**www.DBTechNet.org**
DBTech VET


# SQL Transaction Exercises and Answers using MySQL 5.6

Based on exercises the Appendix1_MySQL.txt file version 2013-09-10.
Term "booklet" below refers to the book/PDF "SQL Transactions" produced in DBTech VET project.

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.

In the following we present the transaction exercises with some extra comments, and answers to the questions. For some of the queries we present on blue background also the sample output results.

```
-- ============================================================*/
-- Part 1   Experimenting with single transactions
--           - "Logical Units of Work"
-- ------------------------------------------------------------

mysql
USE testdb; -- connect your SQL session to access testdb
HELP;       -- will display all commands of the MySQL client program
--
-- As default MySQL session starts in Autocommit mode
-- which you can verify as follows:
SELECT @@autocommit;
```

```
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
```

This means that autocommit mode for the SQL-session is ON. Zero means OFF.

```
-- Exercise 1.1

CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), si SMALLINT)
ENGINE=InnoDB;
```

MySQL comes with multiple storage engines, and by the optional ENGINE clause of CREATE TABLE command it is possible to declare which engine will be used for the table. The ENGINE clause in our example is not necessary, since the default engine in 5.6 is InnoDB.

Like in Oracle, the structure of an existing table can be reported by DESCRIBE <table> command as follows:

```
DESCRIBE T;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | int(11)     | NO   | PRI | NULL    |       |
| s     | varchar(40) | YES  |     | NULL    |       |
| si    | smallint(6) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

```
3 rows in set (0.00 sec)
```

Note: The Extra column would display additional attributes of special columns to be reported, such as AUTO_INCREMENT or automatic update timestamping.

```
Now, let's add some content and experiment with that data
INSERT INTO T (id, s) VALUES (1, 'first');
INSERT INTO T (id, s) VALUES (2, 'second');
INSERT INTO T (id, s) VALUES (3, 'third');
SELECT * FROM T ;
ROLLBACK;
SELECT * FROM T ;
mysql> SELECT * FROM T ;
+----+--------+------+
| id | s      | si   |
+----+--------+------+
|  1 | first  | NULL |
|  2 | second | NULL |
|  3 | third  | NULL |
+----+--------+------+
3 rows in set (0.00 sec)

START TRANSACTION;
INSERT INTO T (id, s) VALUES (4, 'fourth');
SELECT * FROM T ;
ROLLBACK;
SELECT * FROM T;
mysql> SELECT * FROM T ;
+----+--------+------+
| id | s      | si   |
+----+--------+------+
|  1 | first  | NULL |
|  2 | second | NULL |
|  3 | third  | NULL |
+----+--------+------+
3 rows in set (0.00 sec)

-- ------------------------------------------------------------
-- Question:
-- Compare the results obtained by executing the above command sequences.
-- What have we verified about AUTOCOMMIT, transactions and ROLLBACK?
```

Answer:
In autocommit mode every command will be committed automatically. ROLLBACK command will not have any effect, and it will not generate error exception or warning..

```
-- ------------------------------------------------------------

-- Exercise 1.2
INSERT INTO T (id, s) VALUES (5, 'fifth');
ROLLBACK;

SELECT * FROM T;
+----+-------+------+
| id | s     | si   |
+----+-------+------+
|  1 | first | NULL |
|  5 | fifth | NULL |
+----+-------+------+
2 rows in set (0.00 sec)

-- ------------------------------------------------------------
-- Questions:
-- What is the result set obtained by executing the above SELECT * FROM T command?
-- Conclusion(s) reached with regard to the existence of possible limitations
-- in the use of the START TRANSACTION command in MySQL/InnoDB?
```

Answer:
In autocommit mode a START TRANSACTION command will start an explicit transaction, but after the transaction ends, the session will return to autocommit mode.

```
-- ------------------------------------------------------------
```

```
-- Exercise 1.3
-- Turning now transactional mode on
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
COMMIT;
INSERT INTO T (id, s) VALUES (6, 'sixth');
INSERT INTO T (id, s) VALUES (7, 'seventh');
SELECT * FROM T;
ROLLBACK;
SELECT * FROM T;
mysql> SELECT * FROM T;
+----+-------+------+
| id | s     | si   |
+----+-------+------+
|  1 | first | NULL |
+----+-------+------+
1 row in set (0.00 sec)

-- ------------------------------------------------------
-- Question:
-- What is the advantage/disadvantage of using the "START TRANSACTION" statement,
-- as compared to using the "SET AUTOCOMMIT=0" one, in order to switch off MySQL's
-- (default) AUTOCOMMIT mode?
```

Answer:
START TRANSACTION documents explicitly start of a new transaction and it also allows setting extra transaction attribute, modifiers, such as “READ ONLY” or “WITH CONSISTENT SNAPSHOT” as explained at http://dev.mysql.com/doc/refman/5.6/en/commit.html .
A disadvantage is that at any end of the explicit transaction the session returns to autocommit mode. However, START TRANSACTION with modifiers can be used in transactional mode, in which case the session will stay in transactional mode at the end of the transaction.

```
-- ------------------------------------------------------

-- Exercise 1.4
-- Initializing only in case you want to repeat the exercise 1.4
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
DROP TABLE T2;  --
COMMIT;

-- DDL stands for Data Definition Language.  In SQL the statements like
-- CREATE, ALTER and DROP are called DDL statements.
-- Now let's test use of DDL commands in a MySQL/InnoDB transaction!

SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (2, 'will this be committed?');
CREATE TABLE T2 (id INT) ENGINE=InnoDB;
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;

SELECT * FROM T;  -- What has happened to T ?
SELECT * FROM T2; -- What has happened to T2 ?
mysql> SELECT * FROM T2;
Empty set (0.00 sec)
-- Compare this with SELECT from a missing table as follows:
SELECT * FROM T3; -- assuming that we have not created table T3
mysql> SELECT * FROM T3;
ERROR 1146 (42S02): Table 'testdb.T3' doesn't exist

SHOW TABLES;
DROP TABLE T2;
COMMIT;
-- ------------------------------------------------------
-- Question:
-- Conclusions reached?
```

Answer:
SELECT from empty table generates empty resultset, whereas SELECT from a missing table generates an error exception.

```
-- ------------------------------------------------------
```

```
-- Exercise 1.5
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
COMMIT;
SELECT * FROM T;
COMMIT;

-- ---------------------------------------------------------------
-- Testing if an error would lead to automatic rollback in MySQL?
-- ---------------------------------------------------------------

SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (2, 'The test starts by this');
-- division by zero should fail
SELECT (1/0) AS dummy ;
SHOW ERRORS;
SHOW WARNINGS;
-- Oops, see what we just found out!
-- Now updating an non-existing row
UPDATE T SET s = 'foo' WHERE id = 9999 ;
-- and deleting an non-existing row
DELETE FROM T WHERE id = 7777 ;
--
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
INSERT INTO T (id, s) VALUES (3, 'How about inserting too long string value?');
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
SHOW ERRORS;
SHOW WARNINGS;
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
DELETE FROM T WHERE id > 1;
SELECT * FROM T;
COMMIT;
```

```
mysql> SET AUTOCOMMIT=0;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO T (id, s) VALUES (2, 'The test starts by this');
Query OK, 1 row affected (0.00 sec)

mysql> -- division by zero should fail
mysql> SELECT (1/0) AS dummy ;
+-------+
| dummy |
+-------+
|  NULL |
+-------+
1 row in set (0.01 sec)

mysql> SHOW ERRORS;
Empty set (0.00 sec)

mysql> SHOW WARNINGS;
Empty set (0.00 sec)
```

This is interesting: On division by zero MySQL generates a NULL value, whereas other products will generate an error exception.

```
mysql> -- Now updating an non-existing row
mysql> UPDATE T SET s = 'foo' WHERE id = 9999 ;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> -- and deleting an non-existing row
mysql> DELETE FROM T WHERE id = 7777 ;
Query OK, 0 rows affected (0.00 sec)
```

Note: UPDATE or DELETE of non-matching rows is not an error in SQL language, whereas for the application these might be errors. So the application code needs to check the **diagnostics**!

```
mysql> INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'
mysql> INSERT INTO T (id, s) VALUES (3, 'How about inserting too long string value?');
ERROR 1406 (22001): Data too long for column 's' at row 1
mysql> INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
```

```
ERROR 1264 (22003): Out of range value for column 'si' at row 1
mysql> SHOW ERRORS;
+-------+------+---------------------------------------------+
| Level | Code | Message                                     |
+-------+------+---------------------------------------------+
| Error | 1264 | Out of range value for column 'si' at row 1 |
+-------+------+---------------------------------------------+
1 row in set (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+---------------------------------------------+
| Level | Code | Message                                     |
+-------+------+---------------------------------------------+
| Error | 1264 | Out of range value for column 'si' at row 1 |
+-------+------+---------------------------------------------+
1 row in set (0.00 sec)

mysql> INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM T;
+----+-----------------------------------+------+
| id | s                                 | si   |
+----+-----------------------------------+------+
|  1 | first                             | NULL |
|  2 | The test starts by this           | NULL |
|  5 | Is the transaction still active? | NULL |
+----+-----------------------------------+------+
3 rows in set (0.00 sec)

-- ------------------------------------------------------------
-- Questions:
-- a) What have we found out of automatic rollback on SQL errors in MySQL?
      - SQL errors don't seem to generate automatic rollback in MySQL.
-- b) Is division by zero an error?
      - Not in MySQL!
-- c) Does MySQL react on overflows?
      - Overflows will generate error exceptions and will abort the current command!

-- ================================================================
-- A1.2   Experimenting with Transaction Logic
-- ----------------------------------------------------------------

-- Exercise 1.6: COMMIT and ROLLBACK
-- ----------------------------------------------------------------

DROP TABLE Accounts;
SET AUTOCOMMIT=0;

-- Note:
--   We have found that in MySQL/InnoDB the CHECK constraint is
--   available only on row-level, not as column constraint !
--   Even so, it only passes the syntax checking, but does not work!
--   To keep the experiments comparable with other products,
--   we have not removed the CHECK constraint. Products do have bugs.
--   Workaround for this problem is presented in AdvTopics_MySQL.txt
--

CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT unloanable_account CHECK (balance >= 0)
) ENGINE=InnoDB;

-- so this should fail:
INSERT INTO Accounts (acctID,balance) VALUES (100,-1000);

mysql> INSERT INTO Accounts (acctID,balance) VALUES (100,-1000);
Query OK, 1 row affected (0.00 sec)
```

Note: It is a well-known problem that MySQL does not support CHECK constraints.
As workaround CHECKs can be implemented using triggers. See AdvTopics_MySQL.txt

```
SELECT * FROM Accounts;
ROLLBACK;
```

```
-- Let's now load proper contents for our test:
SET AUTOCOMMIT=0;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

-- A. Let's try the bank transfer
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
ROLLBACK;

-- B. Let's test if the CHECK constraint actually works:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts ;
ROLLBACK;
-- So, how it looks?
```

See the trigger solution in AdvTopics_MySQL.txt

```
-- C. Updating a non-existent bank account 777:
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts ;
ROLLBACK;

-- ------------------------------------------------------
-- Questions:
-- a) Do the two UPDATE commands execute despite the fact that the second request
--    for updating a non-existent account/row in the Accounts table?
```

Yes, this not a problem in SQL and DBMS products, but it sure is problem/error for the application.
For this the application needs the **diagnostics** information!

```
-- b) Had the ROLLBACK command in B or C been replaced by a COMMIT one, would the
--    transaction have run with success, having made permanent its effect to the database?
```

See the answer to a). Without more advanced transaction logic the database would contain **incorrect data**!

```
-- c) Which diagnostic indicators of MySQL the user application could use
--    to detect the problems in the above transactions ?
We see that it would be important to read the RETURNED_SQLSTATE after every SQL command, and
ROW_COUNT after every INSERT, UPDATE or DELETE command so that application knows if the
command succeeded in terms of the application/transaction logic.

-- ------------------------------------------------------------------
-- Transaction logic
-- ------------------------------------------------------------------

--  Transaction logic may depend on the diagnostics returned from the requests by
--  database server.  Mysql client displays some diagnostics after every command,
--  but in MySQL's SQL dialect the diagnostics are available first in version 5.6
--  by the new GET DIAGNOSTICS statements of SQL standard, for example as follows:

INSERT INTO T (id, s) VALUES (2, NULL);
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE, @sqlcode = MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;

--  The diagnostic indicator values accessed in local (@)variables can be used by
--  transaction logic in SQL control structures, such as IF .. END IF, etc, but
--  SQL dialect of MySQL these are available only in stored routines.  For examples
--  of these, please see the BankTransfer procedure at the end of this file,
--  and in the file AdvTopics_MySQL.txt

-- ------------------------------------------------------------------
-- Exercise 1.7  Testing the database recovery
-- ------------------------------------------------------------------

SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (9, 'Let''s see what happens if ..');
SELECT * FROM T;
-- Now we will break the client  by control-C

-- ------------------------------------------------
```

```
-- The following is part of a recorded session:
mysql> -- Now we will break the client  by control-C
mysql> ^CCtrl-C -- exit!
Aborted
student@debianDB:~$ ^C
student@debianDB:~$


#-----------------------------------------------------------------
#-- Starting a new terminal window and connecting to our testdb
#-- we can study what happened to our latest uncommitted transaction
#-- just by listing the contents of table T

mysql
USE testdb;
SET AUTOCOMMIT=0;
SELECT * FROM T;
-- Do we see the row of id 9 ?  What does this mean?
COMMIT;
EXIT;  -- closing the mysql client

-- ------------------------------------------------------
-- Question:
-- Does this fit with the explanation presented in Appendix 3?
```

The connection crash is not exactly the same as crashing of the server. However, the uncommitted transactions will be rolled back also at the end of SQL-session.

To experiment with the database recovery in server crash, as root user you could kill the MySQL server process as follows

```
su root
ps -e | grep mysqld
 1088 ?        00:00:00 mysqld debiandb
 1091 ?        00:00:00 mysqld_safe
 1461 ?        00:00:04 mysqld
kill 1461
```

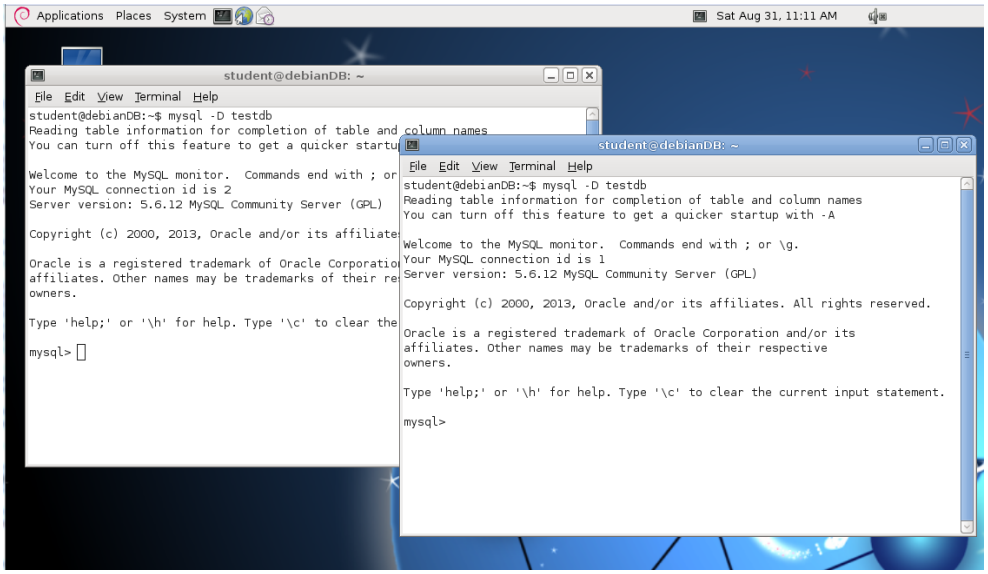
but then to continue exercises you need to reboot your database laboratory, or to know how to restart the mysqld service in DebianDB, otherwise the clients will get following error messages:

```
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
ERROR 2002 (HY000): Can't connect to local MySQL server through socket
'/var/run/mysqld/mysqld.sock' (2)
ERROR:
Can't connect to the server

-- ================================================================
-- Part 2   Experimenting with isolation levels
-- ================================================================

-- Exercises 2.0  (not yet in version 1 of the booklet)
-- Automatic concurrency management services are available
-- by setting proper TRANSACTION ISOLATION LEVEL.  As a best practice
-- the proper isolation level for the transaction should be set
-- in the beginning of the transaction, and it should not be
-- changed between the actions of the transaction.  Let's test
-- how it can be set for explicit transactions in MySQL:

mysql testdb
-- Let's verify what is the default isolation level
SELECT @@GLOBAL.tx_isolation, @@tx_isolation;
mysql> SELECT @@GLOBAL.tx_isolation, @@tx_isolation;
+-----------------------+-----------------+
| @@GLOBAL.tx_isolation | @@tx_isolation  |
+-----------------------+-----------------+
| REPEATABLE-READ       | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.04 sec)

-- Initializing the Accounts table
SET AUTOCOMMIT = 0;
 DROP TABLE Accounts;
 CREATE TABLE Accounts (
 acctID  INTEGER NOT NULL PRIMARY KEY,
```

```
 balance INTEGER NOT NULL,
 CONSTRAINT unloanable_account CHECK (balance >= 0)
 ) ENGINE=InnoDB;
 INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
 INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
 SELECT * FROM Accounts;
 COMMIT;

-- Experimenting with SET TRANSACTION for an explicit transaction

START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;

-- Another try in different order:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;

-- According to ISO SQL standard it is not possible to
-- apply write actions in READ UNCOMMITTED isolation level
-- so let's verify the behavior of MySQL in this case:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
DELETE FROM Accounts;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;

-- ------------------------------------------------------
-- Question:
-- Conclusions reached?
```

So in MySQL the isolation level need to be set **before** the transaction starts, and according to the standard it cannot be changed during the transaction, whereas some other products may allow this. MySQL/InnoDB like most DBMS products don't prevent write actions while isolation level is set to READ UNCOMMITTED, opposite to what is said in ISO SQL standard and textbooks.

```
-- ================================================================
-- Experimenting with Concurrent Transactions
-- ================================================================
-- In the following tests we will use implicit transactions.

-- For concurrency experiments we need to open two parallel
-- mysql client sessions in different terminal windows.
```




```
-- ----------------------------------------------------------------
-- Experiments simulating the "Lost Update Problem"
-- ----------------------------------------------------------------

-- Exercise 2.1
--    Note:
--    Lost Update Problem means that an update made by a transaction
--    is overwritten by some concurrent transaction BEFORE the
--    transaction ends. Every modern DBMS product with concurrency
--    control services prevents this, so the problem is impossible to
--    produce in tests.
--    However, AFTER commit of the transaction any CARELESS concurrent
--    transaction may overwrite the results. Some call also it as
--    "Lost Update Problem", but we call the case "Blind Overwriting"
--    or "Dirty Write", and it is easy to produce.
--    See Table 2.4 in the booklet.  In the following we simulate
--    the application code part by using local (@)variables.

-- 0. Initializing the contents
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

-- 1. client A starts
USE testdb;
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET @amountA = 200; -- amount to be transfered by A
SET @balanceA = 0;  -- init value
SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101;
SET @balanceA = @balanceA - @amountA;
SELECT @balanceA;

mysql> SELECT @balanceA;
+------------+
| @balanceA |
+------------+
|       800 |
+------------+
```

```
1 row in set (0.00 sec)

-- In a new terminal window:
-- 2. client B starts
mysql
USE testdb;
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET @amountB = 500; -- amount to be transfered by B
SET @balanceB = 0;  -- init value
SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101;
SET @balanceB = @balanceB - @amountB;

-- 3. client A continues
UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 4. client B continues after A,
UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;
-- Please note that the default lock timeout in MySQL/InnoDB is 90 seconds!

-- 5. client A should continue without waiting for step 4
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;

-- 6. client B continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;

-- ------------------------------------------------------------
-- Questions:
-- Has the system behaved the way it was expected to?
-- Is there evidence of the lost data in this case?

-- ------------------------------------------------------------

-- Note: The "Blind Overwriting" reliability problem can be solved
--       if UPDATE commands use "sensitive updates", such as
--             SET balance = balance - 500 WHERE ...


-- Exercise 2.2a  "Lost Update Problem" fixed by locks,
--              (competition on a single resource, see Table 2.5a)
-- ---------------------------------------------------------------------

--     Competition on a single resource
--     using SELECT .. UPDATE scenarios both client A and B
--     tries to withdraw amounts from the same account.
--     Note that InnoDB uses MGL only for SERIALIZABLE isolation.
--     The application part is simulated using local parameters.
--

-- 0. First restoring the original contents by client A
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

-- 1. client A starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET @amountA = 200; -- amount to be transfered by A
SET @balanceA = 0;  -- init value
SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101;
SET @balanceA = @balanceA - @amountA;
SELECT @balanceA;

-- 2. Client B starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET @amountB = 500; -- amount to be transfered by B
SET @balanceB = 0;  -- init value
SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101;
SET @balanceB = @balanceB - @amountB;
```

```
SELECT @balanceB;

-- 3. client A continues
UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;

-- This will be blocked. The default lock timeout in MySQL/InnoDB is 90 seconds
-- so continue to step 4 without waiting for A

-- 4. client B continues
UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;

-- 5. client A continues without waiting for step 4
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;

--  6. client B continues if it can ..
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;

-- ------------------------------------------------------------
-- Questions:
-- a) Conclusion(s) reached?
```

For SERIALIZABLE isolation InnoDB uses MGL locking, which in our case will lead to deadlock and rollback of the victim transaction, while the "winner" transaction succeeds and data in database will not be corrupted.

```
-- b) What if 'SERIALIZABLE' is replaced by 'REPEATABLE READ' in both transactions?
```

For 'REPEATABLE READ' isolation InnoDB uses MVCC snapshot for reading. Read actions will not block writers. However writers block writers, so one of the transaction will wait until the other commits, and then update the database with **outdated data (this is a blind overwriting problem).**

```
-- ------------------------------------------------------------------

-- Exercise 2.2b  Competing SELECT-UPDATE scenarios
-- The same as experiment 4, but simplified without parameters.
-- ------------------------------------------------------------------

-- 0. First restoring the original contents by client A
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

-- 1. client A starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT balance FROM Accounts WHERE acctID = 101;

-- 2.  client B starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT balance FROM Accounts WHERE acctID = 101;

-- 3.  client A continues
UPDATE Accounts SET balance = balance - 200 WHERE acctID = 101;

-- 4.  client B continues without waiting for A
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;

-- 5.  client A continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;

-- 6. client B continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;

-- ------------------------------------------------------------
-- Question:
-- How do you explain what happened?
```

Let's look at the last steps in client B's session:

```
mysql> -- 4.  client B continues without waiting for A
mysql> UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

```
mysql> -- 6. client B continues
mysql> SELECT acctID, balance FROM Accounts WHERE acctID = 101;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |     800 |
+--------+---------+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

As a reliable service the server sorted out the deadlock situation, in which it was impossible both to A and B to proceed. Now transaction of A was able to proceed successfully, while B was adviced to retry its transaction. Step 6 of B was actually a separate transaction and B could now see the currently correct data in the database.

```
-- ----------------------------------------------------------------

-- Exercise 2.3   Competition on two resources in different order
--                using UPDATE-UPDATE scenarios (see Table 2.6)
-- ----------------------------------------------------------------
--
-- Client A transfers 100 euros from account 101 to 202
-- Client B transfers 200 euros from account 202 to 101
--
-- 0. First restoring the original contents by client A

SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

-- 1. client A starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;

-- 2. Client B starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE Accounts SET balance = balance - 200 WHERE acctID = 202;

-- 3. Client A continues
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;

-- 4. Client B continues
UPDATE Accounts SET balance = balance + 200 WHERE acctID = 101;

-- 5. Client A continues
SELECT * FROM Accounts;
COMMIT;

-- 6. Client B continues
SELECT * FROM Accounts;
COMMIT;

-- ------------------------------------------------------------
-- Question:
-- Conclusions reached?
```

Competing UPDATE-UPDATE scenarios in different order will always lead to deadlock.
See also the deadlock explanation in 2.2b.

```
-- ----------------------------------------------------------------

-- In the following we will experiment concurrency anomalies i.e.
-- data reliability risks known by ISO SQL standard.
-- First play with the experiment, see the results, and then try
-- to fix the experiment to avoid the anomaly
```

```
-- ------------------------------------------------------------------
-- Exercise 2.4     Dirty Read ?  (see Table 2.7)
-- ------------------------------------------------------------------

-- 0. First restoring the original contents by client A
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

-- 1. client A starts
SET AUTOCOMMIT = 0;
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;

-- 2. Client B starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM Accounts;
COMMIT;

-- 3.  Client A continues
ROLLBACK;
SELECT * FROM Accounts;
COMMIT;

-- ----------------------------------------------------------
-- Questions:
-- What can we say of the reliability of transaction B?
```

Read actions in READ UNCOMMITTED isolation are not protected by S-locks, so there is a risk of reading incorrect data from the database.

```
-- How can we solve the problem?
```

If we need to get reliable data from database (as we usually do), we should protect the reading by explicit locking or proper isolation level.
Note that for READ COMMITTED and REPEATABLE READ InnoDB will return committed snapshot data, so not necessarily the latest content in the database! Accessing the latest content in MySQL can only be guaranteed by explicit locking or SERIALIZABLE isolation.
You may find it interesting to compare this behavior between different DBMS products.

```
-- ------------------------------------------------------------------
-- Exercise 2.5     Non-repeatable Read ? (see Table 2.8)
-- ------------------------------------------------------------------

-- 0. First restoring the original contents by client A
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

-- 1. client A starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
--  Listing accounts having balance > 500 euros:
SELECT * FROM Accounts WHERE balance > 500;

mysql> --  Listing accounts having balance > 500 euros:
mysql> SELECT * FROM Accounts WHERE balance > 500;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |    1000 |
|    202 |    2000 |
+--------+---------+
2 rows in set (0.00 sec)

-- 2. Client B starts
SET AUTOCOMMIT = 0;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
SELECT * FROM Accounts;
```

```
mysql> SELECT * FROM Accounts;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |     500 |
|    202 |    2500 |
+--------+---------+
2 rows in set (0.00 sec)
COMMIT;

-- 3. Client A continues
-- Can we see the same accounts again?
SELECT * FROM Accounts WHERE balance > 500;
COMMIT;

mysql> -- Can we see the same accounts again?
mysql> SELECT * FROM Accounts WHERE balance > 500;
+--------+---------+
| acctID | balance |
+--------+---------+
|    202 |    2500 |
+--------+---------+
1 row in set (0.00 sec)

-- ------------------------------------------------------------
-- Questions:
-- a) Does transaction A read in step 3 the same result it reads in step number 1?
```

No, in step 3 client A sees the contents of the database as updated by client B.

```
-- b) How about setting transaction A's isolation level to REPEATABLE READ?
```

Let's look at our `REPEATABLE READ` test results:

```
mysql> -- 2. Client B starts
mysql> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> -- 3. Client A continues
mysql> -- Can we see the same accounts again?
mysql> SELECT * FROM Accounts WHERE balance > 500;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |    1000 |
|    202 |    2000 |
+--------+---------+
2 rows in set (0.00 sec)
```

Conclusion:
REPEATABLE READ in InnoDB uses MVCC snapshot isolation **for reading**. So it does not prevent client B from the updates, but client A still sees only the snapshot in step 3. MGL based isolation level would have blocked client B from the updates, and therefore the result of client A would have been the same.

```
-- ------------------------------------------------------------------
-- Exercise 2.6     Insert Phantom ?  (see Table 2.9)
-- ------------------------------------------------------------------
-- Note:  InnoDB uses Multi-Versioning for REPEATABLE READ.
--        This means that the client cannot see nor prevent the phantoms.
--

-- 0. First restoring the original contents by client A
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
```

```
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. client A starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
START TRANSACTION READ ONLY ;  -- not needed, just to show the option

-- 2. Client B starts
SET AUTOCOMMIT = 0;
INSERT INTO Accounts (acctID,balance) VALUES (301,3000);
COMMIT;

-- 3. client A does a SELECT
-- Accounts having balance > 1000 euros:
SELECT * FROM Accounts WHERE balance > 1000;

-- 4. Client B starts a new transaction
SET AUTOCOMMIT = 0;
INSERT INTO Accounts (acctID,balance) VALUES (302,3000);
COMMIT;

-- 5. Client A continues
-- Can we see the both new accounts 301 and 302 ?
SELECT * FROM Accounts WHERE balance > 1000;
COMMIT;

-- ------------------------------------------------------------
-- Questions:
-- a) Does the client B need to wait for transaction A?
```

No, transaction A only reads using MVCC, so it will not block client B's transactions.

```
-- b) Are the (newly inserted, by client B) accounts visible in
--    transaction A's environment?
```

Only the first inserted account 301 is visible to A, whereas 302 is an "insert phantom".

```
-- c) Does it affect to the resultset of step 5 if we change the order of
--    steps 2 and 3?
```

Yes, in that case also 301 would be an invisible phantom to A.

```
-- d) MySQL/InnoDB uses Multi-Versioning for REPEATABLE READ isolation,
--    but what is the point in time level of the read snapshot?
```

Based on our experiment the point in time of A's snapshot is the start time of the first SQL command of A (the SELECT in step 3), and not start time of step 1!

```
-- Task:  Consider preventing the phantoms
Replacing isolation level REPEATABLE READ by SERIALIZABLE would prevent the phantoms.


-- ------------------------------------------------------------------
-- Exercise 2.7    A Snapshot study with different kinds of Phantoms
--                 (see Table 2.10)
-- ------------------------------------------------------------------

-- 0. Setup the test by Client C in a new terminal window
mysql testdb
SET AUTOCOMMIT = 0;
DROP TABLE T;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first', 1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'fourth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;
-- Client C will continue in step 3.5

-- 1. client A starts
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
SELECT * FROM T WHERE i = 1;

mysql> SELECT * FROM T WHERE i = 1;
+----+--------------------+------+
| id | s                  | i    |
```

```
+----+--------------------+------+
|  1 | first              |    1 |
|  3 | third              |    1 |
|  5 | to be or not to be |    1 |
+----+--------------------+------+
3 rows in set (0.00 sec)
```

```
-- 2. Client B starts,
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE T SET s = 'Update by B' WHERE id = 1;
INSERT INTO T (id, s, i) VALUES (6, 'Insert Phantom', 1);
UPDATE T SET s = 'Update Phantom', i = 1 WHERE id = 2;
DELETE FROM T WHERE id = 5;
SELECT * FROM T;
```

```
mysql> SELECT * FROM T;
+----+----------------+------+
| id | s              | i    |
+----+----------------+------+
|  1 | Update by B    |    1 |
|  2 | Update Phantom |    1 |
|  3 | third          |    1 |
|  4 | fourth         |    2 |
|  6 | Insert Phantom |    1 |
+----+----------------+------+
5 rows in set (0.00 sec)
```

```
-- 3. Client A continues
-- Let's repeat the query and try some updates
SELECT * FROM T WHERE i = 1;
INSERT INTO T (id, s, i) VALUES (7, 'inserted by A', 1);
UPDATE T SET s = 'update by A inside snapshot' WHERE id = 3;
UPDATE T SET s = 'update by A outside snapshot' WHERE id = 4;
UPDATE T SET s = 'update by A after update by B' WHERE id = 1;
This is blocked by B's update of id 1

-- ------------------------------------------------------------
-- 3.5  Client C queries in a new terminal window
--      to verify the current content in the database
mysql testdb
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM T;
mysql> SELECT * FROM T;
+----+-------------------------------+------+
| id | s                             | i    |
+----+-------------------------------+------+
|  1 | Update by B                   |    1 |
|  2 | Update Phantom                |    1 |
|  3 | update by A inside snapshot   |    1 |
|  4 | update by A outside snapshot  |    2 |
|  6 | Insert Phantom                |    1 |
|  7 | inserted by A                 |    1 |
+----+-------------------------------+------+
6 rows in set (0.00 sec)
-- ------------------------------------------------------------
```

```
-- 4. Client B continues without waiting for A
COMMIT;
SELECT * FROM T;
mysql> SELECT * FROM T;
+----+----------------+------+
| id | s              | i    |
+----+----------------+------+
|  1 | Update by B    |    1 |
|  2 | Update Phantom |    1 |
|  3 | third          |    1 |
|  4 | fourth         |    2 |
|  6 | Insert Phantom |    1 |
+----+----------------+------+
5 rows in set (0.00 sec)
```

```
-- 5. Client A continues
SELECT * FROM T WHERE i = 1;
mysql> SELECT * FROM T WHERE i = 1;
+----+--------------------------------+------+
| id | s                              | i    |
+----+--------------------------------+------+
|  1 | update by A after update by B |    1 |
|  3 | update by A inside snapshot    |    1 |
|  5 | to be or not to be             |    1 |
|  7 | inserted by A                  |    1 |
+----+--------------------------------+------+
4 rows in set (0.00 sec)
UPDATE T SET s = 'update after delete?' WHERE id = 5;
mysql> UPDATE T SET s = 'update after delete?' WHERE id = 5;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
SELECT * FROM T WHERE i = 1;
mysql> SELECT * FROM T WHERE i = 1;
+----+--------------------------------+------+
| id | s                              | i    |
+----+--------------------------------+------+
|  1 | update by A after update by B |    1 |
|  3 | update by A inside snapshot    |    1 |
|  5 | to be or not to be             |    1 |
|  7 | inserted by A                  |    1 |
+----+--------------------------------+------+
4 rows in set (0.00 sec)
COMMIT;

-- 6. Client A continues with a new transaction
SELECT * FROM T;
mysql> SELECT * FROM T;
+----+--------------------------------+------+
| id | s                              | i    |
+----+--------------------------------+------+
|  1 | update by A after update by B |    1 |
|  2 | Update Phantom                 |    1 |
|  3 | update by A inside snapshot    |    1 |
|  4 | update by A outside snapshot   |    2 |
|  6 | Insert Phantom                 |    1 |
|  7 | inserted by A                  |    1 |
+----+--------------------------------+------+
6 rows in set (0.00 sec)
COMMIT;

-- 7. Client C does the final select
SELECT * FROM T;
mysql> SELECT * FROM T;
+----+--------------------------------+------+
| id | s                              | i    |
+----+--------------------------------+------+
|  1 | update by A after update by B |    1 |
|  2 | Update Phantom                 |    1 |
|  3 | update by A inside snapshot    |    1 |
|  4 | update by A outside snapshot   |    2 |
|  6 | Insert Phantom                 |    1 |
|  7 | inserted by A                  |    1 |
+----+--------------------------------+------+
6 rows in set (0.00 sec)
EXIT;
-- ------------------------------------------------------------
-- Questions:
-- a) Are the insert and update made by transaction B visible in
--    transaction A's environment?
```

Since client A does a SELECT already in step 1 the inserts and updates made by B will not be visible in client A's snapshot.

```
-- b) What happens when A tries to update the row 1 updated by transaction B?
It seems that MySQL has a bug: The snapshot of client A's second transaction
still uses the same snapshot timestamp of the first transaction (?)

-- c) What happens when A tries to update the row 5 deleted by transaction B?
The update of row 5 in step 5 does not find the row, but the ghost of the row
still appears in the snapshot, even at step 6.

--          ** End of exercises **
```

```
-- ================================================================
-- Advanced Topics
-- A Stored procedure sample of MySQL
-- demonstrating programmatic access and workarounds
-- ----------------------------------------------------------------
```

Note: While writing this text the MySQL version in DebianDB is the following

```
mysql> SELECT version();
+-----------+
| version() |
+-----------+
| 5.6.12    |
+-----------+
```

For raising exceptions in MySQL we use SIGNAL commands. See example in AdvTopics_MySQL.txt.
According to MySQL documentation "the SIGNAL and RESIGNAL statements are not supported until MySQL 5.5"
and "The GET DIAGNOSTICS statement is not supported until MySQL 5.6.".

In our tests we have found that MySQL 5.1 (and 5.6) does not use CHECK constraints even if it accepts the syntax on row-level. So we need to take care of it either programmatically
or by triggers as presented in AdvTopics_MySQL.txt.

Note: The presented use of COMMIT and ROLLBACK in the following procedure
is NOT a good practice!
The purpose of this sample is only to demonstrate programmatic access
and diagnostic workarounds.

Note: CREATE PROCEDURE is command which contains SQL statements ended by semicolons (;), so we need
to define a temporary delimiter for the CREATE PROCEDURE command, as follows:

```
DELIMITER //
DROP PROCEDURE BankTransfer //
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
                               IN toAcct   INT,
                               IN amount   INT,
                               OUT msg     VARCHAR(100)
                              )
P1: BEGIN
    DECLARE rows INT ;
    DECLARE newbalance INT;
    SELECT COUNT(*) INTO rows FROM Accounts WHERE acctID = fromAcct;
    UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
    SELECT balance INTO newbalance FROM Accounts WHERE acctID = fromAcct;
    IF rows = 0 THEN
        ROLLBACK;
        SET msg = CONCAT('rolled back because of missing account ', fromAcct);
    ELSEIF newbalance < 0 THEN
            ROLLBACK;
            SET msg = CONCAT('rolled back because of negative balance of account ', fromAcct);
        ELSE
            SELECT COUNT(*) INTO rows FROM Accounts WHERE acctID = toAcct;
            UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
            IF rows = 0 THEN
                ROLLBACK;
                SET msg = CONCAT('rolled back because of missing account ', toAcct);
            ELSE
                COMMIT;
                SET msg = 'committed';
            END IF;
        END IF;
END P1 //
DELIMITER ;


-- ----------------------------------------------------------------
-- Test script for the BankTransfer procedure:
-- ----------------------------------------------------------------
USE testdb;
SET AUTOCOMMIT = 0;
```

```
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
SET @out = ' ';
CALL BankTransfer (101, 202, 100, @out);
SELECT @out;
SELECT * FROM Accounts;
CALL BankTransfer (100, 202, 100, @out);
SELECT @out;
SELECT * FROM Accounts;
CALL BankTransfer (101, 200, 100, @out);
SELECT @out;
SELECT * FROM Accounts;
SELECT @out;
CALL BankTransfer (101, 202, 2000, @out);
SELECT @out;
SELECT * FROM Accounts;
ROLLBACK;
EXIT;
```

In the following we have the test results when we have not yet created the triggers for the CHECK constraint:

```
mysql> SELECT * FROM Accounts;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |    1000 |
|    202 |    2000 |
+--------+---------+
2 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SET @out = ' ';
Query OK, 0 rows affected (0.00 sec)

mysql> CALL BankTransfer (101, 202, 100, @out);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @out;
+-----------+
| @out      |
+-----------+
| committed |
+-----------+
1 row in set (0.00 sec)

mysql> SELECT * FROM Accounts;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |     900 |
|    202 |    2100 |
+--------+---------+
2 rows in set (0.00 sec)

mysql> CALL BankTransfer (100, 202, 100, @out);
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SELECT @out;
+-------------------------------------------+
| @out                                      |
+-------------------------------------------+
| rolled back because of missing account 100 |
+-------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT * FROM Accounts;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |     900 |
|    202 |    2100 |
```

```
+--------+---------+
2 rows in set (0.00 sec)

mysql> CALL BankTransfer (101, 200, 100, @out);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @out;
+-----------------------------------------+
| @out                                    |
+-----------------------------------------+
| rolled back because of missing account 200 |
+-----------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM Accounts;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |     900 |
|    202 |    2100 |
+--------+---------+
2 rows in set (0.00 sec)

mysql> SELECT @out;
+-----------------------------------------+
| @out                                    |
+-----------------------------------------+
| rolled back because of missing account 200 |
+-----------------------------------------+
1 row in set (0.01 sec)

mysql> CALL BankTransfer (101, 202, 2000, @out);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @out;
+-------------------------------------------------------+
| @out                                                  |
+-------------------------------------------------------+
| rolled back because of negative balance of account 101 |
+-------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM Accounts;
+--------+---------+
| acctID | balance |
+--------+---------+
|    101 |     900 |
|    202 |    2100 |
+--------+---------+
2 rows in set (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> EXIT;
Bye
student@debianDB:~$
-- ------------------------------------------------------------------------
-- Another version of the procedure is available in AdvTopics_MySQL.txt
```