Exercise: Transactions / Concurrency Control

1. Serializability

Consider the following schedule of two transactions A and B:

$A_{read\ (y)}$ $B_{read\ (x)}$ $A_{read\ (z)}$ $A_{read\ (x)}$ $B_{write\ (x)}$ $B_{read\ (y)}$ $A_{read\ (x)}$ $A_{write\ (y)}$ $B_{write\ (y)}$ $A_{write\ (z)}$

a) Name the types of problems that occur and mention which data items and transactions they concern.

b) Rearrange the operations to form a serializable, but not serial schedule of the transactions A and B.
Are both options possible: to form a schedule with the effect of "A -> B" and of "B -> A"?

2. Isolation Levels, Problem 1

Consider two parallel transactions A and B in MySQL with the innodb storage engine[1].
Both transactions use the isolation level "serializable".
Consider the tables "Professor" and "TA" from our class example (see appendix of the exam paper).
The following table shows the first steps of the transactions in chronological order from top to bottom. Assume that the steps follow each other quickly enough to prevent timeouts.

Step	Transaction A (serializable)	Transaction B (serializable)
1	**Start transaction A**	
2		**Start transaction B**
3		**select salary from Professor where pName = 'Langes';**
4		**update Professor set salary = salary + (select taSalary from TA where matNr = 2345 and classNr = 'DTB-SS93') where pName = 'Langes';**
5	**select taSalary from TA where matNr = 2345 and classNr = 'DTB-SS93';**	

1. The innodb ensures that transaction management like locking works correctly. It is the default storage engine in newer MySQL versions.

Step	Transaction A (serializable)	Transaction B (serializable)
6	**update TA set taSalary = 2222 where matNr = 2345 and classNr = 'DTB-SS93';**	
7		**select taSalary from TA where matNr = 2345 and classNr = 'DTB-SS93';**
8	**select * from Professor;**	


In the following you find a number of statements, some of which are correct and some are incorrect. Please find out for every statement whether it is correct or wrong and why.

1. Transaction A is blocked at step 6, while B may proceed with step 7 and on.
2. Transaction A reads the table Professor in step 8 and sees the new value for the tuple with Langes.
3. Transaction B is blocked at step 7, while transaction A may proceed with step 8 and on.
4. At step 8, transactions A and B are in a deadlock.
5. Transaction A is blocked already in step 5, because it accesses a table on which transaction B is already working.
6. Transaction B contains a non-repeatable read.
7. Transaction B reads the value 2000 in step 7.
8. Transaction A will have a consistent outcome, while transaction B may have an inconsistency.
9. A may proceed after B has committed or aborted (if that happens fast enough).
10. A may be aborted at step 6 because of a timeout if B takes too long to commit or rollback after step 7.

2. Isolation Levels, Problem 2

Work in pairs.
Both of you log in to the same database (your project database or the university database) in two different, parallel sessions.
You can also create the effects on your own when you open two or more shell windows, connect to the database in both of them and execute transactions in parallel within both connections.

Switch off the autocommit. (`Set autocommit = 0;`).
Your tables are transaction safe if they were created with the storage engine innodb.
(Note: You can check whether a table was created with innodb by entering
`show create table bla;`
where bla is the name of the table you are interested in).

Start two parallel transactions that make changes in the database.
You may want to have an additional session open to monitor which transactions are currently known to the system. You can get information about the current transactions with the command:

```
mysql> select * from information_schema.innodb_trx\G
```

Experiment with different isolation levels and try to produce
- dirty reads
- lost updates
- unrepeatable reads
- phantom records
- serializable, interleaved transactions
- a deadlock

Measure the time-out period in MySQL after which a transaction is aborted because the system assumes that it may be in deadlock.

Here are some hints that may be of help:

Transaction syntax:

A transaction is of the form
start transaction
<SQL statements>
commit;
or
rollback;
(If autocommit is switched off, the start transaction clause is actually not necessary).

Checking isolation levels:
show session variables like 'tx_isolation';
(=> display the isolation level that is set for the current session)
show global variables like 'tx_isolation';
(=> display the isolation level that is globally set for future sessions)
select trx_isolation_level from information_schema.innodb_trx;
(=> display the isolation levels of the currently active transactions).

Example for producing a deadlock

From
http://www.xaprb.com/blog/2006/08/08/how-to-deliberately-cause-a-deadlock-in-mysql/

„First, choose an unused table name. I'll use test.innodb_deadlock_maker. Here are the statements you need to execute:

create table test.innodb_deadlock_maker(a int primary key) engine=innodb;
insert into test.innodb_deadlock_maker(a) values(0), (1);

Now the table and its data are set up. Next, execute the following on two different con-

nections:

```
-- connection 0
set transaction isolation level serializable;
start transaction;
select * from test.innodb_deadlock_maker where a = 0;
update test.innodb_deadlock_maker set a = 0 where a <> 0;

-- connection 1
set transaction isolation level serializable;
start transaction;
select * from test.innodb_deadlock_maker where a = 1;
update test.innodb_deadlock_maker set a = 1 where a <> 1;
```

Voilà, you have a deadlock. Notice how connection 0 and connection 1 run the same statements, except they use a different value in their WHERE and SET clauses. This makes it easy to write a program to run these statements, and just pass in a value each connection should use.”

Exercise: Transactions / Concurrency Control - Solutions

1. Serializability

Consider the following schedule of two transactions A and B:

$A_{read\ (y)}$ $B_{read\ (x)}$ $A_{read\ (z)}$ $A_{read\ (x)}$ $B_{write\ (x)}$ $B_{read\ (y)}$ $A_{read\ (x)}$ $A_{write\ (y)}$ $B_{write\ (y)}$ $A_{write\ (z)}$

a) Name the types of problems that occur and mention which data items and transactions they concern.

There is a lost update of y by A and an unrepeatable read of x by A.

Note:
If you execute this scenario in MySQL, you will notice that an unrepeatable read can only be observed in MySQL after the transaction that has changed the value in question has committed. Before the commit of that transaction, the other transaction can read as often as it likes, it will always see the value in the form that it was read at first. This holds even if the other transaction has deleted the value.

b) Rearrange the operations to form a serializable, but not serial schedule of the transactions A and B.

Example:
$A_{read\ (y)}$ $B_{read\ (x)}$ $A_{read\ (z)}$ $A_{read\ (x)}$ $A_{read\ (x)}$ $B_{write\ (x)}$ $A_{write\ (y)}$ $B_{read\ (y)}$ $B_{write\ (y)}$ $A_{write\ (z)}$

This schedule is not serial and has the same effect as the serial execution "A -> B".

It is not possible to find a serializable and not serial rearrangement that has the same effect as "B -> A".
Reason:
A and B both write y. If by effect, B is supposed to come first, this means that $A_{read\ (y)}$ can only be done after $B_{write\ (y)}$.
Since $B_{write\ (y)}$ is the last operation of B, A could only start after B has finished. Naturally, "B -> A" would also be a conflict free, serializable rearrangement of the given schedule. It just would be serial, i.e., without interleaving.

2. Isolation Levels, Problem 1

The following answers are correct:
1, 7, 9, 10

2. Isolation Levels, Problem 2

There is no sample solution. Create your own scenarios, letting two transactions run against each other.