Informatics 2
Laboratory #4.

Transaction Management

1. Goal

Transaction management of database systems is introduced in the Lab. Due to concurrent data processing, several interferences have to be taken into account and these interferences should be handled.

2. Required knowledge

In order to fulfill the exercises, students have to know the basic properties of transactions and problems caused by concurrent data access along with some MS SQL specific transaction management keywords.

2.1 Basic properties of transactions

A database transaction is a unit of interaction with a database management system that must be either entirely completed or aborted. If any of the tasks belonging to the transaction cannot be executed successfully, the whole transaction has to be aborted and made undone.

A transaction may contain several low level tasks and further a transaction is very small unit of any program. A transaction in a database system must maintain some properties in order to ensure the accuracy of its completeness and data integrity. These properties are refer to as ACID properties and are mentioned below:

- **A**tomicity: Though a transaction involves several low level operations but this property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in database where the transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **C**onsistency: This property states that after the transaction is finished, its database must remain in a consistent state. There must not be any possibility that some data is incorrectly affected by the execution of transaction. If the database was in a consistent state before the execution of the transaction, it must remain in consistent state after the execution of the transaction.
- **I**solation: In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all

the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

- **D**urability: This property states that in any case all updates made on the database will persist even if the system fails and restarts. If a transaction writes or updates some data in database and commits that data will always be there in the database. If the transaction commits but data is not written on the disk and the system fails, that data will be updated once the system comes up.

It can be derived from basic properties of transactions that database servers have to provide border markers for the transactions (assign a beginning and an end to them) because this is the only way of grouping tasks into a single unit. The end of transactions is marked by *commit* if the transaction was successful, otherwise by *rollback*.

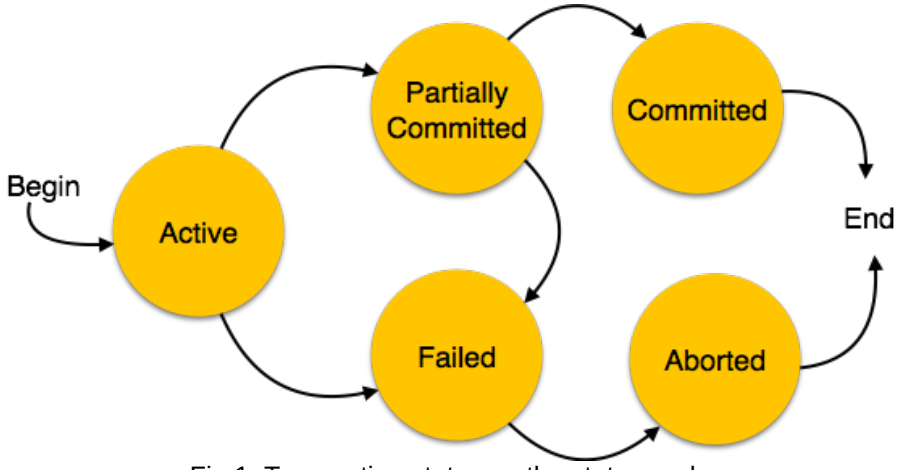

Fig.1. Transaction states on the state graph.

A transaction in a database can be in one of the following state:

- **Active**: In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed**: When a transaction executes its final operation, it is said to be in this state. After execution of all operations, the database system performs some checks e.g. the consistency state of database after applying output of transaction onto the database.
- **Failed**: If any checks made by database recovery system fails, the transaction is said to be in failed state, from where it can no longer proceed further.
- **Aborted**: If any of checks fails and transaction reached in Failed state, the recovery manager rolls back all its write operation on the database to make database in the state where it was prior to start of execution of transaction. Transactions in this state are called aborted. Database recovery module can select one of the two operations after a transaction aborts:
  - Re-start the transaction
  - Kill the transaction
- **Committed**: If transaction executes all its operations successfully it is said to be committed. All its effects are now permanently made on database system.

Isolation methods of transactions are detailed in the following sections. **Transaction logs** are used to restore consistency (to take database systems back to consistent state) after serious system failures. Transaction logs contain the state of any modified record before and after modification along with the transaction limit markers, so the database can always be restored to a consistent state. (Naturally this is only the basic concept of transaction logs, details of transaction log usage is out of the scope of this document.)

2.2 Problems of concurrent transactions

Since several users can use the same database at the same time (concurrent access), they can disturb each other. Concurrent transactions can interfere with each other if they use the same database objects (e.g. they use the same records). Concurrent el data access can cause the following problems:

- **Dirty Read**
  Dirty Read occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row.

  For example, an editor is making changes to an electronic document. During the changes, a second editor takes a copy of the document that includes all the changes made so far, and distributes the document to the intended audience. The first editor then decides the changes made so far are wrong and removes the edits and saves the document. The distributed document contains edits that no longer exist, and should be treated as if they never existed. This problem could be avoided if no one could read the changed document until the first editor determined that the changes were final.

- **Lost Update**
  Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

  For example, two editors make an electronic copy of the same document. Each editor changes the copy independently and then saves the changed copy, thereby overwriting the original document. The editor who saves the changed copy last overwrites changes made by the first editor. This problem could be avoided if the second editor could not make changes until the first editor had finished.

- **Non-repeatable Read**
  Non-repeatable read occurs when a second transaction accesses the same row several times and reads different data each time. It is similar to Dirty Read in that another transaction is changing the data that a second transaction is reading. However, in Non-repeatable Read, the data read by the second transaction was committed by the transaction that made the change. Also, Non-repeatable Read involves multiple reads (two or more) of the same row and each time the information is changed by another transaction; thus, the term Non-repeatable Read.

For example, an editor reads the same document twice, but between each reading, the writer rewrites the document. When the editor reads the document for the second time, it has changed. The original read was not repeatable. This problem could be avoided if the editor could read the document only after the writer has finished writing it.

- **Phantom records (Phantom read)**
  Phantom reads occur when an insert or delete action is performed against a row that belongs to a range of rows being read by a transaction. The transaction's first read of the range of rows shows a row that no longer exists in the second or succeeding read, as a result of a deletion by a different transaction. Similarly, as the result of an insert by a different transaction, the transaction's second or succeeding read shows a row that did not exist in the original read.

  For example, an editor makes changes to a document submitted by a writer, but when the changes are incorporated into the master copy of the document by the production department, they find that new unedited material has been added to the document by the author. This problem could be avoided if no one could add new material to the document until the editor and production department finish working with the original document.

For more details see https://technet.microsoft.com/en-us/library/cc546518.aspx.

2.3 Isolation Levels

The following isolation levels are available to solve problems described in section 2.2 according to SQL specification:

- **Read uncommitted**: transactions can read uncommitted records
- **Read committed:** transactions can only read committed records
- **Repeatable read**: records read by a transaction cannot be changed by other transactions which means that committed records of other transactions cannot be written till the end of such reader transaction.
- **Serializable**: transaction timing is equivalent to the serial timing, which means that transactions behave as if they were executed one after the other.

Each transaction can use a different isolation level if required. Database management systems (like MS SQL Server, Oracle Server) use *Read committed* isolation level by default. This isolation level is used in most cases and it is not changed during system development process.

Different isolation levels can or cannot eliminate some different transaction problems. The following table summarizes problems solved by different isolation levels:

	Lost update	**Dirty read**	**Non-repeatable read**	**Phantom records**
Read uncommitted	solved	no	no	no
Read committed	solved	solved	no	no
Repeatable read	solved	solved	solved	no
Serializable	solved	solved	solved	solved

2.4 Locks

Isolation levels are implemented by database systems with the help of locking mechanisms. Only one model, the simple two-phase locking model is described here. There are two types of locks used in simple locking model:

- **shared** or **read lock**: it can belong to multiple transactions
- **exclusive** or **write lock**: it can belong to only one transaction

Locking system has the following priority: no further locks can be applied to an exclusive lock.

DML operations try to apply locks on resources (e.g. on records or tables); the type of lock to be used depends on the isolation level. If the lock cannot be applied, then one should wait until the required resource becomes available. Transactions hold their locks until their execution is finished, releasing locks earlier is not allowed.

Here we described the simple two-phase locking model. However, database management systems use several locking models to support all isolation levels.

2.5 Deadlock

Deadlock is a situation when two or more concurrent transactions are waiting for each other; that means that none of them can continue running. Waiting for another transaction means waiting for a resource used by the other one.

Even though deadlocks can be prevented, database management systems usually just detect the occurred deadlock since its prediction and prevention has a very high cost. On the other hand, deadlocks are not too likely to occur.

Deadlocks can be detected by using resource-allocation graphs (RAG). A resource allocation graph tracks which resource is held by which transaction and which transaction is waiting for a resource of a particular type. The nodes of the graph are resources and transactions. Meaning of edges is the following:

- resource→transaction: resource belongs to the transaction
- transaction→resource: transaction waits for the resource

Deadlock can occur only if there is a directed cycle in the RAG. Database management systems can detect these cycles and resolve them. This is done by forcing one of the transactions to finish its execution and rollback is performed for this transaction, thus the other transaction(s) can continue their work.

2.6 MS SQL Server specific transaction management

2.6.1 Transaction boundaries

A transaction boundary defines the scope of a transaction. Objects inside a transaction boundary share a common transaction identifier.

MS SQL Server uses *auto commit* behavior by default, which means that each DML operation belongs to a different transaction. Each transaction finishes its execution by commit if no errors occur.

If multiple operations are to be grouped into a single transaction, first **begin** command has to be sent to the database server. The end of the transaction must be explicitly marked by either a **commit** or **rollback** command. *Auto commit* will be used if this transaction is finished, which means that if another grouping is necessary, another **begin** command has to be used.

2.6.2 Selecting isolation levels

Read committed is the default isolation level of MS SQL Server. Whenever isolation level has to be changed, **set transaction isolation level #level#** command must be used. Isolation level will stay unchanged until the end of the database connection (or until a new **set transaction isolation level** command).

2.6.3 Exclusive locking

Data manipulation operations (update, insert etc.) apply an excusive lock on the record to be used, but we can also use exclusive locks. It can happen whenever concurrent transactions have to be synchronized (to implement mutual exclusion). Records can be exclusively locked using **select #columns# from #table_name# with(XLOCK)** command.

3 Sample Review Questions

1. What does atomicity of transactions mean?
2. What does consistency of transactions mean?
3. What does durability of transactions mean?
4. What does isolation of transactions mean?
5. What transaction states do you know? Explain each state and draw the state graph.
6. What is dirty read problem?
7. What is lost update problem?
8. What is non-repeatable read problem?
9. What is phantom records problem?
10. What is isolation levels exist in SQL Server? What problems does each of them solve?
11. What is a deadlock? How can it be detected?
12. What is the resource-allocation graph (RAG) used for in DBMS?
13. … other similar questions may be asked.

Exercices during the Laboratory

1. Get connected to the SQL Server and create a bank account database using the following SQL script.

```
create table account
(id int primary key,
 balance int)
insert into account values(1, 5000)
insert into account values(2, 8500)
insert into account values(4, 6400)
```

2. Transfer 1000 HUF from bank account 1 to 2. In order to check the transfer open another query window, this creates a second parallel connection to the database. Check balance during transfer continuously. What did you notice and why?

	Transfer window	Check window
1.		`select * from account`
2.	`update account set balance=balance-1000 where id=1`	
3.		`select * from account`
4.	`update account set balance=balance+1000 where id=2`	
5.		`select * from account`

3. Change the above script in a way that the data manipulation operations are put into the same transaction. Replay the second task. What did you notice and why?

4. Deadlock creation. There are two transfers running in parallel: one of them sends 500 HUF from account 1 to 2 while the second one sends 300 HUF from bank account 2 to 1. Let the transactions schedule be the following:

	First transfer	Second transfer
1.	`begin transaction`	
2.		`begin transaction`
3.	`update account set balance=balance-500 where id=1`	
4.		`update account set balance=balance-300 where id=2`
5.	`update account set balance=balance+500 where id=2`	
6.		`update account set balance=balance+300 where id=1`

What is the result? Why?

5. Create a simple exam signup system using the following SQL script.

```
drop table signup
drop table exam

create table exam
( id int primary key,
  subject varchar(20),
  date datetime,
  limit int
)
create table signup(
  examid int references exam(id),
  studentid int,
  primary key (examid,studentid)
)
insert into exam
values(1, 'Informatics2',convert(datetime,'2007.06.15',102),3)
insert into exam
values(2, 'Mathematics',convert(datetime,'2007.06.18',102),3)
insert into signup values(1,111)
insert into signup values(1,222)
```

6. Simulate two concurrent signups to the first exam. Schedule of the processes should be the following:

	First signup	Second signup

1.	`select limit`
	`from exam`
	`where id=1`	

2.		`select limit`
		`from exam`
		`where id=1`

3.	`select count(*)`
	`from signup`
	`where examid=1`	

4.		`select count(*)`
		`from signup`
		`where examid=1`

5.	`insert into signup` `values(1, 333)`	
6.		`insert into signup` `values(1, 444)`

What is the result? Why?

7. If transactions are isolated properly, the problem described in exercise 6 can be avoided. Increase isolation levels of transactions to *'serializable'* and replay the previous task as follows.

	First signup	Second signup

1.	`set transaction isolation level serializable`
	
	`Begin transaction`	

2.		`set transaction isolation level serializable`
		
		`Begin transaction`

3.	`select limit from exam where id=1`	
4.		`select limit from exam where id=1`
5.	`select count(*) from signup where examid=1`	
6.		`select count(*) from signup where examid=1`
7.	`insert into signup values(1,555)`	
8.		`insert into signup values(1,666)`

What is the result? Why?

8. Increase limit of the first exam to 5. Replay example 7. What is the result? Why?

9. Replay example 7 but the first student should signup to the first exam while the second one should signup to the second exam. What is the result? Why?

10. Reset our database by executing the script of example 5 again.

11. The efficiency of the system can be increased by using read committed isolation level and mutual locking. If concurrent transactions lock only the record they need, mutual exclusion can be achieved. The schedule of the processes should be the following:

	First signup	Second signup

1.	`set transaction isolation level read committed`
	
	`begin transaction`	

2.		`set transaction isolation level read committed`
		
		`begin transaction`

3.	`select limit from exam with(XLOCK) where id=1`	
4.		`select limit from exam with(XLOCK) where id=1`
5.	`select count(*) from signup where examid=1`	
6.	`insert into signup values(1,333)`	
7.	`commit`	
8.		`select count(*) from signup where examid=1`
9.		`commit`

What is the result? Why?

12. Increase limit of the first exam to 6. Replay example 11. What is the result? Why?

13. Replay example 12 but the first student should signup to the first exam while the second one should signup to the second exam. What is the result? Why?