. Hämtad från <https://www.w3schools.com/sql/>
- Padron-McCarthy, T., & Risch, T. (2005). *Databasteknik* (2:a uppl.). Lund: Studentlitteratur. ISBN: 914404449.

9 Bilagor

Följande filer och material bifogas eller hänvisas till:

- **SQL-filer**
 - create_tables.sql – Skapande av tabellerna *Genre* och *Movie* i MoviesLab2.

-
- insert_data.sql – Infogning av testdata (genrer och filmer).
-
- SQLQuery1.sql – Skapande av databasen
 - queries_select_update_delete.sql – Demonstration av SELECT, UPDATE, DELETE och referensintegritet i MoviesLab2.
 - Uppgift2.sql – Övningsuppgifter 12, 16 och 19 i Northwind (med frågor och resultat).
 - Uppgift3.sql – Uppgifter 21, 23 och 25 (lagrad procedur, vy, transaktion med rollback).
- **ER-diagram**
 - er-diagram-movieslab2.png – ER-diagram för MoviesLab2
- **Rapportfil**
 - Denna rapport i PDF-format.