www.DBTechNet.org

Concurrency Lab 1 - SQL Server Report

Disclaimers

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Trademarks of products mentioned are trademarks of the product vendors.

CONTENTS

Introducing the Lab....1
First Experiments with SQL Transactions....2
Savepoints and Partial Rollbacks....3
Basic Concurrency Anomalies....5
Integrity Scenarios....16
Referential Parent-Child Scenarios....17
Metadata Concurrency Control....21

Introducing the Lab

This lab is based on the DBTech tutorial on "SQL Concurrency Technologies" available at http://www.DBTechNet.org/papers/SQL_ConcurrencyTechnologies.pdf and its collection of exercises "Concurrenry Lab 1" at http://www.DBTechNet.org/labs/ccr_lab/CCLab1.pdf .
The tutorial covers the basic theory of single-connection flat SQL transactions and concurrency control mechanisms used in the commercial mainstream RDBMS systems, from point of view of database professionals working in the ICT industry.

This report applies the exercises of "Concurrency Lab 1" to SQL Server 2008, assuming that the user has unlimited access to the server. SQL Server 2008 Express can be downloaded for free from the web site of Microsoft. Most exercises include results from our test runs, helping you to repeat and verify the results yourself. If you trust on these “paperware” results, you may try to answer the questions based on your knowledge and the listed results only, but we recommend that you verify the results experimenting with the examples yourself. The report can be used as a model on how the exercise collection can be applied to the other DBMS systems, such as DB2, Oracle, or MySQL/InnoDB even if the SQL implementations and the results on these DBMS systems may differ from the results in this report.

First Experiments with SQL Transactions

When you open an SQL connection to a relational database you will start a SQL session. All reliable database accessing should be transactional, but in many case the default data access mode is autocommit which means that every SQL command is committed automatically and you cannot rollback anything. To use transactions in SQL Server you need explicitly start every transaction by entering command `BEGIN TRANSACTION` or configuring your opened SQL session to use implicit transactions by command `SET IMPLICIT_TRANSACTIONS ON`.

Let's start with a minimalistic table T. In case of SQL Server we usually don't need to use semicolons at end of statements.

```
SET IMPLICIT_TRANSACTIONS ON
DROP TABLE T
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(20))
COMMIT
INSERT INTO T (id, s) VALUES (1, 'value 1')
INSERT INTO T (id, s) VALUES (2, 'value 2')
SELECT * FROM T  -- query 1
COMMIT
INSERT INTO T (id, s) VALUES (3, 'something else')
UPDATE T SET s = 'new value' WHERE id = 1
SELECT * FROM T   -- query 2
ROLLBACK
SELECT * FROM T  -- query 3
```

Note: A sequence of two hyphens (--) in SQL command will mark rest of the line as a comment.

We don't list results of the first 2 transactions and below we start listing by the third INSERT command of the experiment, testing effects of COMMIT and ROLLBACK as follows:

```
INSERT INTO T (id, s) VALUES (3, 'something else')

(1 row(s) affected)

UPDATE T SET s = 'new value' WHERE id = 1

(1 row(s) affected)

SELECT * FROM T   -- query 2
id          s
----------- --------------------
1           new value
2           value 2
3           something else

(3 row(s) affected)

ROLLBACK

SELECT * FROM T  -- query 3
id          s
----------- --------------------
1           value 1
2           value 2

(2 row(s) affected)
```

Savepoints and Partial Rollbacks

According to ISO SQL standard it is possible to define multiple savepoints inside a flat transaction and during the transaction rollback all those modifications which the transaction has made in the database after a certain savepoint in the transaction, without rollback of those modifications which were made before that savepoint. The syntax of using savepoints differs between implementations.

Let's experiment with the SQL Server implementation of savepoints using the following simple example

```
SET NOCOUNT ON
BEGIN TRANSACTION
  select 'Initial contents in table T ..'
  SELECT * FROM T
  select 'Inserting a new row, id=5 ..'
  INSERT INTO T VALUES (5,'a new row')
  select 'Setting savepoint1 ..'
SAVE TRANSACTION savepoint1
  select 'Updating the inserted row id=5 ..'
  UPDATE T SET s='updated value' WHERE id=5
  select 'Setting savepoint2 ..'
SAVE TRANSACTION savepoint2
  select 'Current contents in table T ..'
  SELECT * FROM T
  select 'Rollback to savepoint1 ..'
ROLLBACK TRANSACTION savepoint1
  select 'Current contents in table T ..'
  SELECT * FROM T;
-- RELEASE SAVEPOINT savepoint1 is not supported in Transact-SQL
  VALUES 'Inserting a new row, id=6 ..'
  INSERT INTO T VALUES (6,'the latest row')
  VALUES 'Rollback to savepoint2 ?'
ROLLBACK TRANSACTION savepoint2
  select 'Rollback the whole transaction ..'
ROLLBACK;
GO
--
```

When you try this Transact-SQL script, you should get the following results

```
------------------------------
Initial contents in table T ..

id          s
----------- --------------------
1           value 1
2           value 2


---------------------------
Inserting a new row, id=5 ..


--------------------
Setting savepoint1 ..


--------------------------------
Updating the inserted row id=5 ..


--------------------
Setting savepoint2 ..
```

```
------------------------------
Current contents in table T ..

id          s
----------- --------------------
1           value 1
2           value 2
5           updated value


-------------------------
Rollback to savepoint1 ..


------------------------------
Current contents in table T ..

id          s
----------- --------------------
1           value 1
2           value 2
5           a new row


----------------------------
Inserting a new row, id=6 ..


------------------------
Rollback to savepoint2 ?

Msg 6401, Level 16, State 1, Line 23
Cannot roll back savepoint2. No transaction or savepoint of that name was found.

---------------------------------
Rollback the whole transaction ..

------------------------------
Current contents in table T ..

id          s
----------- --------------------
1           value 1
2           value 2
```

Rollback to savepoint2 fails since savepoint2 is "nested" to savepoint1, which means it has been defined after savepoint1, and after the definition the rollback to savepoint1 also rolls back the definition of savepoint2.

Note: According to SQL Server Books Online, savepoints are not supported in distributed Transact-SQL transactions.

Basic Concurrency Anomalies

For these exercises we need to start another concurrent session in a new Query window of SQL Server Management Studio, in which we can arrange the concurrent windows into a Vertical Tab Group as shown in Fig 5. Let's call the first session as Session A and the new one as Session B.

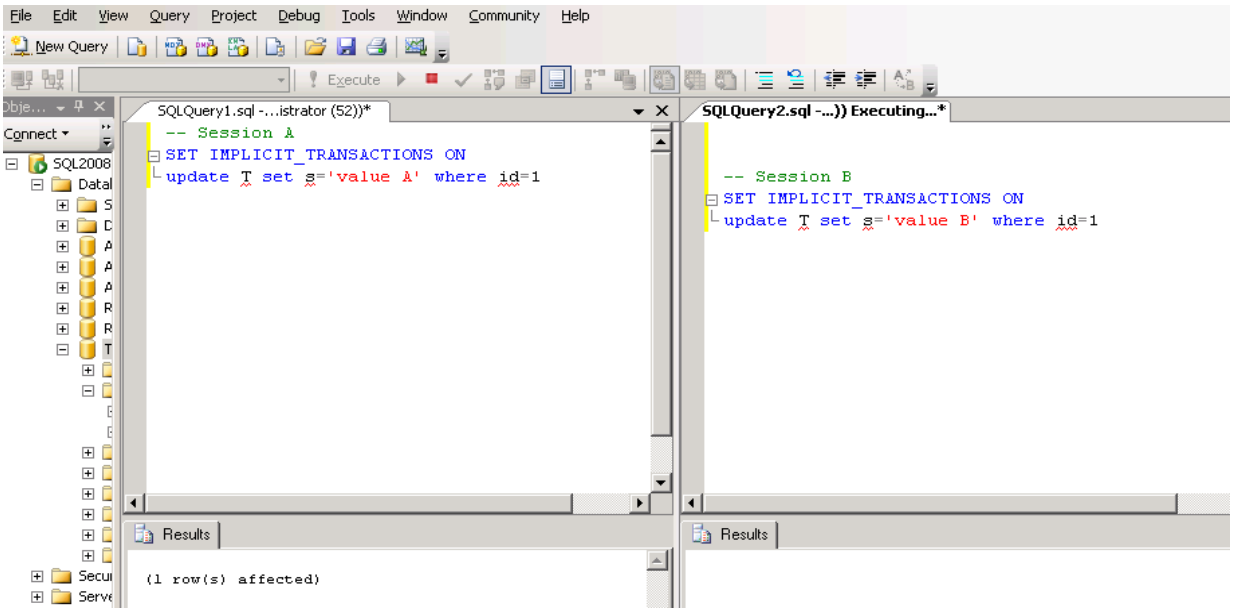

Figure 5

Now we can experiment with the following scenarios proceeding according to the step order in the scenario tables, and in case one of the 2 sessions stucks into waiting state, we should take notes of the situation, and then proceed with the other session until the stucked session can proceed again.

Notes:

In the following commands we have omitted the semicolons as command separators, so for example on using Oracle you need to add them.

Just for readability, we use lower case letters for the SQL reserved words in the textboxes below.

We assume implicit transactions in the following scenarios, starting sessions with the command
SET IMPLICIT TRANSACTIONS ON.
We will usually ROLLBACK every transaction to preserve the original content in the table after every experiment.

The commands in grey are not part of the essential behavior of the topic, but supplied as assisting means only.

Let's try concurrent updates first

Exercise 1

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2	`update T set s='value A' where id=1` `(1 row(s) affected)`	
3		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
3.1		
4		`update T set s='value B' where id=1` Executing query...

5	`select * from T where id=1`
	`id s`
	`---------- --------------------`
	`1 value A`
	
	`(1 row(s) affected)`	

6	`commit` `Command(s) completed successfully.`	 `(1 row(s) affected)`

7		`select * from T where id=1`
		`id s`
		`---------- --------------------`
		`1 value B`
		
		`(1 row(s) affected)`

8		`commit` `Command(s) completed successfully.`

9		`select * from T where id=1`
		`id s`
		`---------- --------------------`
		`1 value B`
		
		`(1 row(s) affected)`

10		`rollback` `Command(s) completed successfully.`

Note: Executing query... If the session is waiting for some locks, this display will stay in the bottom of the result pane of the query window in SQL Server Management Studio.

MCQ: Which claims in the following lists are true?

1. Resultsets in steps 5, 7 and 9 are equal
2. Resultsets 5 and 7 are different
3. A lost update problem appears in steps 4
4. A blind overwriting appears in step 4-6
5. A lost update problem appears in step 8

Answer: On using SQL Server, the claim number 4 is true.

Please comment the result:
Since overwriting of session A's update succeeds after its commit, this is not the classical lost update problem, but due to the form used in the update command of session B, it is a blind

owerwriting, as session B does not check if a concurrent transaction has just updated the same row. The solution of blind overwriting problem is the topic of RVV Paper, the next tutorial in our series of concurrency papers.

Exercise 1b

Try to use the SQL Server specific SET LOCK_TIMEOUT commands to set the timeout to 2000 milliseconds in step 3.1. How does this change the flow of the scenario?

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2	`update T set s='value A' where id=1` `(1 row(s) affected)`	
3		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
3.1		`SET LOCK_TIMEOUT 2000`

4		`update T set s='value B' where id=1`
		Executing query...
		`Msg 1222, Level 16, State 51, Line 2`
		`Lock request time out period exceeded.`
		`The statement has been terminated.`

5	`select * from T where id=1`
	`id s`
	`---------- --------------------`
	`1 value A`
	`(1 row(s) affected)`	

6	`commit` `Command(s) completed successfully.`	

7		`select * from T where id=1`
		`id s`
		`---------- --------------------`
		`1 value A`
		`(1 row(s) affected)`

8		`commit` `Command(s) completed successfully.`

9		`select * from T where id=1`
		`id s`
		`---------- --------------------`
		`1 value A`
		`(1 row(s) affected)`

10		`rollback` `Command(s) completed successfully.`

Answer: Due to the short timeout of the lock request in step 4, the update command is rolled back, but this will not rollback the whole transaction of session B. So if the transaction of session B does not react on this error, it would not work according to the plan of the application developer, - and if the transaction of session B would have included some other modifications to the database contents, the database would get into an inconsistent state, loosing information of the actual business!

Exercise 2

Now, let's change the scenario of exercise 1 a bit, including isolation level SERIALIZABLE and SELECT commands:

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2	`set transaction isolation level serializable`	

3	`select * from T where id=1`
	`id s`
	`---------- --------------------`
	`1 value B`
	
	`(1 row(s) affected)`	

4		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
5		`set transaction isolation level serializable`

6		`select * from T where id=1`
		`id s`
		`---------- --------------------`
		`1 value B`
		
		`(1 row(s) affected)`

7	`update T set s='value A2' where id=1` Executing query...	
7.1		`sp_lock -- *) see results 2-7.1`

8		`update T set s='value B2' where id=1`
		Executing query...
		`Msg 1205, Level 13, State 51, Line 1`
		`Transaction (Process ID 53) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.`

9	`commit` `Command(s) completed successfully.`	

10		`commit`
		`Msg 3902, Level 16, State 1, Line 1`
		`The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.`

11		`select * from T where id=1`
		`id s`
		`---------- --------------------`
		`1 value A2`
		
		`(1 row(s) affected)`

*) Results 2-7.1, Sp_lock results of step 7.1 above

Using the system procedure SP_LOCK, a user with administrator privileges may get the following simple snapshot of current locks :

Results | Messages

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	53	5	0	0	DB		S	GRANT
2	53	5	357576312	1	KEY	(8194443284a0)	S	GRANT
3	53	5	357576312	1	KEY	(8194443284a0)	X	CNVT
4	53	5	357576312	0	TAB		IX	GRANT
5	53	5	357576312	1	PAG	1:151	IX	GRANT
6	54	5	0	0	DB		S	GRANT
7	56	1	1131151075	0	TAB		IS	GRANT
8	56	5	357576312	1	PAG	1:151	IS	GRANT
9	56	5	357576312	0	TAB		IS	GRANT
10	56	5	357576312	1	KEY	(8194443284a0)	S	GRANT
11	56	5	0	0	DB		S	GRANT
12	57	5	0	0	DB		S	GRANT

The columns included:

spid	process id of the session
dbid	internal number of the database in the current server instance
ObjId	internal number of the object, for example a table object
IndId	internal number of the locked index of the object, 0 for other objects. Note: since we defined primary key for our table, as default the table pages are actually allocated on the leaf pages of the clustered index of the primary key.
Type	DB, TAB, PAG, KEY, RID, etc
Resource	Page ID, RID, etc
Mode	S, IS, X, IX, U, IU, SIX, ..

For more details see **sp_lock** documentation in SQL Server Books Online.

The report format of sp_lock is fixed, and instead of calling sp_lock Microsoft recommends SELECT statements to be applied to the new dynamic management view **sys.dm_tran_locks** to report only the selected locks and columns of interest. However, in our concurrency tests we only have the two concurrent sessions at a time, so we can manage using the easy to use sp_lock requests.

Which claims in the following lists are true?

1. Resultsets in steps 3 and 11 are equal
2. Resultsets 3 and 11 are different
3. A lost update problem appears in step 8
4. A deadlock appears in step 8
5. A livelock appears in step 10

Answer: The claim numbers 2 and 8 are true.

Commenting the result:
A deadlock is detected in step 8, and in this case the DBMS has selected the transaction of session B as the victim of the deadlock and rolled it back. Since we were using implicit transactions, the COMMIT command in step 10 cannot be applied to any active transaction and raises an error.

Exercise 2b

If we apply above the default isolation level READ COMMITTED, then acquired S-locks are released automatically after every SELECT command, and thus the experiment behaves like exercise 1 above leading to Blind Overwriting problem.

Exercise 3 UPDATE-UPDATE scenario of multiple updates in conflicting order

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	

2	`update T set s='value A1'`
	`where id=1`
	`(1 row(s) affected)`	

3		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
4		`update T set s='value B1' where id=2` `(1 row(s) affected)`

5	`update T set s='value A2'`
	`where id=2`
	Executing query...	

6		`update T set s='value B2' where id=1` Executing query...
6.1	`Msg 1205, Level 13, State 51, Line 1` `Transaction (Process ID 53) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.`	`(1 row(s) affected)`

7	`commit`
	`Msg 3902, Level 16, State 1, Line 1`
	`The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.`	

8		`commit` `Command(s) completed successfully.`

9		`select * from T`
		`id s`
		`---------- --------------------`
		`1 value B2`
		`2 value B1`
		
		`(2 row(s) affected)`

10		`rollback`

Which claims in the following lists are true?

1. A blind overwriting problem appears in step 4
2. A lost update problem appears in step 5
3. A deadlock appears in step 4
4. A deadlock appears in step 6
5. A lost update problem appears in step 6

Answer: The claim number 4 is is true.

Comment: Deadlock occurs at step 6. After some 2 seconds, at step 6.1, the deadlock detector of SQL Server detects the deadlock, and in this case selects the transaction of session A as the victim of the deadlock, and rolls it back. The transaction of session B continues and commits at step 8.

Exercise 3b

Selection of the victim is an internal decision of the RDBMS. However, in SQL Server we can influence the decision, if we set up the DEADLOCK_PRIORITY. Some other DBMS systems have implemented this kind of transaction tuning options, but DB2 and Oracle don't have this kind of transaction tuning options.

Rerun exercise 3 using the SQL Server specific command SET DEADLOCK_PRIORITY HIGH in beginning of the session which was selected as the victim in exercise 3.

In the following experiment, we prioritize the transaction of session A as more urgent, so the transaction of session B will lose competition in case it gets into deadlock with transaction of session A. Since the task of session B may also be important, session B should retry its transaction after the automatic rollback.

step	Session A	Session B

1	`-- in SQL Server:`
	`SET IMPLICIT_TRANSACTIONS ON`
	
	SET DEADLOCK_PRIORITY HIGH	

2	`update T set s='value A1'`
	`where id=1`
	`(1 row(s) affected)`	

3		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
4		`update T set s='value B1' where id=2` `(1 row(s) affected)`

5	`update T set s='value A2'`
	`where id=2`
	Executing query...	

6		`update T set s='value B2' where id=1` Executing query...
6.1	`(1 row(s) affected)`	`Msg 1205, Level 13, State 51, Line 1` `Transaction (Process ID 55) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.`
7	`commit` `Command(s) completed successfully.`	

8		`commit`
		`Msg 3902, Level 16, State 1, Line 1`
		`The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.`

9		`select * from T`
		`id s`
		`---------- --------------------`
		`1 value A1`
		`2 value A2`
		
		`(2 row(s) affected)`

10		`retry of the transaction?`

Basic Anomalies

Let's restore contents of the table T as follows

```
DELETE FROM T
INSERT INTO T (id, s) VALUES (1, 'value 1')
INSERT INTO T (id, s) VALUES (2, 'value 2')
COMMIT
```

Try the following scenarios, and fix the isolation levels in them.

Exercise 4

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2	`select * from T where id=1` id s ---------- ---------- 1 value 1 (1 row(s) affected)	

3	`update T set s='value A'`
	`where id=1`
	
	(1 row(s) affected)	

4		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
5		`set transaction isolation level` `read uncommitted`
6		`select * from T where id=1` id s ---------- ---------- 1 value A (1 row(s) affected)
7	`rollback` Command(s) completed successfully.	
8		`select * from T where id=1` id s ---------- ---------- 1 value 1 (1 row(s) affected)
9		`rollback`

Which claims in the following lists are true?

1. Resultsets in steps 2 and 6 are equal
2. Resultsets 2 and 8 are equal
3. A lost update problem appears in step 7
4. A dirty read appears in step 6
5. A dirty read appears in step 8

Answer: The claim numbers 2 and 4 are true.

Comment: ISO SQL isolation level read uncommitted allows the transaction of session B to read the requested row without waiting for the covering S-lock, so it will get non-committed data, which eventually will be rolled back by the writer, session A. Transaction of session B should be fixed to use at least the isolation level read committed to avoid the dirty read problem.

Note: Dirty read is not possible in Oracle, due to MVCC concurrency control, in which read uncommitted isolation level is not possible.

Inconsistent Read Problems

Exercise 5 Changing/vanishing read set

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
3		`set transaction isolation level` `read committed`

4		`select * from T`
		`id s`
		`---------- --------------------`
		`1 value 1`
		`2 value 2`
		
		`(2 row(s) affected)`

5	`update T set s='value A'`
	`where id=1`
	`(1 row(s) affected)`	

6	`commit` `Command(s) completed successfully.`	

7		`select * from T`
		`id s`
		`---------- --------------------`
		`1 value A`
		`2 value 2`
		
		`(2 row(s) affected)`

8		`rollback` `Command(s) completed successfully.`

9	`--let's restore the content`
	`update T set s='value 1'`
	`where id=1`
	`update T set s='value 2'`
	`where id=2`	

10	`commit`	

Which claims in the following lists are true?

1. Resultsets in steps 4 and 7 are not equal
2. Resultsets in steps 4 and 7 are equal
3. A dirty read appears in step 4
4. A non-repeatable read appears in step 4
5. A non-repeatable read appears in step 7

Answer: The claim numbers 1, 4 and 5are true.
Comment: Isolation level read committed of session b's transaction provides covering S-lock for reading in steps 4 and 7, but it releases the S-locks immediately after the reading. So session A can update the row between the steps 4 and 7. Some selected rows could also have been deleted meanwhile. Also the select in step 7 is non-repeatable in the sense that if the transaction of B would repeat the read, some competing transaction might have changed the content in database again before the next select.

If the transaction of session B should get the same result both in steps 4 and 7, then transaction of B should use the ISO SQL isolation level repeatable read or serializable instead of read committed.

Exercise 6 Increasing read set

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
3		`set transaction isolation level repeatable read`

4		`select count(*) as count from T`
		`count`
		`-----------`
		`2`
		
		`(1 row(s) affected)`

5	`insert into T (id, s) values (3, 'value 3')`
	
	`(1 row(s) affected)`	

6	`commit` `Command(s) completed successfully.`	

7		`select count(*) as count from T`
		`count`
		`-----------`
		`3`
		
		`(1 row(s) affected)`

8		`rollback` `Command(s) completed successfully.`

9	`--let's restore the content`
	`delete T where id=3`
	
	`(1 row(s) affected)`	

10	`commit` `Command(s) completed successfully.`	

Which claims in the following list are true?

1. Results in steps 4 and 7 are equal
2. A non-repeatable read appears in step 4
3. A phantom problem appears in step 5
4. A phantom problem appears in step 7
5. A non-repeatable read appears in step 7

Answer: The claim number 7 is true.
Explanation: ISO SQL isolation level repeatable read guarantees that no rows which have been read during the transaction can be changed or deleted by concurrent transaction. However, it is possible that transaction itself changes or deletes some of the rows of the read set, and transaction itself or some concurrent transactions can insert new rows or change some rows which were not selected before, so that some following read operation of the transaction will get more rows in the result set.

If the transaction of session B should get the same result both in steps 4 and 7, it should use the ISO SQL isolation level serializable instead of the repeatable read.

Integrity Scenarios

Unique

Exercise 7

step	Session A	Session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2	`delete from T where id=1` `(1 row(s) affected)`	
3		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
4		`insert into T (id, s) values (1, 'new row')` Executing query...)
4.1		

5	`rollback`
	`Command(s) completed successfully.`
		`Msg 2627, Level 14, State 1, Line 1`
		`Violation of PRIMARY KEY constraint 'PK__T__58D1301D'. Cannot insert duplicate key in object 'dbo.T'.`
		`The statement has been terminated.`

6		`commit` `Command(s) completed successfully.`

Explain why INSERT command of step 4 fails in step 5 and COMMIT command in step 6 succeeds.

Answer:
Looking at SP_LOCK results in step4.1

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	53	5	0	0	DB		S	GRANT
2	53	5	1157579162	1	KEY	(8194443284a0)	X	GRANT
3	53	5	1157579162	1	PAG	1:121	IX	GRANT
4	53	1	1131151075	0	TAB		IS	GRANT
5	53	5	1157579162	0	TAB		IX	GRANT
6	54	5	0	0	DB		S	GRANT
7	55	5	1157579162	0	TAB		IX	GRANT
8	55	5	0	0	DB		S	GRANT
9	55	5	1157579162	1	KEY	(8194443284a0)	X	WAIT
10	55	5	1157579162	1	PAG	1:121	IX	GRANT
11	56	5	0	0	DB		S	GRANT

we can see that A still has X-lock on the deleted row, and B needs to wait for its competing X-lock on the row. Rollback of A will release the lock, and B fails in writing a row that violates the primary key integrity of the recovered row.

Referential Parent-Child Scenarios

Now, try the following scenario ”Contention as side effect of referential integrity checks” (Parent-Child) of the tutorial ”SQL Concurrency Technologies” using SQL Server.

```
CREATE TABLE Parent (
pid INT NOT NULL PRIMARY KEY,
pv  INT
);
CREATE TABLE Child (
cid INT NOT NULL PRIMARY KEY,
cv  INT,
fk  INT,
FOREIGN KEY (fk) REFERENCES Parent
);
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Parent VALUES (1, 0);
INSERT INTO Child VALUES (1, 4, 1);
COMMIT;
-- Session A
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Parent SET pv= pv+1 WHERE pid = 1;
-- Session B
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO Child VALUES (2, 5, 1);
-- Session A
-- At this step it would be useful to have a look at the current locks.
-- In SQL Server this goes like this:
EXEC sp_lock;
ROLLBACK;
-- Session B
ROLLBACK;
```

Exercise 8 Updating Parent row of a Child

The same scenario as above, now in table format:

step	session A	session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	

2	`update Parent set pv= pv+1`
	`where pid = 1`
	
	`(1 row(s) affected)`	

3		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
4		`set transaction isolation level` `read committed`
5		`insert into Child values (2, 5, 1)` Executing query...
6	`-- Let's look at the locks:` `sp_lock -- *) see results 8-6`	
7	`rollback` `Command(s) completed successfully.`	`(1 row(s) affected)`
8		`rollback` `Command(s) completed successfully.`

*) Sp_lock results 8-6

Results | Messages

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	53	5	0	0	DB		S	GRANT
2	53	5	421576540	1	KEY	(8194443284a0)	X	GRANT
3	53	5	421576540	1	PAG	1:174	IX	GRANT
4	53	1	1131151075	0	TAB		IS	GRANT
5	53	5	421576540	0	TAB		IX	GRANT
6	56	5	485576768	1	KEY	(61a06abd401c)	X	GRANT
7	56	5	0	0	DB		S	GRANT
8	56	5	421576540	0	TAB		IS	GRANT
9	56	5	485576768	1	PAG	1:184	IX	GRANT
10	56	5	485576768	0	TAB		IX	GRANT
11	56	5	421576540	1	KEY	(8194443284a0)	S	WAIT
12	56	5	421576540	1	PAG	1:174	IS	GRANT

Based on the sp_lock report 8-6 try to explain how SQL Server operates in this referential integrity tests.

Answer: From the locks we can see that SQL Server has managed to get all locks for inserting the row into Child table, but for checking the referential integrity of the foreign key, it actually wants to read the corresponding Parent row. However, this is prevented by the X-lock of transaction of A until the lock of A is released,

Exercise 9 Deleting Parent row of a Child

Let's continue with a different scenario of the parent-child experiment adding first a new parent and then deleting it. Then we try to add a new child to the parent which just has been deleted.

step	session A	session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2	`insert into Parent values (2, 0)` `(1 row(s) affected)`	
3	`commit`	
4	`delete from Parent where pid =2;` `(1 row(s) affected)`	
5		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`
6		`insert into Child values (2, 1, 2)` Executing query...
7	`sp_lock; -- *) see results 9-7`	

8	`commit`
	`Command(s) completed successfully.`
		`Msg 547, Level 16, State 0, Line 1`
		`The INSERT statement conflicted with the FOREIGN KEY constraint "FK__Child__fk__20C1E124". The conflict occurred in database "TEST", table "dbo.Parent", column 'pid'.`
		`The statement has been terminated.`

9		`rollback`

Based on the sp_lock report 9-7 try to explain how SQL Server operates in this referential integrity tests.

SP_LOCK report 9-7

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	53	5	0	0	DB		S	GRANT
2	53	5	421576540	1	PAG	1:174	IX	GRANT
3	53	1	1131151075	0	TAB		IS	GRANT
4	53	5	421576540	0	TAB		IX	GRANT
5	53	5	421576540	1	KEY	(61a06abd401c)	X	GRANT
6	55	5	421576540	1	KEY	(61a06abd401c)	S	WAIT
7	55	5	0	0	DB		S	GRANT
8	55	5	485576768	1	KEY	(61a06abd401c)	X	GRANT
9	55	5	421576540	0	TAB		IS	GRANT
10	55	5	485576768	1	PAG	1:184	IX	GRANT
11	55	5	485576768	0	TAB		IX	GRANT
12	55	5	421576540	1	PAG	1:174	IS	GRANT

Answer: Transaction of A still has X-lock on the deleted Parent row preventing transaction of B from getting a S-lock on the Parent row for cecking existence of the parent.

Exercise 10 Deleting Parent of a deleted Child

step	session A	session B
1	`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`	
2	`insert into Parent` `values (2, 0)`	
3	`insert into Child` `values (2, 1, 2)`	
4	`commit`	
5	`delete from Child` `where fk = 2`	
5		`-- in SQL Server:` `SET IMPLICIT_TRANSACTIONS ON`

6		`delete from Parent`
		`where pid = 2`
		Executing query...

7	`EXEC sp_lock; -- only for SQL Server`	

8	`rollback`
	`Command(s) completed successfully.`
		`Msg 547, Level 16, State 0, Line 1`
		`The DELETE statement conflicted with the REFERENCE constraint "FK__Child__fk__20C1E124".`
		`The conflict occurred in database "TEST", table "dbo.Child", column 'fk'.`
		`The statement has been terminated.`

9		`rollback`

Based on the sp_lock report 10-7 try to explain how SQL Server operates in this referential integrity tests.

SP_LOCK report of step 10-7

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	53	5	0	0	DB		S	GRANT
2	53	5	485576768	1	PAG	1:184	IX	GRANT
3	53	5	485576768	0	TAB		IX	GRANT
4	53	1	1131151075	0	TAB		IS	GRANT
5	53	5	485576768	1	KEY	(61a06abd401c)	X	GRANT
6	55	5	485576768	1	KEY	(61a06abd401c)	S	WAIT
7	55	5	421576540	1	KEY	(61a06abd401c)	X	GRANT
8	55	5	421576540	0	TAB		IX	GRANT
9	55	5	485576768	0	TAB		IS	GRANT
10	55	5	421576540	1	PAG	1:174	IX	GRANT
11	55	5	485576768	1	PAG	1:184	IS	GRANT
12	55	5	0	0	DB		S	GRANT

Answer: On delete of a parent, SQL Server looks for the existing child rows after all locks for the deletion of the parent row itself have been received.

Metadata Concurrency Control

SQL DDL commands are transactional in SQL Server, and obviously data structure modifications need integrity protection while under construction. Even if it is possible to make online structure changes, it is not a recommended practice in production environment.

Exercise 10
Let's make just a short experiment on what is going on behind the scene, as we make some DDL changes:

In a session – let's call it session A – we create a table and transactionally an index for that table

```
SET NOCOUNT ON
SET IMPLICIT_TRANSACTIONS ON
CREATE TABLE Metatest (
id  INT NOT NULL PRIMARY KEY,
s   INT
) ;
COMMIT ;
```

Exercise 10a – transactionally creating an index

```
CREATE INDEX Meta_Xs ON Metatest(s)
```

Now let's connect to the same database in a new Query window starting concurrent session B and try to insert a row into the table

```
BEGIN TRANSACTION
INSERT INTO Metatest VALUES (1, 2)
```

Executing query...

Returning to session A we can now **paste** (!) the following command into the Query window of A and execute the command to see the current locks

```
EXEC sp_LOCK ;
```

spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
53	5	0	0	DB		S	GRANT
53	5	7	0	TAB		IX	GRANT
53	5	3	0	TAB		IX	GRANT
53	5	5	0	TAB		IX	GRANT
53	5	0	0	DB	[DDL]	S	GRANT
53	5	933578364	2	HBT		Sch-M	GRANT
53	5	34	0	TAB		IX	GRANT
53	5	55	0	TAB		IX	GRANT
53	5	54	0	TAB		IX	GRANT
53	5	60	0	TAB		IX	GRANT
53	5	54	2	HBT		IX	GRANT
53	5	54	1	KEY	(e641fc438b75)	X	GRANT
53	5	54	2	KEY	(6d1f38fbe365)	X	GRANT
53	5	0	0	MD	8(1:0:0)	S	GRANT
53	5	5	1	KEY	(aa0679bda732)	X	GRANT
53	1	1131151075	0	TAB		IS	GRANT
53	5	7	1	KEY	(d9f7ed548807)	X	GRANT
53	5	7	2	KEY	(53f4cc6125e8)	X	GRANT
53	5	3	1	KEY	(bee4daf7499d)	X	GRANT
53	5	0	0	MD	1(37a5467c:2:0)	Sch-M	GRANT
53	5	60	1	KEY	(9b99f6ca546d)	X	GRANT
53	5	3	1	KEY	(5ed0f4788d21)	X	GRANT
53	5	34	1	KEY	(c5b5d403ed69)	X	GRANT
53	5	55	2	KEY	(a759ab45551c)	X	GRANT

```
53      5       55          1       KEY   (45ac49681319)                  X         GRANT
53      5       933578364   0       TAB   [INDEX_OPERATION]               Sch-S     GRANT
53      5       933578364   0       TAB                                   S         GRANT
55      5       933578364   0       TAB                                   Sch-S     GRANT
55      5       0           0       MD    1(37a5467c:2:0)                 Sch-S     WAIT
55      5       0           0       MD    1(37a5467c:1:0)                 Sch-S     GRANT
55      5       869578136   0       TAB                                   IX        GRANT
55      5       0           0       DB                                    S         GRANT
55      5       0           0       DB    [PLANGUIDE]                     S         GRANT
```

and after we commit the CREATE INDEX transaction of session A using

```
COMMIT ;
```

the INSERT command of session B gets the locks it needs to execute the INSERT and we can find out the locks, including the locks on the new index (IndId value 2 below)

```
EXEC sp_lock
spid   dbid   ObjId       IndId  Type Resource                        Mode     Status
------ ------ ----------- ------ ---- --------------------------------- -------- ------
53     5      0           0      DB                                    S        GRANT
55     5      869578136   0      TAB                                   IX       GRANT
55     5      933578364   1      KEY  (8194443284a0)                  X        GRANT
55     5      933578364   2      PAG  1:188                            IX       GRANT
55     5      933578364   1      PAG  1:186                            IX       GRANT
55     1      1131151075  0      TAB                                   IS       GRANT
55     5      933578364   2      KEY  (0971161ce8c0)                  X        GRANT
55     5      933578364   0      TAB                                   IX       GRANT
55     5      0           0      DB                                    S        GRANT
```

Note: As a word of warning, if you try to type "`EXEC sp`" for the EXEC sp_lock command above instead of pasting the command string, the Management Studio gets blocked with the locks of session A as its intellisense service is trying lookup of the possible system procedures.

The following exercises are left as your private challenge:

Exercise 10b – transactionally altering the table

Exercise 10c – transactionally dropping the table