DBMSs – seminar 3 - extra doc

Examples we'd discuss (mainly at the board). Run the code and go through the explanations.

MyImdb

```
USE MyImdb
GO
IF OBJECT_ID('dbo.Review', 'U') IS NOT NULL
	DROP TABLE dbo.Review
IF OBJECT_ID('dbo.Movie', 'U') IS NOT NULL
	DROP TABLE dbo.Movie
IF OBJECT_ID('dbo.Reviewer', 'U') IS NOT NULL
	DROP TABLE dbo.Reviewer
GO

CREATE TABLE Movie
	(MovieID INT PRIMARY KEY IDENTITY(1,1),
	Title NVARCHAR(100),
	Director NVARCHAR(100),
	YearOfRelease SMALLINT,
	Nominations SMALLINT)

CREATE TABLE Reviewer
	(ReviewerID INT PRIMARY KEY IDENTITY(1,1),
	Name NVARCHAR(100))

CREATE TABLE Review
	(MovieID INT REFERENCES Movie(MovieID),
	ReviewerID INT REFERENCES Reviewer(ReviewerID),
	Stars TINYINT CHECK (Stars >= 0 AND Stars <=10),
	DateOfReview DATETIME2,
	PRIMARY KEY(MovieID, ReviewerID))
GO

INSERT INTO MOVIE (Title, Director, YearOfRelease, Nominations)
VALUES ('E.T. the Extra-Terrestrial', 'Steven Spielberg', 1982, 28),
	('Moscova nu crede in lacrimi', 'Vladimir Menshov', 1979, 1),
	('Close Encounters of the Third Kind', 'Steven Spielberg', 1977, 33),
	('Contact', 'Robert Zemeckis', 1997, 16),
	('Las Fierbinti', null, 2012, 0),
	('The Lord of the Rings: The Fellowship of the Ring', 'Peter Jackson',
2001, 90),
	('The Book Thief', 'Brian Percival', 2013, 10),
	('2001: A Space Odyssey', 'Stanley Kubrick', 1968, 6)

INSERT INTO Reviewer (Name)
VALUES ('Cristian Tudor Popescu'), ('Magda Mihailescu'), ('Irina Margareta
Nistor')

INSERT INTO Review (MovieID, ReviewerID, Stars, DateOfReview)
VALUES(1,1,6,'1-13-2014'),
	(1,2,7,'12-31-2013'),
	(2,1,10,'1-12-2014'),
      (2,2,10,'1-10-2014'),
      (2,3,10,'2-13-2014'),
      (3,1,9,'2-25-2014'),
      (3,3,8,'11-30-2014'),
      (4, 1, 9, '1-1-2014'),
      (4, 2, 9, '2-2-2014'),
      (4, 3, 10, '3-3-2014'),
      (5,2,0,'1-1-2014'),
      (5,3,1,'2-2-2014')
--
GO
SELECT * FROM Movie
SELECT * FROM Reviewer
SELECT * FROM Review
```

Transactions & Concurrency Management in SQL Server

*** <u>explicit transactions</u> - statements**

- BEGIN TRAN
  - marks the beginning of an explicit transaction
  - increments @@TRANCOUNT by 1
- COMMIT TRAN
  - marks the end of a successful transaction
  - decrements @@TRANCOUNT
  - if @@TRANCOUNT = 1, changes are made permanent in the DB
- ROLLBACK TRAN
  - rolls back a transaction (to its beginning or to a save point inside it)

- @@TRANCOUNT – number of BEGIN TRAN statements executed on the current connection

- example:

```
--two rows with Title = Close Encounters of the Third Kind and Director = wrong
director
SET NOCOUNT ON

PRINT '@@TRANCOUNT before ''BEGIN TRANSACTION'': ' + CAST(@@TRANCOUNT AS
NVARCHAR(2)) --0
BEGIN TRANSACTION
  PRINT @@TRANCOUNT --1

  UPDATE Movie
  SET Director = 'Steven Spielberg'
  WHERE UPPER(Title) = 'CLOSE ENCOUNTERS OF THE THIRD KIND'
--now the Director value for both Close Encounters ... movies is Steven Spielberg

  IF @@ROWCOUNT > 1
  BEGIN
    PRINT 'Your data has issues, rolling back tran.'
    ROLLBACK TRAN
  END
  ELSE
  BEGIN
    PRINT 'About to commit.'
    COMMIT TRANSACTION
  END
GO

PRINT @@TRANCOUNT --0
--tran rolled back, so Director still = wrong director
```

* implicit transactions

- SET IMPLICIT_TRANSACTIONS ON
  - system is in *implicit transaction* mode, i.e., if @@TRANCOUNT = 0, statements like *INSERT*, *UPDATE*, *DELETE,* etc, begin a new transaction

- example:

```
SET IMPLICIT_TRANSACTIONS OFF
INSERT INTO Movie VALUES ('The Book Thief', 'Brian Percival', 2013, 10)
--individual statement that is committed if it completes successfully (autocommit
transaction)
PRINT @@TRANCOUNT --0
SELECT * FROM Movie --Book Thief is in the result set
--trying to execute COMMIT TRAN => err: The COMMIT TRANSACTION request has no
corresponding BEGIN TRANSACTION.
```

- by contrast:

```
SET IMPLICIT_TRANSACTIONS ON
INSERT INTO Movie VALUES ('The Book Thief', 'Brian Percival', 2013, 10)
PRINT @@TRANCOUNT --1
ROLLBACK TRAN -- executes successfully (so would COMMIT TRAN)
--SELECT * FROM Movie - no row was inserted
PRINT @@TRANCOUNT --0
```

* rolling back transactions automatically

- SET XACT_ABORT ON
  - the current transaction is automatically rolled back when a statement raises a run-time error

- example:

```
--suppose there is a movie with MovieID=1 in the Movie table, but there's no movie with
MovieID=100
--there is a reviewer with ReviewerID=3 in Reviewer

SET XACT_ABORT OFF;
BEGIN TRAN
  INSERT INTO Review(MovieID, ReviewerID, Stars) VALUES (100,3,5) --FK violation error
  INSERT INTO Review(MovieID, ReviewerID, Stars) VALUES (1,3,5)   --executes successfully
COMMIT TRAN                                                       --successfully commits


SET XACT_ABORT ON;
BEGIN TRAN
  INSERT INTO Review(MovieID, ReviewerID, Stars) VALUES (100,3,5)
  INSERT INTO Review(MovieID, ReviewerID, Stars) VALUES (1,3,5)
COMMIT TRAN
--nothing is inserted
--the transaction was automatically rolled back since the 1st INSERT raised a run-time
error
```

* nesting transactions (but not nested transactions)

- example:

```
BEGIN TRAN
  INSERT Movie (Title, YearOfRelease)
  VALUES ('It''s a mad, mad, mad, mad world', 1963)

  BEGIN TRAN
    PRINT @@TRANCOUNT--2
    INSERT Movie (Title, Director, YearOfRelease)
    VALUES('The Hobbit: The Desolation of Smaug', 'Peter Jackson', 2013)
  COMMIT TRAN                --nothing is committed, @@TRANCOUNT is decremented to 1

  PRINT @@TRANCOUNT  --1

COMMIT TRAN                  --commits outer tran
PRINT @@TRANCOUNT    --0

--both The Hobbit... and It's a mad... are now in the DB
```

- remember, COMMIT makes the changes a permanent part of the DB if @@TRANCOUNT is 1.

- replace the 1st COMMIT TRAN with ROLLBACK TRAN; observe the effects in the DB:

```
BEGIN TRAN
  INSERT Movie (Title, YearOfRelease)
  VALUES ('It''s a mad, mad, mad, mad world', 1963)

  BEGIN TRAN
    PRINT @@TRANCOUNT--2
    INSERT Movie (Title, Director, YearOfRelease)
    VALUES('The Hobbit: The Desolation of Smaug', 'Peter Jackson', 2013)
  ROLLBACK TRAN              --everything is rolled back

  PRINT @@TRANCOUNT --0

COMMIT TRAN  --err: The COMMIT TRANSACTION request has no corresponding BEGIN
TRANSACTION.
```

* named savepoints

- what do you think the result of executing the following code will be? run and explain:

```
BEGIN TRAN
  INSERT Movie (Title, YearOfRelease)
  VALUES ('It''s a mad, mad, mad, mad world', 1963)

  SAVE TRAN InsrtMovieSvp

  INSERT Movie (Title, YearOfRelease)
  VALUES('The Hobbit: The Desolation of Smaug', 2013)
  INSERT Movie (MovieID, Title, YearOfRelease)
  VALUES(100, '12 Angry Men', 1957)

  IF (@@ERROR <> 0)
    ROLLBACK TRAN InsrtMovieSvp

  PRINT @@TRANCOUNT --1
COMMIT TRAN
```

-the Movie table should now include *It's a mad...*; why? hint – remember the Movie create table statement.

* locks

To execute a write operation (e.g., I/U/D), a transaction must acquire an exclusive (write) (X) lock. This lock is released at the end of the transaction.
Shared (read) locks (S) can be acquired for read operations; they can be released at the end of the operation or at the end of the transaction.

S/X etc – lock modes

Lock modes compatibility matrix:

	S	x
S	Yes	No
X	No	No

Locks can be acquired at different granularity levels, e.g., index key, row, table, etc.

Check locks in the (very cool) Dynamic Management View `sys.dm_tran_locks.`

!Questions to ask when analyzing locks: what's the granularity level of the lock?, what's its lock mode?, when will the lock be released?

* isolation levels

Isolation levels are concurrency management policies, specifying the degree to which a transaction should be isolated from changes made by other transactions. Isolation levels control (among other things) the behavior of read operations; they:
- specify if S locks are acquired on resources read by a transaction;
- specify how long S locks are held, e.g., they could be released at the end of the read operation or at the end of the transaction.

Read uncommitted
- lowest isolation level
- no S locks when reading data
- as a result, it allows dirty reads (a transaction can see uncommitted changes made by another ongoing transaction).

Read committed (pessimistic)
- default isolation level in SQL Server
- to read a resource, a transaction must acquire an S lock on it; this lock is released at the end of the operation (not at the end of the transaction);

- as a result, dirty reads can't occur; e.g., to update row R, tran T_1 acquires an X lock on it; now if tran T_2 tries to read the same row before T_1 commits or rolls back, it will ask for an S lock on it, but S locks are incompatible with X locks, so T_2 is suspended; T_2 waits for the X lock on R to be released, i.e., for T_1 to commit / rollback;
- non-repeatable reads and phantoms can occur.

Repeatable Read
- shared locks are held until the end of the transaction;
- non-repeatable reads can't occur; e.g., to read row R, tran T_1 acquires an S lock on it that lasts until the end of the transaction; if tran T_2 now tries to change the same row, it attempts to acquire an X lock on it, which is incompatible with the existing S lock acquired by T_1 (so T_2 cannot change the row until T_1 releases the S lock);
- it allows phantoms to occur.

Serializable
- strongest isolation level
- holds S locks until the end of the transaction + locks data that doesn't exist (key range locks), so phantoms cannot occur anymore.

* <u>anomalies – example on Dirty reads</u>
(non-repeatable reads, phantoms, deadlocks – similar; adapt examples in the lecture notes)

Before running the example below, make sure you don't have other active transactions (`PRINT @@TRANCOUNT`).

Dirty reads – a transaction is able to read data that has been changed by another ongoing transaction.
Assume record (MovieID, Director) with values (5, NULL) in the Movie table.

1. SQL Server Management Studio -> File -> New -> Query with Current Connection (2 times => C1 and C2)

2. C1:
- check isolation level in C1 (the isolation level for this session doesn't really matter, from the lowest to the highest, the dirty read will occur only if in C2 we set the isolation level to READ UNCOMMITTED):
```
SELECT transaction_isolation_level
FROM sys.dm_exec_sessions
WHERE session_id = @@SPID
--1 read uncommitted
--2 read committed
--3 repeatable read
--4 serializable
```

- code transaction T1 that updates data, then rolls back:
e.g.,
```
BEGIN TRAN
       UPDATE Movie SET Director = 'unknown'
       WHERE Director IS NULL
ROLLBACK TRAN
```

3. C2:
- set isolation level to READ UNCOMMITTED:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

- code transaction T2 that reads data and commits (no S locks under this IL):
e.g.,

```
BEGIN TRAN
       SELECT * FROM Movie
COMMIT TRAN
```

4. Dirty read:
- run the update in T1 (i.e., execute BEGIN TRAN + UPDATE statement), don't rollback yet;
- run the select in T2 (i.e., execute BEGIN TRAN + SELECT statement); T2 can read (MovieID, Director) = (5, 'unknown'), i.e., dirty data (T1 will rollback).

5. Check the locks right after the update operation, and then after the rollback in T1 with the following select:

```
SELECT resource_type, resource_database_id, request_mode, request_type, request_status,
request_session_id, request_owner_type, request_owner_id
FROM sys.dm_tran_locks
WHERE resource_database_id = (select database_id from sys.databases where name =
'MyImdb')
--or use DB_ID()
```

Ex. Join with `sys.dm_tran_active_transactions` and display the begin time and name of the transaction that acquires the locks.

6. Change isolation level in C2 to READ COMMITTED (then to REPEATABLE READ, SERIALIZABLE). Even if the isolation level in C1 is READ UNCOMMITTED, the dirty read does not occur. T2 is suspended when trying to read because it must acquire an S lock that conflicts with an existing X lock. T2 resumes execution only after T1 rolls back.