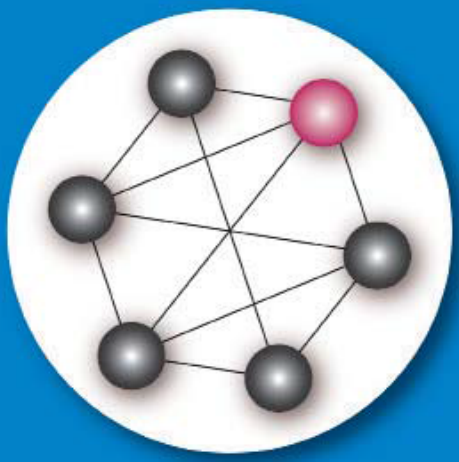


DBTechNet

DBTech VET

SQL Transactions

Theory and hands-on exercises

In English

Lifelong Learning Programme

This publication has been developed in the framework of the project
DBTech VET Teachers (DBTech VET). Code: 2012-1-FI1-LEO05-09365.

DBTech VET is Leonardo da Vinci Multilateral Transfer of Innovation project,
funded by the European Commission and project partners.

www.DBTechNet.org
DBTech VET

SQL Transactions – Theory and Hands-on Exercises
Version 1.4 of the first edition 2013

Authors: Martti Laiho, Dimitris A. Dervos, Kari Silpiö
Production: DBTech VET Teachers project

ISBN 978-952-93-2420-0 (paperback)
ISBN 978-952-93-2421-7 (PDF)

SQL Transactions – Student's Guide

Objectives

Reliable data access must be based on properly designed SQL transactions with ZERO TOLERANCE for incorrect data. A programmer who does not understand the required transaction technology can easily violate the data integrity in a database contents and block or slow down the production performance. Just like in traffic, the rules in accessing database must be known and obeyed.

The purpose of this tutorial is to present the basics of transaction programming using the mainstream DBMS products. Some typical problems and transaction tuning possibilities are also presented.

Target user groups

The target groups of this tutorial include teachers, trainers, and students in vocational institutes and industry-oriented higher education institutes. Also application developers in ICT industry may find this as a useful introduction for understanding other mainstream DBMS products than they are using daily.

Prerequisites

Readers should have hands-on skills on the use of basic SQL of some DBMS product.

Learning methods

Learners are encouraged to experiment and verify themselves the topics presented in this tutorial using real DBMS products. For that purpose a free virtual database laboratory and sample scripts have been compiled and links to these are provided in the "References and Links".

Content

1 SQL Transaction – a Logical Unit of Work

1.1 Introducing transactions

In everyday life, people conduct different kind of business transactions buying products, ordering travels, changing or canceling orders, buying tickets to concerts, paying rents, electricity bills, insurance invoices, etc. Transactions do not relate only to computers, of course. Any type of human activity comprising a **logical unit of work** meant to either be executed as a whole or to be cancelled in its entirety comprises a **transaction**.

Almost all information systems utilize the services of some database management system (DBMS) for storing and retrieving data. Today's DBMS products are technically sophisticated securing data integrity in their databases and providing fast access to data even to multiple concurrent users. They provide reliable services to applications for managing persistency of data, but only if applications use these reliable services properly. This is done by building the **data access** parts of the application logic using database **transactions**.

Improper transaction management and control by the application software may, for example, result in

- customer orders, payments, and product shipment orders being lost in the case of a web store
- failures in the registration of seat reservations or double-bookings to be made for train/airplane passengers
- lost emergency call registrations at emergency response centers etc.

Problematic situations like the above occur frequently in real life, but the people in charge often prefer not to reveal the stories to the public. The mission of the DBTech VET project is to set up a framework of best practices and a methodology for avoiding such type of unfortunate incidents.

Transactions are recoverable units of data access tasks in terms of database content manipulation. They also comprise units of recovery for the entire database in case of system crashes, as presented in Appendix 3. They also provide basis for concurrency management in multi-user environment.

1.2 Client/Server Concepts in SQL Environment

In this tutorial, we focus on data access using **SQL transactions** while executing SQL code interactively, but keeping in mind that the corresponding programmatic data access uses a slightly different paradigm which we present by means of an example in Appendix 2.

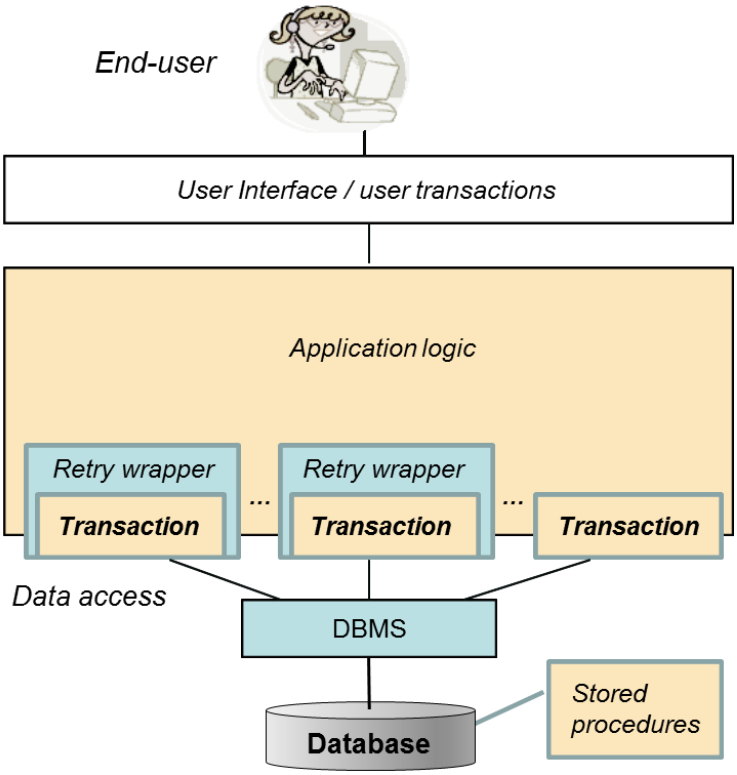


Figure 1.1 Position of SQL transactions in application layers

Figure 1.1 presents a simplified view of the architecture of a typical database application, positioning the database transactions on a different software layer than the user interface layer. From the end user's point of view, for processing a business transaction one or more use cases can be defined for the application and implemented as **user transactions**. A single user transaction may involve multiple SQL transactions, some of which will involve retrieval, and usually the final transaction in the series updating the contents in the database. **Retry wrappers** in the application logic comprise the means for implementing programmatic retry actions in case of concurrency failures of SQL transactions.

To properly understand SQL transactions, we need to agree on some basic concepts concerning the client-server handshaking dialogue. To access a database, the *application* needs to initiate a ***database connection*** which sets up the context of an ***SQL-session***. For simplicity, the latter is said to comprise the ***SQL-client***, and the database server comprises the ***SQL-server***. From the database server point of view, the application uses database services in client/server mode by passing **SQL commands** as parameters to functions/methods via a data access API (application programming interface)[1]. Regardless of the data access interface used, the "logical level" dialog with the server is based on the SQL language, and reliable data access is materialized with the proper use of SQL transactions**.**

[1] Such as ODBC, JDBC, ADO.NET, LINQ, etc. depending on the programming language used, such as C++, C#, Java, PHP, etc.

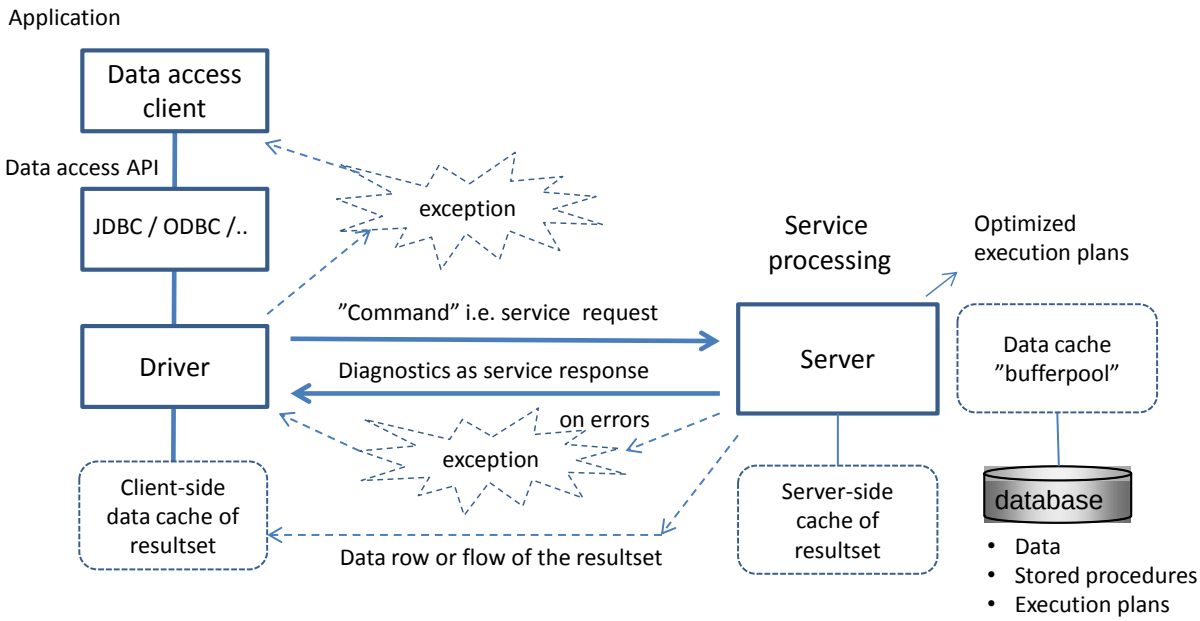


Figure 1.2 SQL command processing explained

Figure 1.2 explains the "round trip", the processing cycle of an SQL command, started by the client as a **service *request*** to the server using a middleware stack and network services, for processing by server, and the returned response to the request. The SQL command may involve one or more ***SQL statements***. The SQL statement(s) of the command are parsed, analyzed on the basis of the database metadata, then optimized and finally executed. To improve the performance degradation due to slow disk I/O operations, the server retains all the recently used rows in a RAM-residing buffer pool and all data processing takes place there.

The execution of the entered SQL command in the server is *atomic* in the sense that the whole SQL command has to succeed; otherwise the whole command will be cancelled (rolled back). As a response to the SQL command, the server sends *diagnostic* message(s) reporting of the success or failures of the command. Command execution errors are reflected to the client as a sequence of exceptions. However, it is important to understand that SQL statements like UPDATE or DELETE succeed in their execution even in the absence of matching rows. From the application's point of view, such cases may be seen to comprise a failure, but as far as command execution is concerned, they comprise a success. Therefore the application code needs to carefully check the diagnostic messages issued by the server to determine the number of rows that have been affected by the operation in question.

In case of a SELECT statement, the generated *resultset* will be fetched row-by-row at the client's side; resultset rows are either fetched directly from the server, over the network, or they are fetched from the client's cache.

1.3 SQL Transactions

When the application logic needs to execute a sequence of SQL commands in an atomic fashion, then the commands need to be grouped as a logical unit of work (LUW) called SQL transaction which, while processing the data, transforms the database from a consistent state to another consistent state, and thus be considered as **unit of consistency**. Any successful execution of the transaction is ended by a **COMMIT** command, whereas unsuccessful execution need to be

ended by a **ROLLBACK** command which automatically recovers from the database all changes made by the transaction. Thus SQL transaction can also be considered as **unit of recovery**. The advantage of the ROLLBACK command is that when the application logic programmed in the transaction cannot be completed, there is no need for conducting a series of reverse operations command-by-command, but the work can be cancelled simply by the ROLLBACK command, the operation of which will always succeed. Uncommitted transactions in case of broken connections, or end of program, or at system crash will be automatically rolled back by the system. Also in case of concurrency conflicts, some DBMS products will automatically rollback a transaction, as explained below.

The advantage of the (standard SQL) ROLLBACK statement is that when the application logic implemented in the transaction's body cannot run to completion, there is no need for (the application logic, or the programmer/user) to conduct a series of reverse, command-by-command, operations. Instead, all the changes already made to the database by the incomplete transaction get cancelled simply by processing a ROLLBACK statement which guaranteed to always successfully run to completion. Transactions that are active (i.e. have not committed) at the time of a system crash are automatically rolled back when the system is brought back to normal operation.

Note: According to the ISO SQL standard, and as implemented, for example, in DB2 and Oracle, any SQL command, in the beginning of an SQL session or following the end of a transaction, will automatically start a new SQL transaction. This case is said to comprise an ***implicit start*** of an SQL transaction.

Some DBMS products, for example, SQL Server, MySQL/InnoDB, PostgreSQL and Pyrrho operate by default in the **AUTOCOMMIT** mode. This means that the result of every single SQL command will is automatically committed to the database, thus the effects/changes made to the database by the statement in question **cannot be rolled back.** So, in case of errors the application needs do reverse-operations for the logical unit of work, which may be impossible after operations of concurrent SQL-clients. Also in case of broken connections the database might be left in inconsistent state. In order to use real transactional logic, one needs to start every new transaction with an explicit start command, like: BEGIN WORK, BEGIN TRANSACTION, or START TRANSACTION, depending on the DBMS product used.

Note: In MySQL/InnoDB, an ongoing SQL session can be set to use either implicit or explicit transactions by executing the statement:

```
SET AUTOCOMMIT = { 0 | 1 }
```

where 0 implies the use of implicit transactions, and 1 means operating in AUTOCOMMIT mode.

Note: Some DBMS products, such as Oracle, **implicitly commit** transaction upon executing any SQL DDL statement (e.g. CREATE, ALTER or DROP of some database object, such as TABLE, INDEX, VIEW, etc.).

Note: In SQL Server, the whole **instance**, including its databases, can be configured to use implicit transactions, and a started **SQL session** (connection) can be switched to use implicit transactions or to return to AUTOCOMMIT mode by executing the statement:

```
SET IMPLICIT_TRANSACTIONS { ON | OFF }
```

Note: Some utility programs, such as the command line processor (CLP) of IBM's DB2, and some data access interfaces, such as ODBC and JDBC, operate by default in the AUTOCOMMIT mode. For example, in the JDBC API, every transaction needs to be started by the following method call of the connection object:

```
<connection>.setAutoCommit(false);
```

Instead of a simple sequence of data access tasks, some SQL transactions may involve complex program logic. In such cases, the transaction logic will make choices at the execution time, depending on the information retrieved from the database. Even then, the SQL transaction can be regarded as an indivisible "logical unit of work" (LUW), which either succeeds or is rolled back. However, a failure in transaction does not usually automatically generate ROLLBACK[2], but should be diagnosed by the application code (see "Diagnosing the SQL errors" below) and the application itself is in charge of requiring the ROLLBACK.

1.4 Transaction Logic

Let's consider the following table of bank accounts

```
CREATE TABLE Accounts (
acctId  INTEGER NOT NULL PRIMARY KEY,
balance DECIMAL(11,2) CHECK (balance >= 0.00)
);
```

A typical textbook example of SQL transactions is the transferring of a certain amount (for example 100 euros) from one account to another:

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE acctId = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctId = 202;
COMMIT;
```

If the system fails, or the client-server network connection breaks down after the first UPDATE command, the transaction protocol guarantees that no money will be lost from account number 101 since, finally, the transaction will be rolled back. However, the transaction above is far from being a reliable one:

a) In case either one of the two bank accounts does not exist, the UPDATE commands would still execute and "succeed" in terms of SQL. Therefore, one should inspect the available SQL diagnostics and check the number of rows that have been affected by each one of the two UPDATE commands.

b) In case the first UPDATE command fails due to the 101 account's balance going negative (consequently: violating the corresponding CHECK constraint), then proceeding with and successfully executing the second UPDATE command will lead into a logically erroneous state in the database.

[2] But of the DBMS products in our DebianDB laboratory, after an error in transaction PostgreSQL will reject all commands and accepts only ROLLBACK.

From this simple example, we realize that application developers need to be aware of the way DBMS products behave, and of how the SQL diagnostics are inspected in the data access interface the API used. Even then, there is much more to be learned and a number of database tuning operations to be conducted.

1.5 Diagnosing SQL errors

Instead of a simple sequence of data access tasks, some SQL transactions may involve complex program logic. In such cases, the transaction logic makes choices at execution time, depending on the information retrieved from the database. Even then, the SQL transaction can be regarded as an indivisible "logical unit of work", which either succeeds or it is rolled back. However, a transaction failure does not usually automatically generate a ROLLBACK[3]. After every SQL command the application code needs to inspect the diagnostic errors issued by the server and determine whether to issue a ROLLBACK, or not.

For this purpose the early ISO SQL-89 standard defined the integer typed indicator **SQLCODE** which at the end of every SQL command by value 0 indicates successful execution, while value 100 indicates that no matching rows were found, all other values are product specific. Other positive values indicate some warnings and negative values indicate different kind of errors, and are explained in the corresponding product reference manuals.

In the ISO SQL-92 standard, the SQLCODE was deprecated and the new indicator **SQLSTATE** was introduced, a string of 5 characters, of which the first 2 characters comprise the class code of SQL errors and warnings, and the last 3 characters encode subclass codes. The all zeros string ("00000") in SQLSTATE stands for a successful execution. Hundreds of other values have been standardized (e.g. for SQL constraint violations), yet a large number of extra values remain as DBMS product-specific. The SQLSTATE values starting with "40" indicate a lost transaction for example due to concurrency conflict, maybe an error in a stored routine, lost connection, or server problem.

To provide better diagnostic information to client application on what has occurred on the server side, X/Open group has extended the SQL language by GET DIAGNOSTICS statement which can used to get more detailed information items and can be repeated to browse through the diagnostics records reporting multiple errors or warnings. This has also been extended in ISO SQL standards since SQL:1999, but only parts of the items have been implemented in DBMS products, for example DB2, Mimer, and MySQL 5.6. The following MySQL 5.6 example presents the idea of reading the diagnostic items

```
GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
                            @sqlcode  = MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;
```

Some SQL implementations with procedural features make some diagnostic indicators available in the special registers or functions of the language. For example, in MS SQL Server's Transact-SQL (also called as T-SQL), some diagnostic indicators are available in @@-variables such as @@ERROR for the native error code, or @@ROWCOUNT for the number of rows just previously processed.

[3] PostgreSQL in DBTech VET's Debian VM, following a transaction error, rejects all commands and accepts only a ROLLBACK

In the native IBM DB2 SQL language, the ISO SQL diagnostic indicators SQLCODE and SQLSTATE are available in the language's procedural extension for stored procedures, as in the following

```
<SQL statement>
IF (SQLSTATE <> '00000') THEN
     <error handling>
END IF;
```

In the BEGIN-END blocks of the Oracle PL/SQL language, error (or exception) handling is encoded at the bottom section of the code by means of the EXCEPTION directive as follows:

```
BEGIN
     <processing>
EXCEPTION
WHEN <exception name> THEN
     <exception handling>;
...
WHEN OTHERS THEN
     err_code := sqlcode;
     err_text := sqlerrm;
     <exception handling>;
END;
```

The earliest diagnostic record related implementations can be found in ODBC and SQLExceptions and SQLWarnings of JDBC. In JDBC API of the Java language, SQL errors raise SQL exceptions. The latter need to be caught via try-catch control structures in the application logic as follows (see Appendix 2):

```
... throws SQLexception {
...
try {
     ...
     <JDBC statement(s)>
}
catch (SQLException ex) {
     <exception handling>
}
```

In JDBC, the diagnostic item *rowcount* i.e. number of processed rows is returned by the execute methods.

1.6 Hands-on laboratory practice

> **Note**: Don't believe all that you read! For developing reliable applications, you need to **experiment and verify** yourself with the services of your DBMS product. DBMS products differ in the way they support even the basics of SQL transaction services.

In Appendix 1, we present tests on explicit and implicit SQL transactions, COMMIT and ROLLBACK, and transaction logic using MS SQL Server, but you should try them yourself in order to verify the SQL Server behavior presented in parts A1.1 – A1.2. For your experiments, you may download SQL Server Express for free from Microsoft's website.

In the Hands-on Laboratory (**HoL**) part of lesson number one, the trainee is introduced to the use of the DBTechNet's free virtual database laboratory DebianDB that comes bundled with a number of pre-installed free DBMS products like IBM DB2 Express-C, Oracle XE, MySQL GA, PostgreSQL, and Pyrrho. MySQL GA is not be the most reliable DBMS product in our lab, but since it is widely used in schools, in the following we will use it to conduct a first set of transaction-based data manipulation tasks.

So, welcome to the "Transaction Mystery Tour" using MySQL. The behavior of MySQL depends on which database engine is in use. The earlier default engine of MySQL did not even support transactions, but starting from MySQL version 5.1 the default database engine is InnoDB which supports transactions. Still, occasionally some services may produce surprising results.

> **Note**: The series of experiments follow the same theme for all DBMS products in DebianDB and can be found in the script files of type Appendix1_<dbms>.txt stored in "**Transactions"** directory of user "student". The test series are not planned to demonstrate problems in MySQL, but we don't avoid presenting them, since application developers need to know also the product bugs and to be able to apply workarounds to the bugs.

We expect that the reader has already consulted the DBTech VET document entitled "Quick Start to the DebianDB Database Laboratory" (Quick Start Guide); a document that explains the installation and the initial configuration tasks that need be carried out to the DebianDB, once the latter becomes operational in the host virtualization environment (usually: Oracle's VirtualBox).

Once the DebianDB is up and running, it is worth noting that, by default, the user is logged on as (username, password) = (**student**, **password**). To be able to create the first MySQL database, the user needs to switch to using the (username, password) = (**root**, **P4ssw0rd**) account, as explained in the Quick Start Guide. This is done by clicking on the "Terminal/Use the command line" icon, in the top menu bar of the Virtual Machine (Figure 1.4).

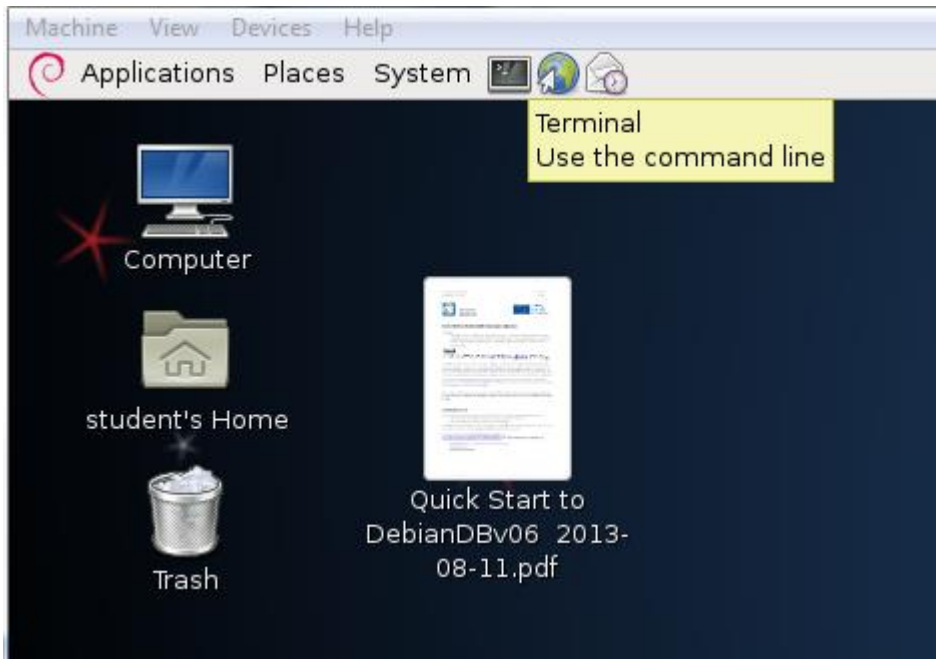


Figure 1.4 The "Terminal / Use the command line" Virtual Machine menu icon

Next, the following Linux commands are entered while in the terminal/command line window (Figure 1.5) starting the **`mysql`** client program:

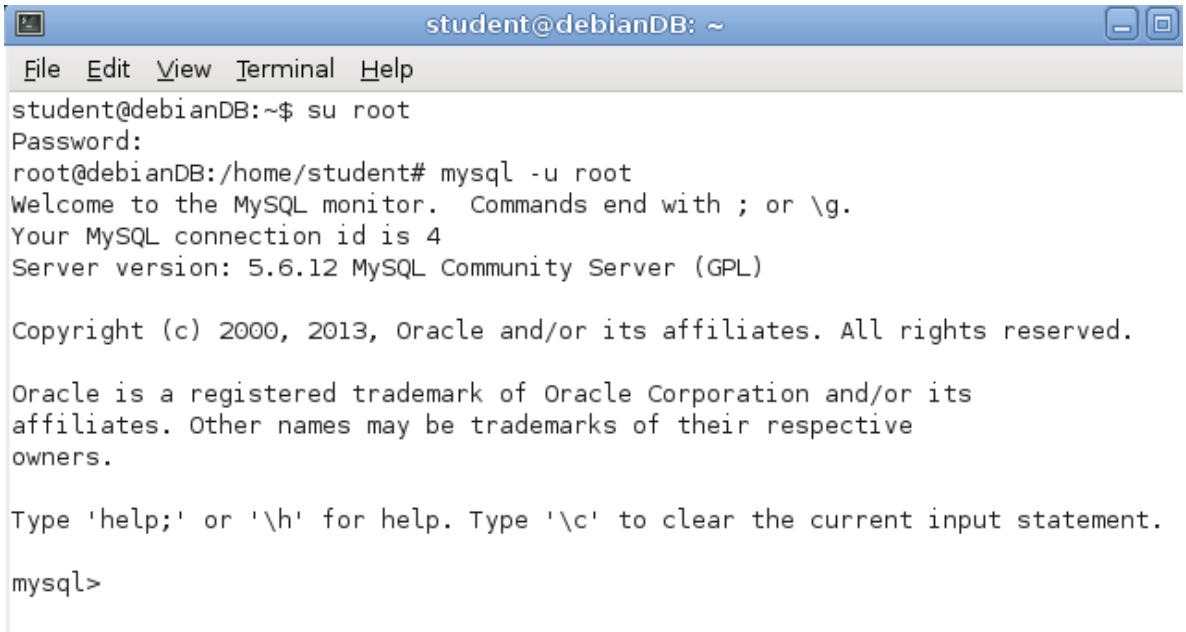


Figure 1.5 Initiating a MySQL session as 'root' user

The SQL command that will create a new database named "TestDB" is the next one to be entered, as follows:

```
-- ---------------------------------
CREATE DATABASE TestDB;
-- ---------------------------------

To grant access and all possible privileges in the created database TestDB,
the "root" user needs to enter the following command:
```

```
-- ---------------------------------
GRANT ALL ON TestDB.* TO 'student'@'localhost';
-- ---------------------------------
```

It is now time to exit from/terminate the "root" user EXIT from the MySQL session, and exit the root's session, returning to the session of "student" user by issuing the following two commands:

```
-- ---------------------------------
EXIT;
exit
```

> **Note**: If, during the self-practicing session, it so happens and the DebianDB screen turns black, prompting for a password in order to become active again, the password that need be entered is that of the "student" account: "password" ☺

Now, the default user "student" can start MySQL client and access the TestDB database as follows:

```
-- ---------------------------------
mysql
use TestDB;
-- ---------------------------------
```

This is the beginning of a new MySQL session.

EXERCISE 1.1

We will now create a new table named "T", having three columns: id (of type integer, the primary key), s (of type character string with a length varying from 1 to 40 characters), and si (of type small integer):

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

After every SQL command the MySQL client displays some diagnostics on the execution.

To make sure that the table is there, having the desirable structure, the use of MySQL's extended DESCRIBE command comes in handy:

```
DESCRIBE T;
```

> **Note**: MySQL in Linux is case insensitive, with the exception of table and database names. This means that the following work fine: "Describe T", "describe T", "create TaBle T ...", but "use testDB", and "describe t" refer to a database and a table other than the ones intended in this hands-on laboratory.

It is now time to append/insert some rows to the newly created table:

```
-- --------------------------------
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 ;
-- --------------------------------
```

A "SELECT * FROM T" command confirms that the three new lines have been appended to the table (note the NULL values registered under the "si" column).

> **Note**: Always make sure to first type a semicolon (";") at the end of each one command, and then hit "enter".

Having in mind what has been said so far on the topic of SQL transactions, an attempt is made to cancel/rollback the current transaction, by issuing the following command:

```
-- --------------------------------
ROLLBACK;
SELECT * FROM T ;
-- --------------------------------
```

It appears to have worked, however: after issuing a new "SELECT * FROM T" command, the table is seen to continue registering the three rows. A surprise…

The source of the surprise has a name: "AUTOCOMMIT". MySQL starts in the AUTOCOMMIT mode in which every transaction need to be started by "START TRANSACTION" command, and after end of the transaction MySQL returns to the AUTOCOMMIT mode again. To verify the above, the following set of SQL commands is executed:

```
-- --------------------------------
START TRANSACTION;
INSERT INTO T (id, s) VALUES (4, 'fourth');
SELECT * FROM T ;
ROLLBACK;

SELECT * FROM T;
-- --------------------------------
```

Question

- Compare the results obtained by executing the above two SELECT * FROM T statements.

EXERCISE 1.2

Next, the following statements are executed:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (5, 'fifth');
ROLLBACK;
SELECT * FROM T;
-----------------------------------
```

Questions

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

EXERCISE 1.3

Now, the session's AUTOCOMMIT mode is turned off, using the "SET AUTOCOMMIT" statement:

```
-----------------------------------
SET AUTOCOMMIT = 0;
-----------------------------------
```

First, all but one of the table's rows are deleted:

```
-----------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
-----------------------------------
```

Time to insert new rows, again:

```
-----------------------------------
INSERT INTO T (id, s) VALUES (6, 'sixth');
INSERT INTO T (id, s) VALUES (7, 'seventh');
SELECT * FROM T;
-----------------------------------
```

… and ROLLBACK:

```
-----------------------------------
ROLLBACK;
SELECT * FROM T;
-----------------------------------
```

Question

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

Note: While in the MySQL client in Linux terminal window environment, one may make use of the keyboard's up- and down-arrow keys to automatically move forward and backward across the text (statements) entered already. The latter is not always possible with the rest of the DBMS environments that are pre-installed in the DebianDB.

Note: Two consecutive dash ("-") characters in SQL signal that the rest of the command line is comment, i.e. the text following the two dashes up to when the next "enter" character is ignored by the MySQL parser.

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;
```

It is known already that some SQL statements are categorized as being of the Data Definition Language (DDL) type, and some others as being of the Data Manipulation Language (DML) type. Examples of the former are statements like CREATE TABLE, CREATE INDEX, and DROP TABLE, whereas examples of the second (DML) category are statements like SELECT, INSERT, and DELETE. Having the above in mind, it is worth investigating further the 'range of action" the ROLLBACK statement has in the following:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (9, 'will this be committed?');
CREATE TABLE T2 (id INT);
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 ?
-- Compare this with SELECT from a missing table as follows:
SELECT * FROM T3; -- assuming that we have not created table T3

SHOW TABLES;
DROP TABLE T2;
COMMIT;
```

Question

- Conclusion(s) reached?

EXERCISE 1.5

Table T's content is reset to its initial state:

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

It is now time to test whether the occurrence of an error triggers an automatic ROLLBACK in MySQL. The following SQL commands are executed:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (2, 'Error test starts here');
-- division by zero should fail
SELECT (1/0) AS dummy FROM DUAL;
-- Now update a non-existing row
UPDATE T SET s = 'foo' WHERE id = 9999 ;
-- and delete 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 of a 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;
-----------------------------------
```

Questions

a) What have we found out of automatic rollback on SQL errors in MySQL?
b) Is division by zero an error?
c) Does MySQL react on overflows?
d) What do we learn of the following results:

```
-----------------------------------
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> INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'
-----------------------------------
```

In this case the value "23000" displayed by the MySQL client is the standardized SQLSTATE value indicating violation of the primary key constraint, and 1062 is the corresponding error code of the MySQL product.

The diagnostics for the failing INSERT command, in our example above, can now be accessed by SQL's new GET DIAGNOSTICS statements in MySQL version 5.6, where, as follows:

```
mysql> GET DIAGNOSTICS @rowcount = ROW_COUNT;
Query OK, 0 rows affected (0.00 sec)

mysql> GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
    ->                             @sqlcode = MYSQL_ERRNO ;
Query OK, 0 rows affected (0.00 sec)
```

The variables starting with an at sign "@" are untyped local variables in the SQL language of MySQL. We make use of the MySQL local variables in our exercises just for simulating application level, since in this book we are playing mainly on SQL language level. In data access APIs the diagnostic indicator values can be read directly into host variables, but following example shows how we can read the values also from the local variables:

```
mysql> SELECT @sqlstate, @sqlcode, @rowcount;
+-----------+----------+-----------+
| @sqlstate | @sqlcode | @rowcount |
+-----------+----------+-----------+
| 23000     |     1062 |        -1 |
+-----------+----------+-----------+
1 row in set (0.00 sec)
```

EXERCISE 1.6

```
DROP TABLE Accounts;
SET AUTOCOMMIT=0;
```

The current version of MySQL doesn't support the syntax of **column-level CHECK** constraints which we have used for other DBMS products for this exercise, as follows:

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

It accepts the syntax of **row-level CHECK**s, as follows:

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

But even if it accepts the syntax, it does not use the CHECK constraints, as we will see on experimenting with the following test, which should fail:

```
INSERT INTO Accounts (acctID, balance) VALUES (100,-1000);
SELECT * FROM Accounts;
ROLLBACK;
```

Note: To keep the experiments comparable with other products we have not removed the CHECK constraint. All products have some bugs and application developers need to cope with these. The problem of missing support of CHECKs can be solved creating SQL triggers which are out of the scope of this tutorial, but interested readers will find our workaround examples in the script file `AdvTopics_MySQL.txt`.

```
-- Let's load also 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 that 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;
```

The next SQL transaction to be tested in one whereby an attempt is made to transfer 500 euros from account number 101 to a non-existent bank account number, say one with acctID = 777:

```
-- 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 one corresponds to a request for updating a non-existent account/row in the Accounts table?

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

c) Which diagnostic indicators of MySQL the user application could use to detect the problems in the above transactions ?

EXERCISE 1.7 – SQL Transaction as Unit of Recovery

We will now experiment with the "**unit of recovery**" property of SQL transactions in case of uncommitted transactions. To test this we will start a transaction, and then break the SQL connection of the MySQL client, and then reconnecting to the database to see if the changes made by the uncommitted transaction can be found in the database.

Take note of how an active (i.e. uncommitted) transaction gets affected by the broken connection (which may not be very rare case in Internet applications):

First, a new row is inserted into T:

```
-----------------------------------
SET AUTOCOMMIT = 0;
INSERT INTO T (id, s) VALUES (9, 'Let''s see what happens if ..');
SELECT * FROM T;
-----------------------------------
```

Next, the (client) session is interrupted violently with a "Control C" (Ctrl-C) command (Figure 1.6):

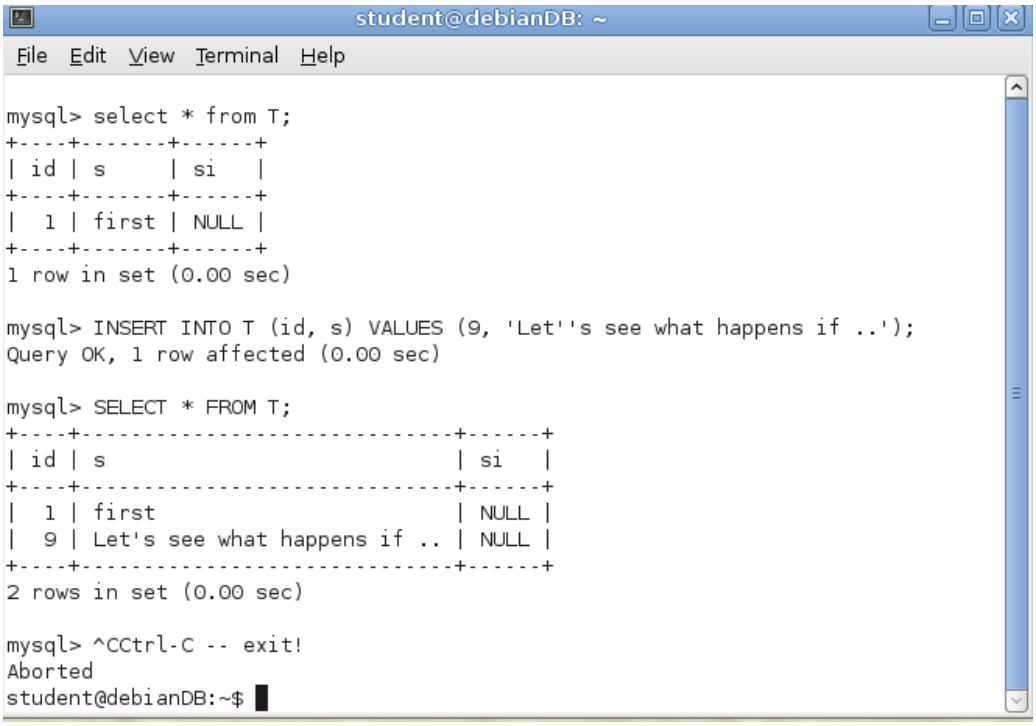


Figure 1.6 Simulating a "DBMS crash" situation

Next, the DebianDB terminal window is closed, a new one is opened, and a new MySQL session with the TestDB database is started:

```
mysql
USE TestDB;
SET AUTOCOMMIT = 0;
SELECT * FROM T;
COMMIT;
EXIT;
```

Question

- Any comments about T's row content, this time?

Note: All changes made into the database are traced into the **transaction log** of the database. Appendix 3 explains how transaction the database servers use transaction logs to recover the database up to the latest committed transaction before a system crash. The exercise 1.7 could be extended to test system crash, if instead cancelling only the MySQL client, the MySQL server process **`mysqld`** is aborted, as follows:

```
student@debianDB:~$ su root
Password:
root@debianDB:/home/student# ps -e | grep mysqld
 1092 ?        00:00:00 mysqld_debianDB
 1095 ?        00:00:00 mysqld_safe
 1465 ?        00:00:01 mysqld
root@debianDB:/home/student# kill 1465
```

2 Concurrent Transactions

Word of advice!

Don't believe everything that you hear/read about transactions support by various DBMS! In order to develop reliable applications, you need to experiment and verify by yourself the functionality of the services supported by your DBMS. DBMS products differ in the way they implement and support even basic of SQL transaction services.

An application program working correctly in single-user environment may accidently not work reliably when running concurrently with other clients (the instances of the same or other application programs) in multi-user environment presented in Figure 2.1.

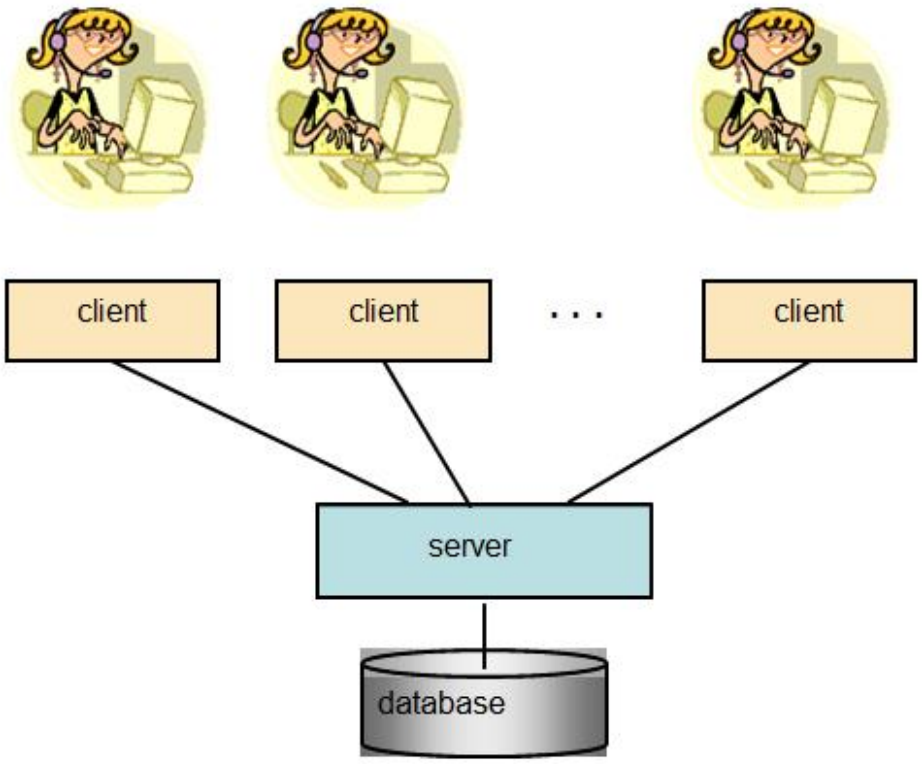


Figure 2.1 Multiple clients accessing the same database (in multi-user environment)

2.1 Concurrency Problems – Possible Risks of Reliability

Without having proper concurrency control services in the database product or lacking knowledge on how to use the services properly, the **content** in the database or **results** to our queries might become corrupted i.e. **unreliable**.

In the following, we cover the typical concurrency problems (anomalies):

- The lost update problem
- The dirty read problem, i.e. reading of uncommitted data of some concurrent transaction
- The non-repeatable read problem, i.e. repeated read may not return all same rows
- The phantom read problem, i.e. during the transaction some qualifying rows may not be seen by the transaction.

after which we will present how these problems can be solved according to ISO SQL standard and by the real DBMS products.

2.1.1 The Lost Update Problem

C. J. Date has presented the following example in figure 2.2 where two users at different automatic teller machines (ATMs) are withdrawing money from the same bank account, the balance of which is originally 1000 euros.

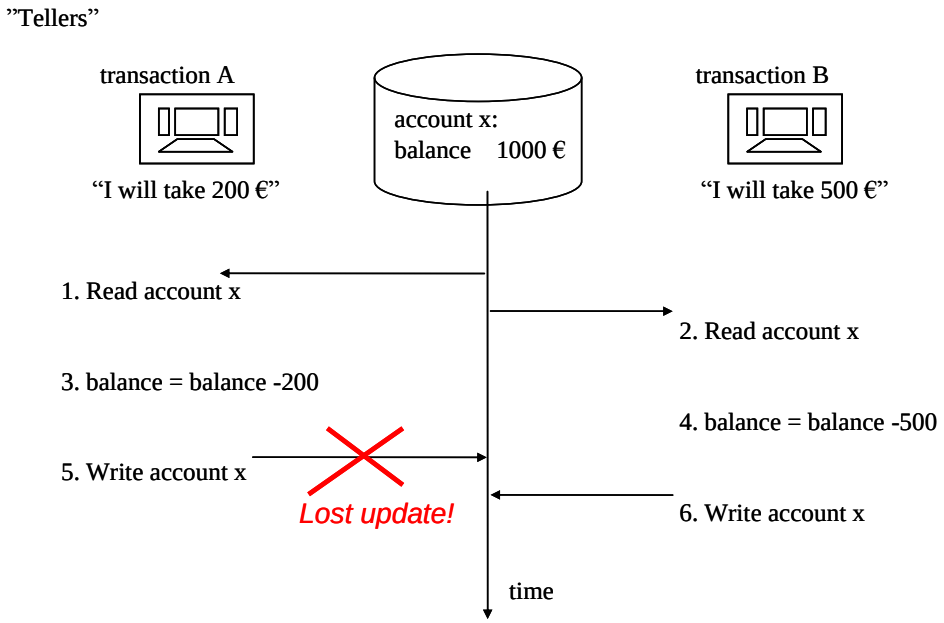


Figure 2.2 The Lost Update problem

Without concurrency control, the result of 800 euros of the write operation of the transaction A in step 5 will be lost in step 6, since the transaction B will write blindly the new balance of 500 euros it has calculated. If this happened before the end of transaction A, then the phenomena would be called "**Lost Update**". However, every modern DBMS product has implemented some kind of concurrency control mechanism, which protects the write operations against getting overwritten by concurrent transactions before the end of transaction.

If the scenario above is implemented as "SELECT .. UPDATE" sequences and protected by a locking scheme, then instead of Lost Update, the scenario will proceed to DEADLOCK (explained later), in which case, for example, transaction B would be rolled back by DBMS and the transaction A could proceed.

If the scenario above is implemented using ***sensitive updates*** (based on the current values) like

```
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 100;
```

and protected by locking scheme (to be explained later), then the scenario would work without problems.

2.1.2 The Dirty Read Problem

Dirty Read anomaly presented in figure 2.3 means that the transaction accepts the risk of reading also unreliable (uncommitted) data which may change or the updates into that data may get rolled back. This kind of transaction must not make any updates into the database since this would lead into corrupted data contents. In fact, any use of the uncommitted data is risky and may lead to incorrect decisions and actions.

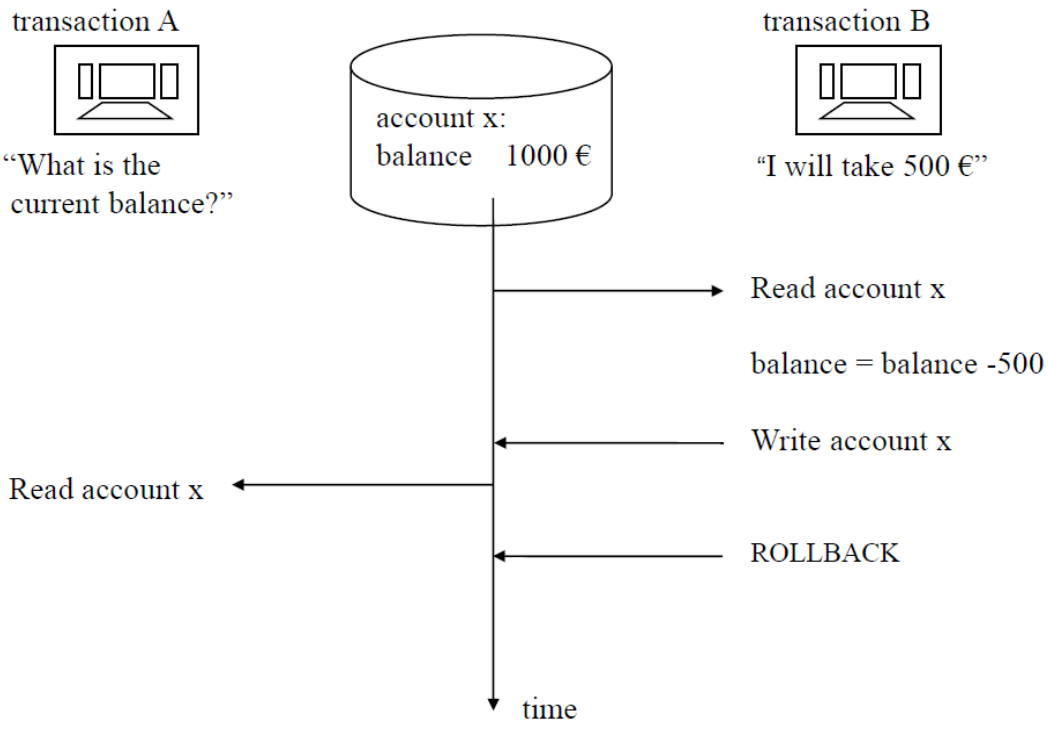


Figure 2.3 Example of a Dirty Read

2.1.3 The Non-repeatable Read Problem

Non-repeatable Read anomaly, presented in figure 2.4, means that the resultsets of the queries in the transaction are not stable, so that if the queries need to be repeated, some of the previously retrieved rows may not be available any more as they were originally. This does not exclude the possibility that also some new rows would appear into the resultsets of the repeated queries.

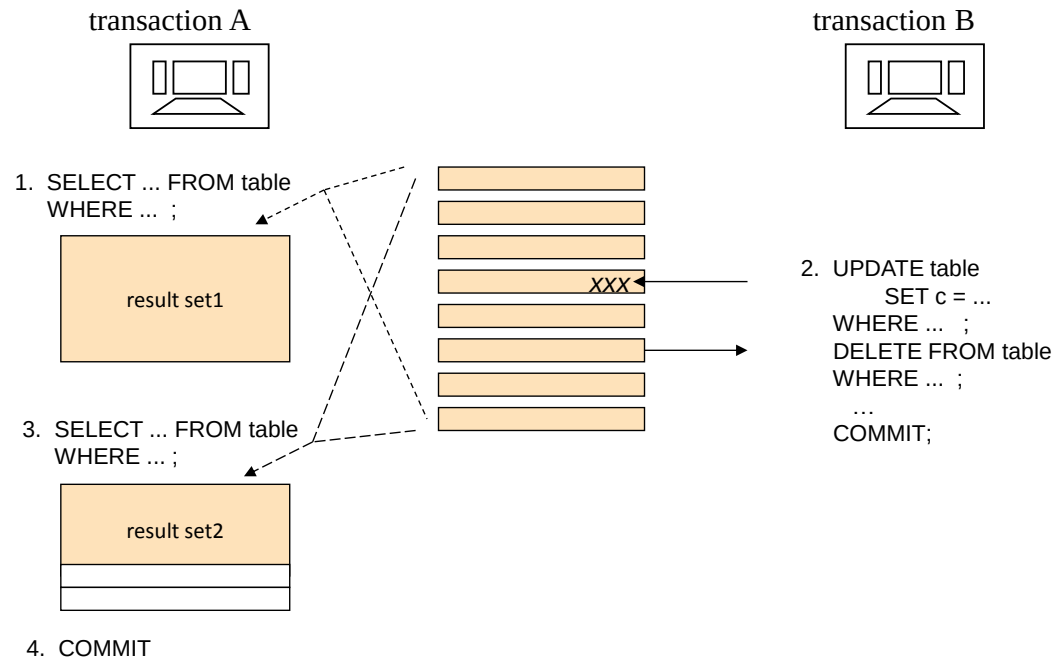


Figure 2.4 Non-repeatable Read problem in transaction A

2.1.4 The Phantom Problem

Phantom anomaly, presented in figure 2.5, means that the result sets of the queries in the transaction may include new rows if some queries will be repeated. These may include newly inserted rows or updated rows where the column values have changed to fulfill the search conditions in the queries.

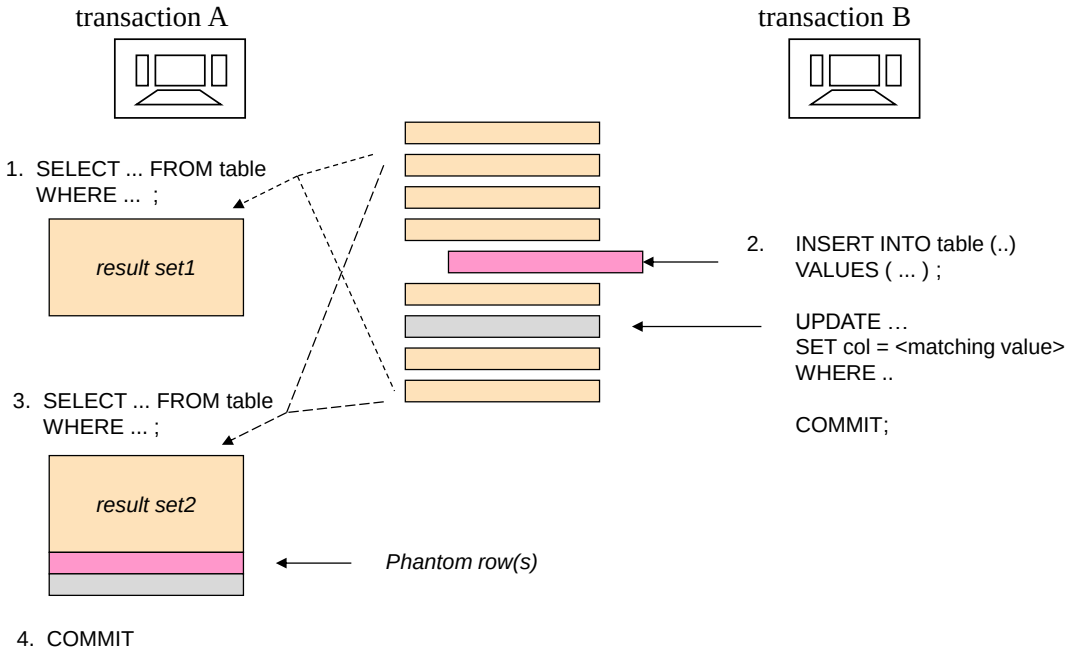


Figure 2.5 Example of a Phantom problem

2.2 The ACID Principle of Ideal Transaction

The ACID principle, presented by Theo Härder and Andreas Reuter in 1983 at ACM Computing Surveys, defines the ideal of reliable SQL transactions in multi-client environment. The ACID acronym comes from initials of the following four transaction properties:

Atomic A transaction needs to be an atomic ("All or nothing") series of operations which either succeeds and will be committed or all its operations will be rolled back from the database.

Consistent The series of operations will transfer the contents of the database from a consistent state to another consistent state. So, at latest, at the commit time, the operations of the transaction have not violated any database constraints (primary keys, unique keys, foreign keys, checks).

Most DBMS products apply the constraints immediately[4] to every operation. Our more restrictive interpretation of consistency requires that the application logic in the transaction has to be correct and properly tested (well-formed transaction) including exception handlings.

Isolated The original definition by Härder and Reuter, "Events within a transaction must be hidden from other transactions running concurrently" will not be fully satisfied by most DBMS products, which we have explained in our "Concurrency Paper", but need to be considered also by the application developers. The current DBMS products use various concurrency control technologies for protecting concurrent transactions from the side effects of others and application developers need to know how to use these services properly.

Durable The committed results in the database will survive on disks in spite of possible system failures.

The ACID principle requires that a transaction which does not fulfill these properties shall not be committed, but either the application or the database server has to rollback such transactions.

2.3 Transaction Isolation Levels

The Isolated property in ACID principle is challenging. Depending on the concurrency control mechanisms, it can lead to concurrency conflicts and too long waiting times, slowing down the production use of the database.

The ISO SQL standard does not define how the concurrency control should be implemented, but based on the concurrency anomalies i.e. bad behaving phenomena which we have illustrated above, it defines the isolation levels which need to solve these anomalies as defined in Table 2.1 and short explanations of the reading restrictions resulting these isolation level settings are listed in Table 2.2. Some of the isolation levels are less restrictive and some are more restrictive giving better isolation, but perhaps at the cost of performance slowdown.

It is worth noticing that isolation levels say nothing about write restrictions. For write operations some lock protection is typically needed, and a successful write is always protected against overwriting by others up to the end of transaction.

[4] Only few DBMS products support the SQL standard option of deferring constraints to be checked at transaction commit phase.

Table 2.1 ISO SQL transaction isolation levels solving concurrency anomalies

Phenomena: / Isolation Level:	Lost Update	Dirty Read	Non-Repeatable Read	Phantoms
READ UNCOMMITTED	NOT possible	possible !	possible !	possible !
READ COMMITTED	NOT possible	NOT possible	possible !	possible !
REPEATABLE READ	NOT possible	NOT possible	NOT possible	possible !
SERIALIZABLE	NOT possible	NOT possible	NOT possible	NOT possible

Table 2.2 Transaction Isolation Levels of ISO SQL (and DB2) explained

ISO SQL Isolation level	**DB2** Isol. level	**Isolation service explained**
Read Uncommitted	UR	allows "dirty" reading of uncommitted data written by concurrent transactions.
Read Committed	CS (CC)	does not allow reading of uncommitted data of concurrent transactions. Oracle and DB2 (9.7 and later) will read the latest committed version of the data (in DB2 this is also called "Currently Committed", CC), while some products will wait until the data gets committed first.
Repeatable Read	RS	allows reading of only committed data, and it is possible to repeat the reading without any UPDATE or DELETE changes made by the concurrent transactions in the set of accessed rows.
Serializable	RR	allows reading of only committed data, and it is possible to repeat the reading without any INSERT, UPDATE, or DELETE changes made by the concurrent transactions in the set of accessed tables.

Note 1: Note the difference between isolation level names in ISO SQL and DB2. DB2 originally defined only 2 isolation levels: CS for Cursor Stability, and RR for Repeatable Read. The names have not been changed, even if the ISO SQL later defined the current 4 isolation levels and gave different semantics for Repeatable Read.

Note 2: In addition to ISO SQL isolation levels, in both Oracle and SQL Server an isolation level called **Snapshot** has been implemented. Here the transaction sees only a snapshot of the committed data as it was in the beginning of the transaction, but does not see any changes made by concurrent transactions. However, Oracle calls this SERIALIZABLE.

We will later discuss which isolation levels are supported in the DBMS products which we are studying and how these are implemented. Depending on the DBMS product, the isolation levels can be defined as the database level default; as the SQL session level default in the beginning of the transaction; or in some products even as statement/table level execution hints. As the best practice and according to ISO SQL, we recommend that isolation level should be configured at the beginning of every transaction and according to the actual isolation need of that transaction. According to ISO SQL standard, Oracle and SQL Server the isolation level will be set by the following command syntax

```
SET TRANSACTION ISOLATION LEVEL <isolation level>
```

but, for example, DB2 uses the following syntax

```
SET CURRENT ISOLATION = <isolation level>
```

In spite of different syntaxes and isolation level names used by the DBMS products, ODBC and JDBC API know only the isolation level names of the ISO SQL, and, for example, in JDBC the isolation level is set as the parameter of connection object's method *setTransactionIsolation* as follows:

```
<connection>.setTransactionIsolation(Connection.<transaction isolation>);
```

where `<transaction isolation>` is defined using reserved words corresponding the isolation level, for example, TRANSACTION_SERIALIZABLE for Serializable isolation. The isolation level defined will be mapped to the corresponding isolation level of the product by the JDBC driver of the DBMS product. If the matching isolation level is missing, then an SQLException will be raised by the JDBC driver.

2.4 Concurrency Control Mechanisms

Modern DBMS products are mainly using the following Concurrency Control (CC) mechanisms for isolation

- Multi-Granular Locking scheme[5] (called **MGL** or just **LSCC**)
- Multi-Versioning Concurrency Control (called **MVCC**)
- Optimistic Concurrency Control (**OCC**).

[5] In literature some authors call locking schemes (LSCC) as "pessimistic concurrency control" (PCC) and MVCC as "optimistic concurrency control" although the real OCC has different concurrency semantics. MVCC is used for reading, but still needs some LSCC to protect writing.

2.4.1 Locking Scheme Concurrency Control (LSCC, MGL)

Table 2.3 defines the basic locking scheme which the ***lock manager*** part of the database server uses automatically to protect the integrity of data in read and write operations. Since only one write operation at a time is possible for a row, lock manager tries to get an exclusive i.e. **X-lock** protection on rows for write operations, such as INSERT, UPDATE, or DELETE. An X-lock will be granted for write operations only when no other transaction has any locks on the same resource(s), as shown in Table 2.3, and when granted the **X-locks will be kept up to the end of the transaction**.

The Lock manager protects the integrity of the read operations, such as SELECT, by shared i.e. **S-locks**, which can be granted to multiple concurrent reading clients since they don't disturb each other, but this depends on the isolation level of the transaction. The **`READ UNCOMMITTED`** isolation level does not require S-lock protection for reading, but in case of other isolation levels, S-lock is needed for reading, and will be granted if no other transaction has an X-lock on the row(s).

Table 2.3 Compatibility of S-locks and X-locks

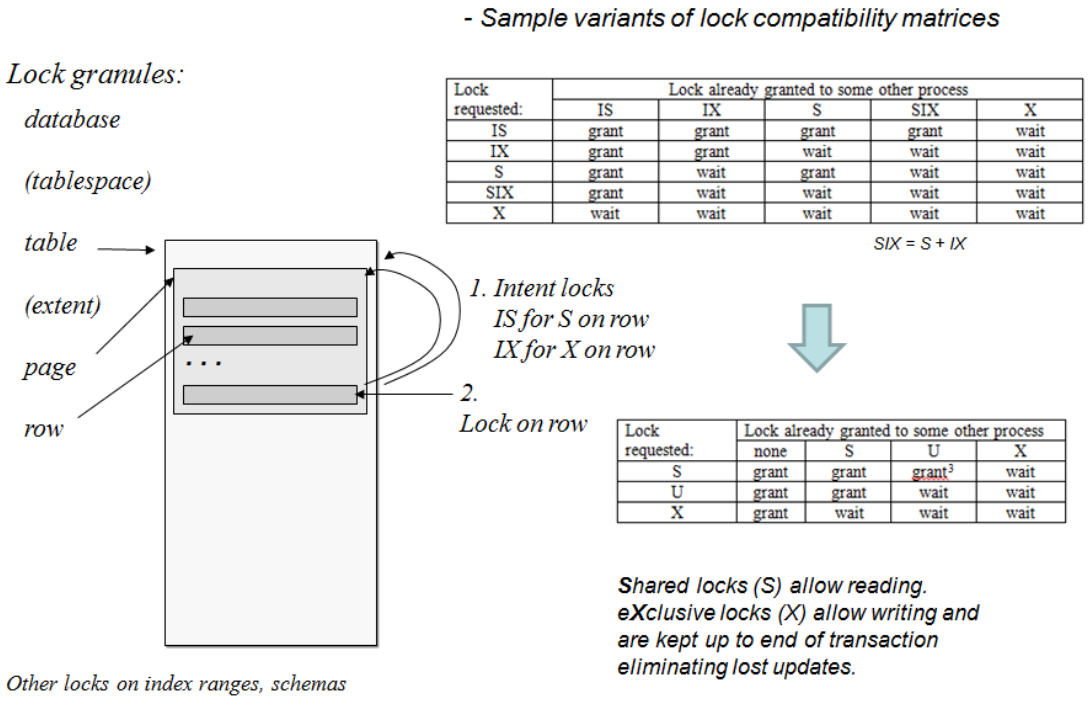
When a transaction needs following lock for a row	When another transaction already has got the following lock for the same row		When no other transaction has any lock on the same row
	S-lock	**X-lock**	
S-lock	lock granted	wait for release	lock granted
X-lock	wait for release	wait for release	lock granted

In case of the **`READ COMMITTED`** isolation, the S-lock on a row will be released right after the row has been read, whereas in **`REPEATABLE READ`** and **`SERIALIZABLE`** isolation levels the S-locks will be kept up to the end of the transaction. All locks of the transaction will be released at the end of the transaction[6] independently on how the transaction has been ended.

> **Note**: In some DBMS products, the SQL dialect includes explicit LOCK TABLE commands, but also these locks are always released implicitly at end of the transaction and the S-lock earlier in case of the READ COMMITTED isolation. No explicit UNLOCK TABLE commands are usually available in SQL dialects, except for example in MySQL/InnoDB.

The real locking schemes of DBMS products are more complicated, using a lock on different granules, such as row, page, table, index range, schema, etc, and, in addition to S-locks and X-locks, some other locking modes. For row-level lock requests, the lock managers generate first the corresponding intent lock requests on larger granule levels, which makes it possible to control the compatibility of the locks on **Multi-Granular Locking** scheme (**MGL**) as described in Figure 2.6.

[6] Except of the current row of a WITH HOLD cursor of some products, such as DB2.



Lock requested:	Lock already granted to some other process				
	IS	IX	S	SIX	X
IS	grant	grant	grant	grant	wait
IX	grant	grant	wait	wait	wait
S	grant	wait	grant	wait	wait
SIX	grant	wait	wait	wait	wait
X	wait	wait	wait	wait	wait

Lock requested:	Lock already granted to some other process			
	none	S	U	X
S	grant	grant	grant[3]	wait
U	grant	grant	wait	wait
X	grant	wait	wait	wait

Figure 2.6 Compatibility control of locks on different granule levels

The locking protocol will sort out the Lost Update Problem, but if the competing transactions are using an isolation level which keeps the S-locks up to end of the transactions, then the locking will lead to another problem as shown in Figure 2.7. Both transactions will be waiting for each other in a "never ending waiting cycle" called **Deadlock**. In early database products this was a crucial problem, but modern DBMS products include a sleeping thread called **Deadlock Detector** which usually wakes up every *2* seconds (sleeping time can be configured) to look for the deadlocks, and after finding one, will select one of the waiting transactions as the deadlock victim and does an automatic rollback to the victim.

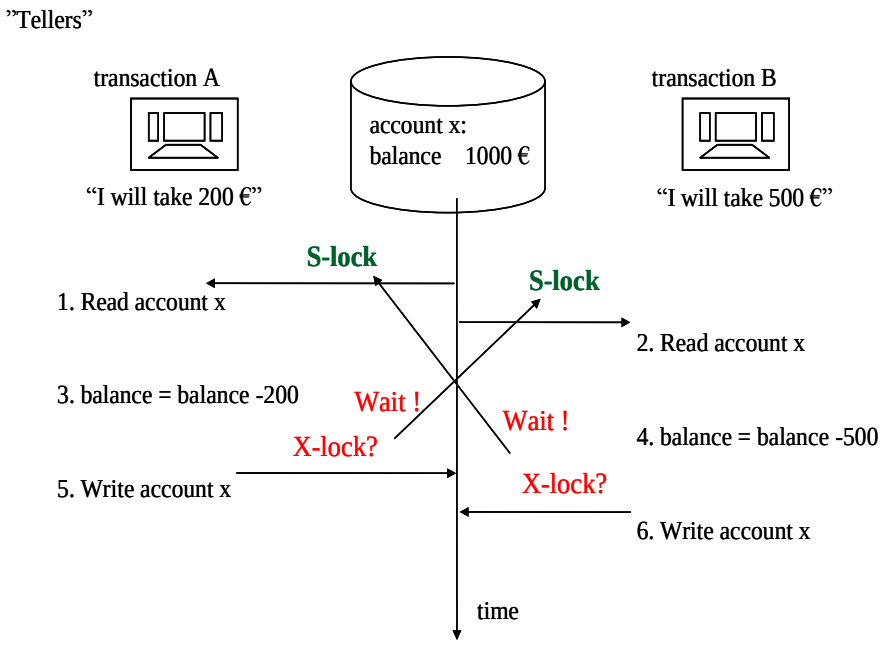


Figure 2.7 Lost update problem solved by LSCC but leading to a deadlock

The application client of the victim will get a deadlock exception message, and should retry the transaction after a random but a short waiting time. See the "**retry wrapper**" part in the Java code for the BankTransfer example in Appendix 2.

> **Note**: It is important to remember that ***no DBMS can automatically restart the aborted victim*** in case of a deadlock, but it is the responsibility of the application code, or the application server in which the data access client component has been installed. It is also important to understand that a deadlock is not an error, and aborting the victim transaction is a service provided by the server, so that the application clients can carry on the production in case those concurrent transactions just cannot both proceed.

2.4.2 Multi-Versioning Concurrency Control (MVCC)

In the MVCC technique, the server maintains a history chain in some timestamp order of the updated versions for all data rows, so that for any updated row, a committed version at the start time of any concurrent transaction can be found. This concurrency control technique eliminates the waiting times in reading and provides just 2 isolation levels: any transaction with the **`READ COMMITTED`** isolation level will get **the latest committed row** versions from the history chains, and transaction with the **`SNAPSHOT`** isolation level will see only **the latest committed versions of the rows at the transaction start time** (or written by the transaction itself). In Snapshot isolation level the transaction never sees the phantom rows and cannot even prevent concurrent transactions on writing phantom rows, whereas `SERIALIZABLE` isolation based on LSCC (MGL) mechanism prevents concurrent transactions from writing of the phantoms rows into the database. In fact, the transaction continues to see in the snapshot the "**ghost rows**", rows which concurrent transactions have meanwhile deleted from the database.

In spite of the isolation level, writing is typically protected also in MVCC systems by some kind of row locking. Updating of rows, contents of which have meanwhile been updated by others, will generate some "Your snapshot is too old" type of error.

MVCC Implementations

As an example of MVCC implementation, Figure 2.8 presents the MVCC mechanism as implementation in Oracle, but can be applied as explanation of MVCC implementations in general. Oracle calls the `SNAPSHOT` isolation as `SERIALIZABLE`. The version store of Oracle is implemented in Undo Tablespace. In SQL Server the TempDB database is used as the version store of the `SNAPSHOT` isolation for all databases of the instance.

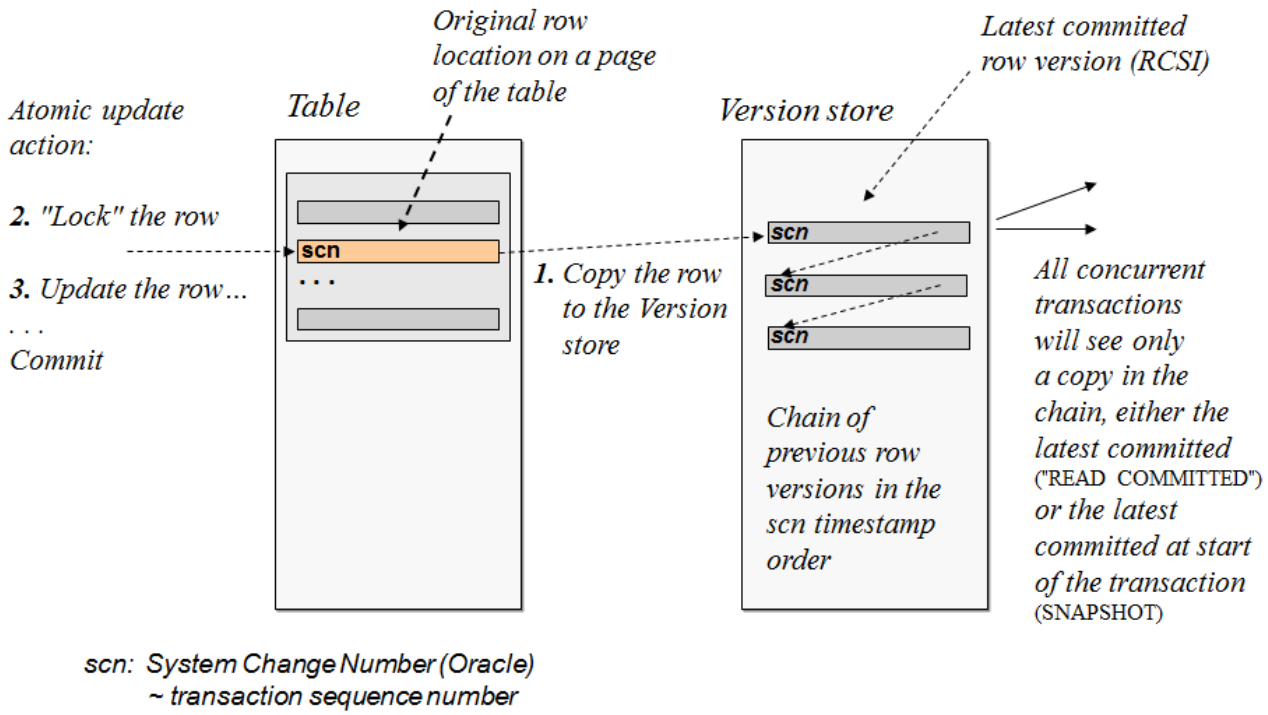


Figure 2.8 Multi-versioning (MVCC) based on history chains of committed rows

In Oracle's MVCC, the first transaction to write a row (that is to insert, update or delete) will get a kind of lock on the row and will win the competition of writing, and the concurrent writers are put to waiting queue. Row locks are implemented by stamping the written rows by System Change Numbers (SCN) of the transaction, sequence numbers of started transactions[7]. As far as the SCN of a row belongs to an active transaction, that row is locked for that transaction. Use of write locks means that deadlocks are possible, but instead of automatically aborting the deadlock victim, Oracle finds immediately the row lock which would lead to a deadlock, and raises exception to the client application and waits for the client to sort out the deadlock by an explicit `ROLLBACK` command.

Concurrency control technique in **Oracle** can be called as hybrid CC since in addition to MVCC with implicit row locking, Oracle provides explicit "`LOCK TABLE`" commands and also explicit locking of rows by command "SELECT ... FOR UPDATE" which provides means for preventing even the invisible Phantom rows. In Oracle, a transaction can also be declared as a **Read Only** transaction.

Microsoft, too, has noticed the benefits of MVCC, and since the **SQL Server** version 2005, it has been possible to configure SQL Server to use row versioning by the following settings of database properties by Transact-SQL commands, and since version 2012 by database properties as presented in Figure 2.9.

[7] transaction sequence number XSN in the SQL Server MVCC implementation (Delaney 2012)

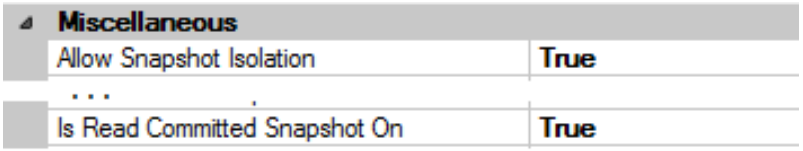


Figure 2.9 Configuring SQL Server 2012 to support Snapshots

Concurrency control mechanism of **MySQL/InnoDB** is a real hybrid CC providing the following four isolation levels for reading:

- `READ UNCOMMITTED` reading the table rows without S-locking,
- `READ COMMITTED` (actually "Read Latest Committed") reading the table rows and when these are locked using MVCC,
- `REPEATABLE READ` (actually "Snapshot") using MVCC
- `SERIALIZABLE` using MGL CC with S-locks, preventing Phantom rows.

Note: independently of the isolation level, writing will always need protection by exclusive locking.

2.4.3 Optimistic Concurrency Control (OCC)

In the original OCC, all the changes made by the transaction are kept apart from the database and synchronized into the database only at the COMMIT phase. This has been implemented, for example, in the Pyrrho DBMS of the University of the West of Scotland. The only and implicit isolation level available in Pyrrho DBMS is SERIALIZABLE (http://**www.pyrrhodb.com**).

2.4.4 Summary

The ISO SQL standard is developed by ANSI and is originally based on a DB2's SQL language version, which IBM donated to ANSI. Concurrency control mechanism in DB2 is based on Multi-Granular Locking (MGL) scheme and those days DB2 had only 2 isolation levels: Cursor Stability (CS) and Repeatable Read (RR). Obviously this have had influence on semantics of the new isolation levels (see Table 2.2 above) defined for the ANSI/ISO SQL, which can be understood in terms of locking mechanisms.

The SQL standard does not say anything about implementations, so also other concurrency mechanisms have been applied and same isolation names have been used with slightly different interpretations, as we have seen above. Under the title "Isolation Levels" in table 2.4 the isolation levels (in blue background) Read Uncommitted, Read Committed, Repeatable Read and Serializable represent the isolation levels as we understand the SQL standard semantics. The same names in quotes on columns of the products, for example "serializable", indicate different semantics than the isolation levels of the standard. The isolation level name "read latest committed" is invented by us, since various names are used by products for the isolation level in which read operation occurs without need for lock waiting: either the current row or the latest committed version of the row, in case the row has been updated but not yet committed by some concurrent transaction. These snapshot based semantics have raised confusion, and isolation concepts used in the standard might need clarifications and extensions. A group of developers of the ANSI/SQL standard and database textbook writers confess the problem in their article "A Critique of ANSI SQL Isolation Levels" (Berenson et al, 1995).

Applying the Exercise 2.7 of the next hands-on practice using different DBMS products shows some problems of snapshots in case of writing to database, since the writing will violate the snapshot consistency. A safe use of snapshot is therefore only for reporting purposes. Table 2.4 summaries also some other differences between the DBMS products.

Table 2.4 Supported transaction functionalities in the ISO SQL standard and DBMS products

	ANSI/ISO SQL	DB2	Oracle	SQL SERVER	MySQL/InnoDB	PostgreSQL	Pyrrho
	SQL:2006	LUW 9.7	12g1	2012	5.6	9.2	4.8
autocommit (server-side)	n/a	n/a	n/a	yes	yes	yes	yes
Transaction Limits							
explicit start	yes	n/a	n/a	yes	yes	yes	yes
implicit start	yes	yes	yes	(configurable)	(configurable)	n/a	n/a
COMMIT	yes	yes	yes	yes	yes	yes	yes
implicit commit on DDL	n/a	n/a	yes	n/a	yes	n/a	n/a
ROLLBACK	yes	yes	yes	yes	yes	yes	yes
implicit rollback on concurrency conflict (deadlock)	yes	yes	no (exception raised)	yes	yes	no (transaction invalidated)	yes, at commit
implicit rollback on error	implementation dependent	n/a	n/a	(configurable)	n/a	no (transaction invalidated)	yes
SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
ROLLBACK TO SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
RELEASE SAVEPOINT	yes	yes	yes	n/a	yes	yes	n/a
Isolation levels							
READ UNCOMMITTED	yes	UR	n/a	yes	yes	n/a (1)	n/a
"read latest committed"	n/a	CS (currently committed)	"read committed"	(configurable)	"read committed"	"read committed"	n/a
READ COMMITTED	yes	CS	n/a	yes	n/a	n/a (2)	n/a
REPEATABLE READ	yes	RS	n/a	yes	n/a	n/a (2)	n/a
snapshot		n/a	"serializable"	(configurable)	"repeatable read"	"serializable"	"serializable"
SERIALIZABLE	yes	RR	explicit locking	yes	yes	explicit locking	"serializable"
note: isolation levels in upper-case stand for ISO/SQL semantics						(1) migrate to "read latest committed"	
						(2) migrate to snapshot	

Snapshot means a consistent "view" of the database at the beginning of the transaction. As such it suits perfectly to *read-only transactions*, since in that case the transaction does not block concurrent transactions.

The Snapshot semantics do not eliminate existence of phantoms, as these are just not included in the snapshot. In products using MVCC the ISO SQL Serializable semantics preventing existence of phantoms can be implemented by using properly covering table-level locks, for example in Oracle and PostgreSQL. In case of snapshot isolation all products allow INSERT commands, but services on other write operations seem to vary.

From the table 2.4 we can see that DB2 is the only DBMS product in our DebianDB that does not support snapshot isolation.

2.5 Hands-on laboratory practice

A new MySQL user session gets initiated, in the usual way (Figure 2.10):

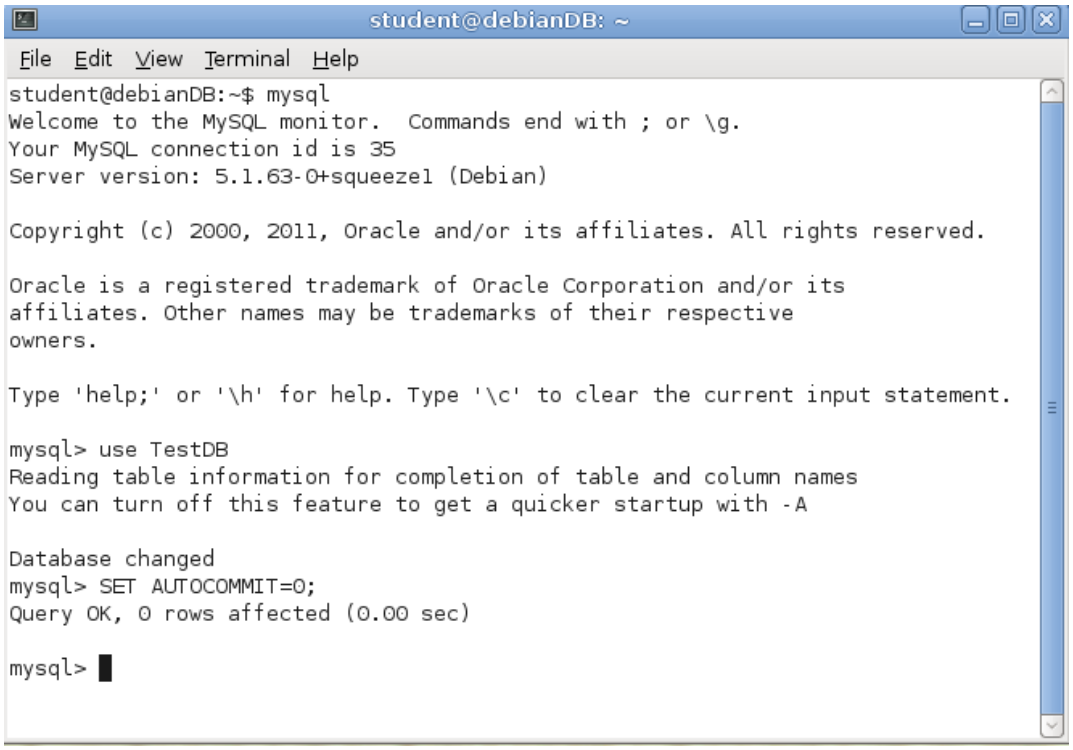


Figure 2.10 MySQL user session initiation

EXERCISE 2.0

For concurrency tests we need a second terminal window and a new MySQL session as presented in Figure 2.10. Let the left hand side terminal session correspond to Session A, and the right hand side one be Session B. Both terminal sessions begin by connecting to database TestDB, and by turning off the AUTOCOMMIT mode:

```
-----------------------------------
use TestDB
SET AUTOCOMMIT = 0;
-----------------------------------
```

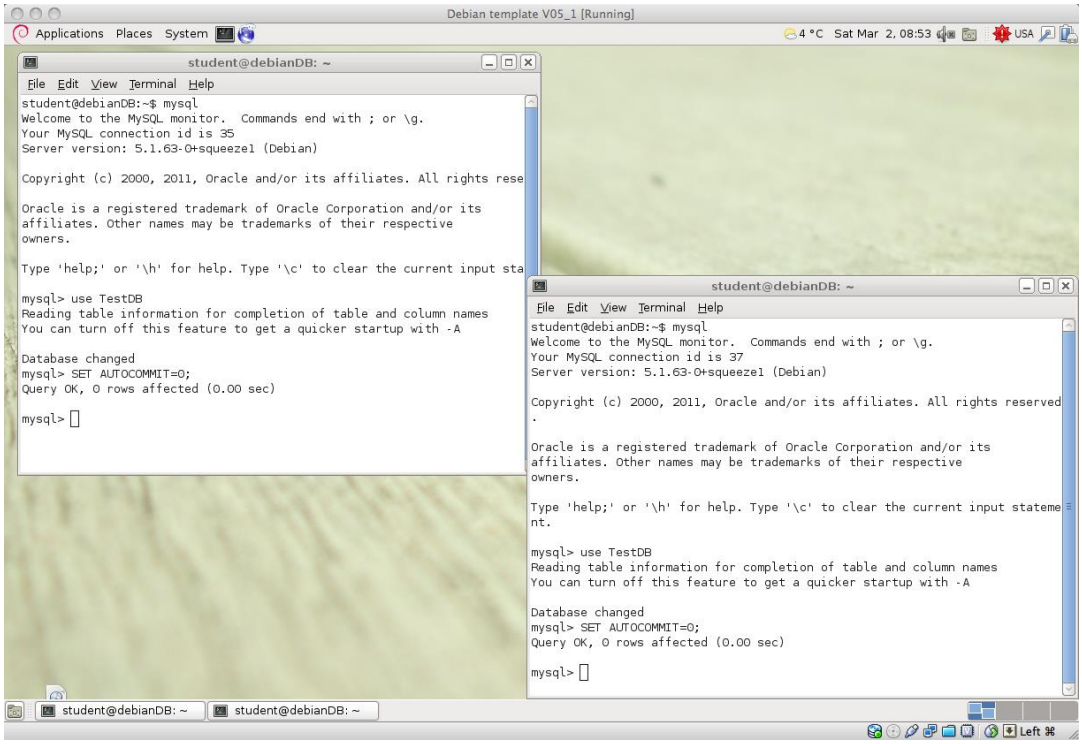


Figure 2.11 Concurrent MySQL sessions in the DebianDB virtual machine

Concurrent transactions accessing the same data may block others, as we will see. Therefore, transactions should be designed to be as short as possible, just to do the necessary work. The inclusion of a dialogue with the end-user in the SQL transaction logic would lead to catastrophic waiting times in the production environment. Therefore, **it is necessary that no SQL transactions will give control to user interface level before the transaction has ended.**

The current isolation level settings are inspected using the system variables in SELECT command as follows:

```
-----------------------------------
SELECT @@GLOBAL.tx_isolation, @@tx_isolation;
-----------------------------------
```

It is seen that MySQL/InnoDB defaults to the REPEATABLE READ isolation level, at the global as well as at the local (session) level.

To be safe, the Accounts table is dropped and it is re-created/initialized to register two rows of data:

```
-----------------------------------
DROP TABLE Accounts;
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT remains_nonnegative CHECK (balance >= 0)
);
SET AUTOCOMMIT = 0;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

According to ISO SQL the transaction isolation level should be set at the beginning of the transaction, and it cannot be changed later in the transaction. Since implementations vary, we will verify in the following when the transaction isolation level can be set in MySQL/InnoDB in case of explicit transactions:

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

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

According to ISO SQL 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;
```

Questions

- Conclusions reached?

EXERCISE 2.1

As explained earlier, the **lost update problem** involves overwriting of the just updated value of a row by another (concurrently running) transaction, BEFORE the termination of the given transaction. 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 without first reading the current committed value. We call the case "**Blind Overwriting**" or "**Dirty Write**", and it is even too easy to produce.

The blind overwriting situation occurs, for example, when application program reads values from the database, updates the values in memory, and then writes the updated value back to database. In the following table we simulate the application program part using local variables in MySQL. A local variable is just a named placeholder in the current session's working memory. It is identified by the at symbol (@) appended to the beginning of its name, and it may be embedded in the syntax of SQL statements, provided that it is made sure that it always receives a single scalar value.

Table 2.4 suggests an experiment with concurrent sessions (A and B, presented in columns of their own). Ordering of the SQL statements to be executed in this exercise is marked in the first column. The aim is to simulate the lost update situation illustrated in Figure 2.2: the transaction in Session A is to withdraw €200 from account 101, and the transaction in Session B is to withdraw €500 from the same account and it will OVERWRITE the account's balance value (losing the information about the €200 withdrawal by transaction A). So the result of Blind Overwriting is the same as if it were result of a lost update problem.

To keep the database content independent from previous experiments, we always first restore the original content of the Accounts table as follows:

```
-----------------------------------
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;
-----------------------------------
```

Table 2.4 Blind Overwriting problem, application simulated by use of local variables

	Session A	Session B

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL READ COMMITTED;`
	
	`-- Amount to be transfered by A`
	`SET @amountA = 200;`
	`SET @balanceA = 0;  -- Init value`
	
	`SELECT balance INTO @balanceA`
	`FROM Accounts WHERE acctID = 101;`
	
	`SET @balanceA = @balanceA - @amountA;`
	`SELECT @balanceA;`	

2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** -- Amount to be transfered by B SET @amountB = 500; SET @balanceB = 0; -- Init value SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101; SET @balanceB = @balanceB - @amountB;
3	UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;	
4		UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Note: When in step number 4, one should keep in mind that MySQL's default lock timeout is 90 seconds. So, transaction (client) A should proceed to step with number 5 without delay, following B's step number 4.

Considering the concurrent (interleaved, to be more exact) execution of transactions A and B in Table 2.4, A gets ready to withdraw €200 from the bank account in step 1 (not having updated the database, yet), B gets ready to withdraw €500 in step 2, A updates the database in step 3, B updates the database in step 4, A checks the bank account's balance to make sure it is as expected before it commits in step number 5, and B does the same thing in step number 6.

Questions

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

Note: All DBMS products implement concurrency control so that in all isolation levels the lost update anomaly never shows up. Still, there is always the possibility of carelessly written application code cases where 'blind overwrite' operations effectively produce the same catastrophic effects with the update anomaly. In practice, this is like having the lost update anomaly creep into concurrently running transactions scenarios from the back door!

EXERCISE 2.2a

Repeating exercise 2.1, but now using isolation level SERIALIZABLE for MGL.

First the original content of the Accounts table is restored:

```
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;
-- --------------------------------
```

Table 2.5a Scenario of 2.1 keeping the S-locks

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL SERIALIZABLE;`
	
	`-- Amount to be transfered by A`
	`SET @amountA = 200;`
	`SET @balanceA = 0;  -- Init value`
	
	`SELECT balance INTO @balanceA`
	`FROM Accounts WHERE acctID = 101;`
	
	`SET @balanceA = @balanceA - @amountA;`
	
	`SELECT @balanceA;`	

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

3	`UPDATE Accounts`
	`SET balance = @balanceA`
	`WHERE acctID = 101;`	

4		`-- continue without waiting for A!`
		`UPDATE Accounts`
		`SET balance = @balanceB`
		`WHERE acctID = 101;`

5	`SELECT acctID, balance`
	`FROM Accounts WHERE acctID = 101;`
	
	`COMMIT;`	

6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Questions

a) Conclusion(s) reached?
b) What if 'SERIALIZABLE' is replaced by 'REPEATABLE READ' in both transactions?

Note: MySQL/InnoDB implements REPEATABLE READ using MVCC for reading, and SERIALIZABLE using MGL locking scheme

EXERCISE 2.2b

Repeating exercise 2.2a, but now using "sensitive updates" in SELECT – UPDATE scenarios without local variables.

First the original content of the Accounts table is restored:

```
-----------------------------------
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;
-----------------------------------
```

Table 2.5b SELECT – UPDATE competition using sensitive updates

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL SERIALIZABLE;`
	
	`SELECT balance FROM Accounts`
	`WHERE acctID = 101;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`  ISOLATION LEVEL SERIALIZABLE;`
		
		`SELECT balance FROM Accounts`
		`WHERE acctID = 101;`

3	`UPDATE Accounts`
	`SET balance = balance - 200`
	`WHERE acctID = 101;`	

4		`UPDATE Accounts`
		`SET balance = balance - 500`
		`WHERE acctID = 101;`

5	`SELECT acctID, balance`
	`FROM Accounts`
	`WHERE acctID = 101;`
	
	`COMMIT;`	

6		`SELECT acctID, balance`
		`FROM Accounts`
		`WHERE acctID = 101;`
		
		`COMMIT;`

Question

- Conclusion(s) reached?

EXERCISE 2.3 UPDATE – UPDATE competition on two resources in different order

First the original content of the Accounts table is restored:

```
----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
----------------------------------
```

Table 2.6 UPDATE-UPDATE Scenario

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`ISOLATION LEVEL READ COMMITTED;`
	
	`UPDATE Accounts`
	`SET balance = balance - 100`
	`WHERE acctID = 101;`	

2		`SET AUTOCOMMIT = 0;`
		`SET TRANSACTION`
		`ISOLATION LEVEL READ COMMITTED;`
		
		`UPDATE Accounts`
		`SET balance = balance - 200`
		`WHERE acctID = 202;`

3	`UPDATE Accounts`
	`SET balance = balance + 100`
	`WHERE acctID = 202;`	

4		`UPDATE Accounts`
		`SET balance = balance + 200`
		`WHERE acctID = 101;`

5	**`COMMIT;`**	
6		**`COMMIT;`**

Question

- Conclusion(s) reached?

Note: Transaction Isolation Level has no role in this scenario, but it is a good practice to always define isolation level in the beginning of every transaction! There can be some hidden processing, for example by foreign key checking and triggers using read access. Design of triggers should actually be responsibility of database administrators, and it is not in the scope of this tutorial.

EXERCISE 2.4 (Dirty Read)

To continue with the transaction anomalies, an attempt is now made to produce the occurrence of a dirty read situation. Transaction A runs in (MySQL's default) REPEATABLE READ, whereas transaction B is set to run in READ UNCOMMITTED isolation level:

First the original content of the Accounts table is restored:

```
-----------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Table 2.7 Dirty read problem

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ UNCOMMITTED;** SELECT * FROM Accounts; **COMMIT;**
3	**ROLLBACK;** SELECT * FROM Accounts; **COMMIT;**	

Questions

a) Conclusion(s) reached? What can we say of the reliability of transaction B?

b) How can we solve the problem?

EXERCISE 2.5 Non-Repeatable Read

Next comes the non-repeatable read anomaly:

First the original content of the Accounts table is restored:

```
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;
```

Table 2.8 A non-repeatable read problem

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL READ COMMITTED;`
	
	`SELECT * FROM Accounts`
	`WHERE balance > 500;`	

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

3	`-- Repeating the same query`
	`SELECT * FROM Accounts`
	`WHERE balance > 500;`
	
	`COMMIT;`	

Questions

a) Does transaction A read in step 3 the same result it has read in step number 1?

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

EXERCISE 2.6

An attempt is now made to produce the classical '**insert phantom**' of textbooks:

First the original content of the Accounts table is restored:

```
-----------------------------------
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;
-----------------------------------
```

Table 2.9 Insert phantom problem

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL REPEATABLE READ;`
	`START TRANSACTION READ ONLY;`	

2		`SET AUTOCOMMIT = 0;`
		`INSERT INTO Accounts (acctID,`
		`balance) VALUES (301,3000);`
		`COMMIT;`

3	`SELECT * FROM Accounts` `WHERE balance > 1000;`	

		`INSERT INTO Accounts (acctID,`
		`balance) VALUES (302,3000);`
		`COMMIT;`

3	`-- Can we see accounts 301 and 302?`
	`SELECT * FROM Accounts`
	`WHERE balance > 1000;`
	
	`COMMIT;`	

Questions

a) Does the transaction B need to wait for transaction A at any step?

b) Are the *newly inserted* accounts 301 and/or 302 visible in transaction A's environment?

c) Does it affect to the resultset of step 4 if we change the order of steps 2 and 3?

d) *(advanced level question)*
MySQL/InnoDB uses Multi-Versioning for REPEATABLE READ isolation, but what is the proper timestamp of the snapshot i.e. is it the time of the START TRANSACTION or the first SQL command? Note that even in transactional mode we can use the START TRANSACTION command (and set some properties of the transaction, in this example declaring the transaction as READ ONLY transaction).

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

Let's setup the starting a new terminal window for "Client C" (which we will use also later in our test)

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;
```

Table 2.10 Insert and Update phantoms problems, and updating a deleted "ghost row "

	Session A	**Session B**

1	`SET AUTOCOMMIT = 0;`
	`SET TRANSACTION`
	`  ISOLATION LEVEL REPEATABLE READ;`
	
	`SELECT * FROM T WHERE i = 1;`	

2		`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;`

3	`-- Repeat the query and do 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 the snapshot'** WHERE id = 3; UPDATE T SET s = **'update by A outside the snapshot'** WHERE id = 4; UPDATE T SET s = **'update by A after update by B'** WHERE id = 1;	
3.5		**-- ClientC:** **-- what's the current content?** SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SELECT * FROM T;
4		**-- Client B continues** **-- without waiting for A** **COMMIT;** **SELECT * FROM T;**
5	SELECT * FROM T WHERE i = 1; UPDATE T SET s = **'updated after delete?'** WHERE id = 5; SELECT * FROM T WHERE i = 1; **COMMIT;**	
6	SELECT * FROM T; **COMMIT;**	
7		-- Client C does the final select SELECT * FROM T; COMMIT;

Questions

a) Are the insert and update made by transaction B visible in transaction A's environment?

b) What happens when A tries to update the row 1 updated by transaction B?

c) What happens when A tries to update the row with id=5 deleted by transaction B?

Note: A SELECT operation in a MySQL/InnoDB transaction which runs using REPEATABLE READ isolation level creates a consistent snapshot. If the transaction manipulates rows in the base table(s) of the snapshot, then the snapshot is not consistent any more.

Listing 2.1 Sample results of a test run of exercise 2.7

```
mysql> -- 5. Client A continues
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)

mysql> UPDATE T SET s = 'updated after delete?' WHERE id = 5;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
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)
mysql> COMMIT;
Query OK, 0 rows affected (0.03 sec)

mysql> -- 6. Client A continues with a new transaction
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)

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

Note: At the end of Appendix 1 we have results of exercise 2.7 from SQL Server 2012 for comparison.

3 Some Best Practices

A user transaction typically needs multiple dialogues with the database. Some of these dialogues will only collect data from the database supporting the user transaction, and as a final step of user transaction, some "save" button will trigger an SQL transaction which will update the database.

SQL transactions, even in the same user transaction sequence, may have different reliability and isolation requirements. Always define transaction isolation level in the beginning of every transaction.

According to the SQL standard READ UNCOMMITTED isolation level can only be used in READ ONLY transactions (Melton and Simon 2002), but the products don't force this.

DBMS products differ from each other in terms of concurrency control services and transaction managing behavior, so it is important for reliability and performance that the application developer knows the behavior of the DBMS to be used.

Reliability is the number 1 priority, before performance etc. but the default transaction isolation level used by DBMS products often favors performance before reliability! Proper isolation level should be planned with extra care, and the SERIALIZABLE isolation level with the ISO SQL semantics should be used, if the developer cannot decide which isolation level provides reliable enough isolation. It is important to understand that the SNAPSHOT isolation guarantees only consistent result sets, but does not preserve the database contents. If you cannot allow phantoms and isolation levels in your DBMS support only snapshots, you need to study possibilities of explicit locking.

SQL transactions should not contain any dialogue with the end user, as this would slow down the processing. Since SQL transactions may get rolled back during the transaction, they should not affect anything else but the database. The SQL transaction should be **as short as possible**, to minimize the concurrency competition, and blocking the concurrent transactions.

Avoid DDL commands in transactions. Implicit commits due to DDL may result unintentional transactions.

Every SQL transaction should have a well-defined task, starting and ending in the same application component. In this tutorial, we will not cover the topic of using SQL transactions in **stored routines**, since the stored procedure languages and implementations are different in different DBMS products. Some of the DBMS products in our DebianDB lab don't allow COMMIT statements in stored routines. However, a transaction may get forced to rollback even in the middle of some stored routine, and this has to be handled also in the calling application code.

The technical context of an SQL transaction is a single database connection. If a transaction fails due to a concurrency conflict, it should, in many cases, have a **retry wrapper** in the application code with a limit of some 10 retries. However, if the **transaction is dependent on** the database content retrieved in **some previous SQL transaction** of the same user transaction, and some concurrent transactions have changed that content in the database and the current transaction

therefore fails to update the content, then this transaction should not have a retry wrapper, but the control should be returned to the user for possible restart of the whole user transaction. We will cover this in our "RVV Paper".

A new database connection needs to be opened for the retry of the SQL transaction in case the connection was lost due to some network problems.

Further Readings, References, and Downloads

Further Readings and References

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. & O'Neil, P., "A Critique of ANSI SQL Isolation Levels", Technical Report MSR-TR-95-51, Microsoft Research, 1995.
Available freely on multiple websites.

Delaney, K., “SQL Server Concurrency – Locking, Blocking and Row Versioning”, Simple Talk Publishing, July 2012

Melton, J. & Simon, A. R., "SQL:1999: Understanding Relational Language components", Morgan Kaufmann, 2002.

Data Management: Structured Query Language (SQL) Version 2, The Open Group, 1996. Available at **http://www.opengroup.org/onlinepubs/9695959099/toc.pdf**

Selected DBTechNet Papers

For more information on concurrency control technologies of DB2, Oracle and SQL Server see the "Concurrency Paper" at

http://www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf

For more information on SQL transactions as part of a user transaction and applying of row version verification (RVV aka ”optimistic locking”) in various data access technologies see the "RVV Paper" at

http://www.dbtechnet.org/papers/RVV_Paper.pdf

Virtual Database Laboratory Downloads

1. The OVA file of our virtual Database Laboratory "DebianDB"

   http://www.dbtechnet.org/download/DebianDBVM06.zip
   (4.8 GB, including MySQL 5.6, DB2 Express-C 9.7, Oracle XE 10.1, PostgreSQL 8.4, Pyrrho 4.8, and hands-on scripts)

2. "Quick Start Guide" for accessing the DBMS products installed in the Lab

   http://www.dbtechnet.org/download/QuickStartToDebianDB.pdf

3. Scripts for experiments of Appendix 1 applied to DB2, Oracle, MySQL, and PostgreSQL

   http://www.dbtechnet.org/download/SQL_Transactions_Appendix1.zip

Appendix 1 Experimenting with SQL Server Transactions

Welcome to a "mystery tour" in the world of SQL transactions using your favorite DBMS product, to experiment and verify yourself what has been presented in this tutorial. The DBMS products behave slightly differently on transaction services, which may surprise you - and your customers - if you are not aware of these differences when developing applications. If you have time to look at more than one product, you will get a broader vision on transaction services provided by the DBMS products on the market.

> **Note**: The scripts tailored for the DBMS products available in our database laboratory DebianDB can be found at
>
> **http://www.dbtechnet.org/download/SQL_Transactions_Appendix1.zip**

In this appendix we present our experiment series applied to SQL Server Express 2012. Since SQL Server runs only on Windows platforms, and is not available in DebianDB, we present also most of results of the exercises, so that you can compare the results of your own experiments. If you want to verify our results, you can download SQL Server Express 2012 for free from the website of Microsoft to your Windows workstation (to Windows 7 or later).

In the following experiments, we use SQL Server Management Studio (SSMS) program. First, we will create a new database "TestDB" as follows using defaults for all configuration parameters, and by USE command start using it as the current database context in our SQL-session.

```
-----------------------------------
CREATE DATABASE TestDB;
USE TestDB;
-----------------------------------
```

PART 1. EXPERIMENTING WITH SINGLE TRANSACTIONS

As default, SQL Server sessions operate in AUTOCOMMIT mode, and using explicit transactions, we can build transactions of multiple SQL commands. However, the whole server instance can be configured to use implicit transactions. Also a single SQL session can be configured to use implicit transactions using the following SQL Server's command

```
SET IMPLICIT_TRANSACTIONS ON;
```

which will be in force until the end of the session, and it can be turned off by the following command:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Let's start experimenting with single transactions. For the beginning we will display some of the essential results too:

```
-----------------------------------------------------------------
-- Exercise 1.1
-----------------------------------------------------------------
--  Autocommit mode
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si SMALLINT);
Command(s) completed successfully.

INSERT INTO T (id, s) VALUES (1, 'first');
(1 row(s) affected)

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)

ROLLBACK; -- What happens?
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)


BEGIN TRANSACTION;  -- An explicit transaction begins

INSERT INTO T (id, s) VALUES (2, 'second');

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           second                         NULL
(2 row(s) affected)

ROLLBACK;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)
```

At the transaction the ROLLBACK command has worked, but now we are back at AUTOCOMMIT mode!

```
-----------------------------------------------------------------
-- Exercise 1.2
-----------------------------------------------------------------
INSERT INTO T (id, s) VALUES (3, 'third');
(1 row(s) affected)

ROLLBACK;
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
3           third                          NULL
(2 row(s) affected)

COMMIT;
Msg 3902, Level 16, State 1, Line 2
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.



-----------------------------------------------------------------
-- Exercise 1.3
-----------------------------------------------------------------
BEGIN TRANSACTION;

DELETE FROM T WHERE id > 1;
(1 row(s) affected)

COMMIT;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL

(1 row(s) affected)
```

```
-----------------------------------------------------------------
-- Exercise 1.4
-- 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 transaction!
-----------------------------------------------------------------
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO T (id, s) VALUES (2, 'will this be committed?');
CREATE TABLE T2 (id INT);  -- testing use of a DDL command in atransaction!
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;
GO  -- GO marks the end of a batch of SQL commands to be sent to the server
(1 row(s) affected)

(1 row(s) affected)
id
-----------
1
(1 row(s) affected)

SELECT * FROM T;    -- What has happened to T ?
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)

SELECT * FROM T2;   -- What has happened to T2 ?
Msg 208, Level 16, State 1, Line 2
Invalid object name 'T2'.



-----------------------------------------------------------------
-- Exercise 1.5a
-----------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
-----------------------------------------------------------------
-- Testing if an error would lead to automatic rollback in SQL Server.
--   @@ERROR is the SQLCode indicator in Transact-SQL, and
--   @@ROWCOUNT is the count indicator of the effected rows
-----------------------------------------------------------------
INSERT INTO T (id, s) VALUES (2, 'The test starts by this');
(1 row(s) affected)

SELECT 1/0 AS dummy;   -- Division by zero should fail!
dummy
-----------
Msg 8134, Level 16, State 1, Line 1
Divide by zero error encountered.

SELECT @@ERROR AS 'sqlcode'
sqlcode
-----------
8134
```

```
(1 row(s) affected)

UPDATE T SET s = 'foo' WHERE id = 9999;  -- Updating a non-existing row
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Updated'
Updated
-----------
0
(1 row(s) affected)

DELETE FROM T WHERE id = 7777;   -- Deleting a non-existing row
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Deleted'
Deleted
-----------
0
(1 row(s) affected)

COMMIT;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)

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);
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
GO

Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__T__3213E83FD0A494FC'. Cannot insert
duplicate key in object 'dbo.T'. The duplicate key value is (2).
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 2
String or binary data would be truncated.
The statement has been terminated.
Msg 220, Level 16, State 1, Line 3
Arithmetic overflow error for data type smallint, value = 32769.
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 4
String or binary data would be truncated.
The statement has been terminated.
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)
```

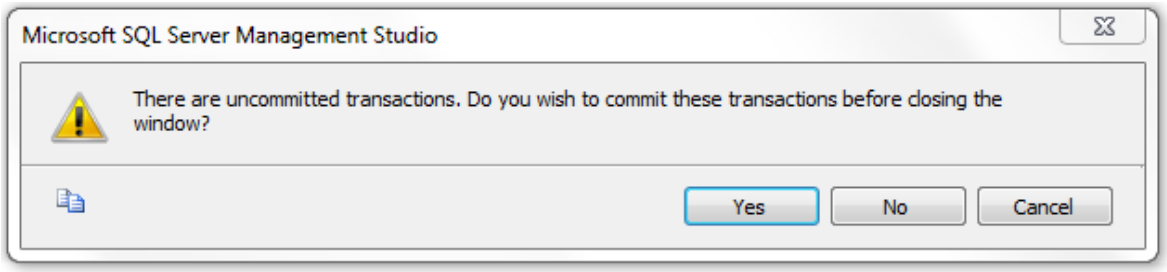
```
BEGIN TRANSACTION;
SELECT * FROM T;
DELETE FROM T WHERE id > 1;
COMMIT;



-----------------------------------------------------------------
-- Exercise 1.5b
-- This is special to SQL Server only!
-----------------------------------------------------------------
SET XACT_ABORT ON;  -- In this mode an error generates automatic rollback
SET IMPLICIT_TRANSACTIONS ON;

SELECT 1/0 AS dummy;    -- Division by zero

INSERT INTO T (id, s) VALUES (6, 'insert after arithm. error');
COMMIT;

SELECT @@TRANCOUNT AS 'do we have an transaction?'
GO

dummy
-----------
Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.

SET XACT_ABORT OFF; -- In this mode an error does not generate automatic rollback

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)

-- What happened to the transaction?
```

```
-----------------------------------------------------------------
-- Exercise 1.6   Experimenting with Transaction Logic
-----------------------------------------------------------------
SET NOCOUNT ON;  -- Skipping the "n row(s) affected" messages
DROP TABLE Accounts;
SET IMPLICIT_TRANSACTIONS ON;

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

COMMIT;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

COMMIT;

-- 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;
acctID      balance
----------- -----------
101         900
202         2100

ROLLBACK;

-- Let's test if the CHECK constraint actually works:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 2
The UPDATE statement conflicted with the CHECK constraint
"unloanable_account". The conflict occurred in database "TestDB", table
"dbo.Accounts", column 'balance'.
The statement has been terminated.

UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         4000

ROLLBACK;
```

```
-- Transaction logic using the IF structure of Transact-SQL
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 4
The UPDATE statement conflicted with the CHECK constraint
"unloanable_account". The conflict occurred in database "TestDB", table
"dbo.Accounts", column 'balance'.
The statement has been terminated.

IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
COMMIT;

-- How about using a non-existent bank account?
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;

SELECT * FROM Accounts ;
acctID      balance
----------- -----------
101         500
202         2000
ROLLBACK;

-- Fixing the case using the IF structure of Transact-SQL
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
```

```
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
IF @@error <> 0 OR @@rowcount = 0
   ROLLBACK
ELSE BEGIN
   UPDATE Accounts SET balance = balance + 500 WHERE acctID = 707;
   IF @@error <> 0 OR @@rowcount = 0
      ROLLBACK
   ELSE
      COMMIT;
   END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000


-----------------------------------------------------------------------
-- Exercise 1.7   Testing the database recovery
-----------------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;

BEGIN TRANSACTION;
INSERT INTO T (id, s) VALUES (9, 'What happens if ..');

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
9           What happens if ..             NULL
```

On **exiting now the SQL Server Management Studio**, we will get following message



and for purposes of our experiment we select "No".

On **restarting the Management Studio** and connecting to our TestDB we can study what happened to our latest uncommitted transaction just by listing the contents of the table T.

```
SET NOCOUNT ON;
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
```

PART 2 EXPERIMENTING WITH CONCURRENT TRANSACTIONS

For concurrency experiments we will open two parallel SQL query windows having SQL sessions "client A" and "client B" accessing the same database TestDB. For both sessions, we select the result to be listed in text mode by the following menu selections

Query > Results To > Results to Text

and set both sessions to use implicit transactions

```
SET IMPLICIT_TRANSACTIONS ON;
```

For making best visual use of the Management Studio, we can organize the parallel SQLQuery windows appear vertically side-by-side by pressing the alternate mouse button on the title of either SQLQuery window and selecting from the pop-up window the alternative "New Vertical Tab Group" (see figure 1.1).

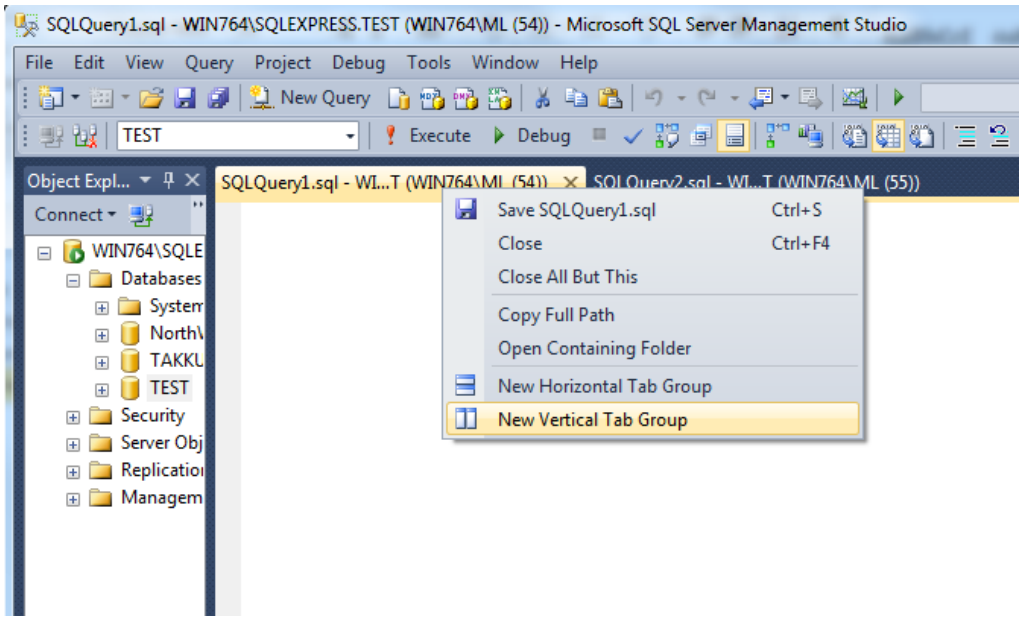


Figure A1.1 Opening two SQLQuery windows side-by-side

```
-----------------------------------------------------------------
-- Exercise 2.1   "Lost update problem simulation"
-----------------------------------------------------------------
-- 0. To start with fresh contents we enter following commands on a session
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
```

The Lost Update problem rises if some INSERTed or UPDATEd row is updated or deleted by some concurrent transaction before the first transaction ends. This might be possible in file-based "NoSQL" solutions, but modern DBMS products will prevent this. However, after the first transaction commits, any competing transaction can overwrite the rows of the committed transaction.

In the following we simulate the Lost Update scenario using the READ COMMITTED isolation, which does not keep the S-locks. First, the client applications read the balance values releasing the S-locks.

```
-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

acctID      balance
----------- -----------
101         1000

-- 2. Client B starts
SET NOCOUNT ON;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
```

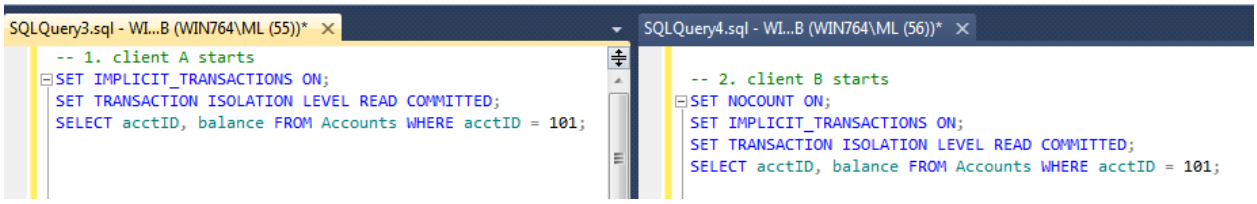
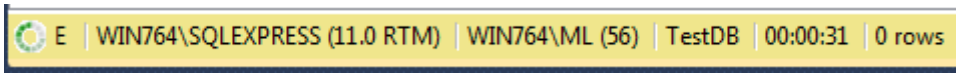


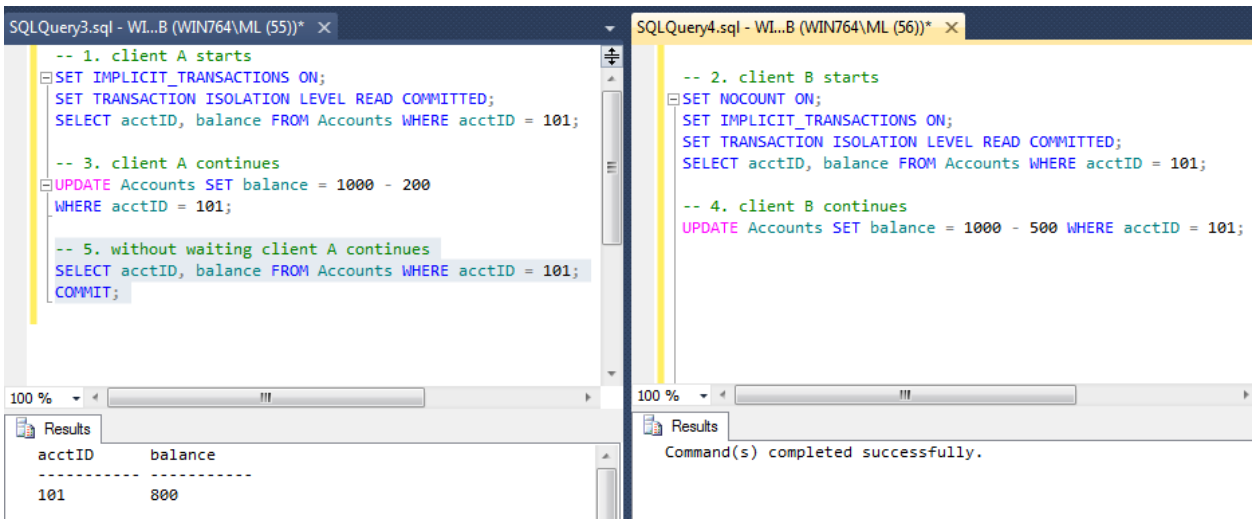
Figure A1.2 Competing SQL sessions in their windows

```
-- 3. Client A continues
UPDATE Accounts SET balance = 1000 - 200 WHERE acctID = 101;

-- 4. Client B continues
UPDATE Accounts SET balance = 1000 - 500 WHERE acctID = 101;
```

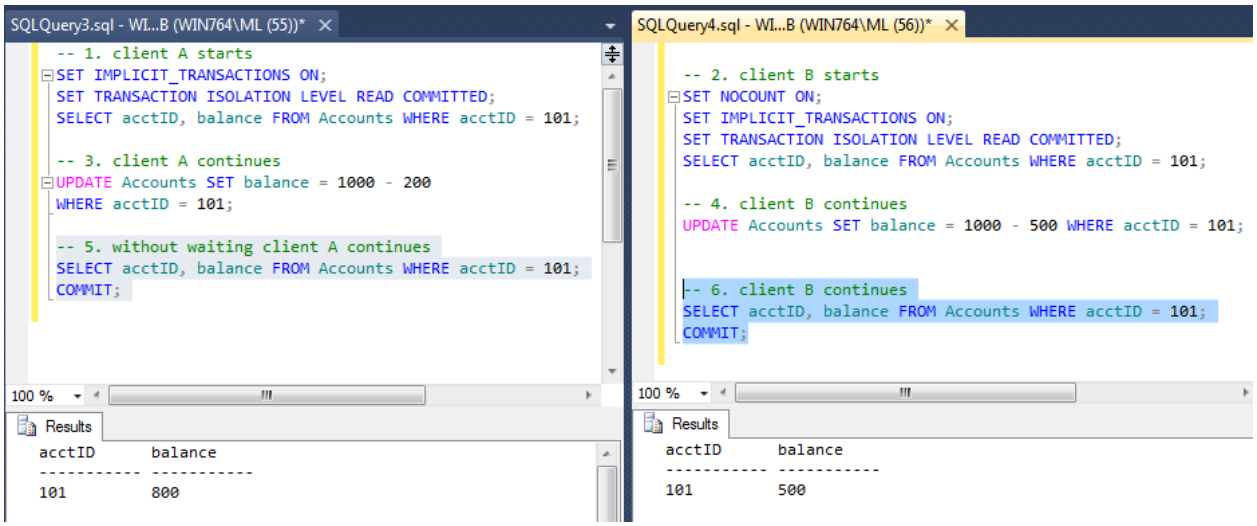


... waiting for A ...

```
-- 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;
```

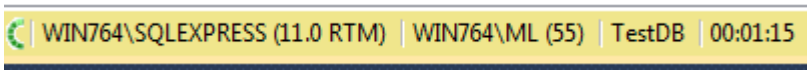


So, the final result is erroneous!

> **Note**: In this experiment, we did not have the real "Lost Update Problem", but after A commits its transaction, B can proceed and it overwrites the update made by A. We call this reliability problem "**Blind Overwriting**" or "**Dirty Write**". This can be solved if the UPDATE commands use ***sensitive updates*** like
>
> ```
> SET balance = balance - 500
> ```

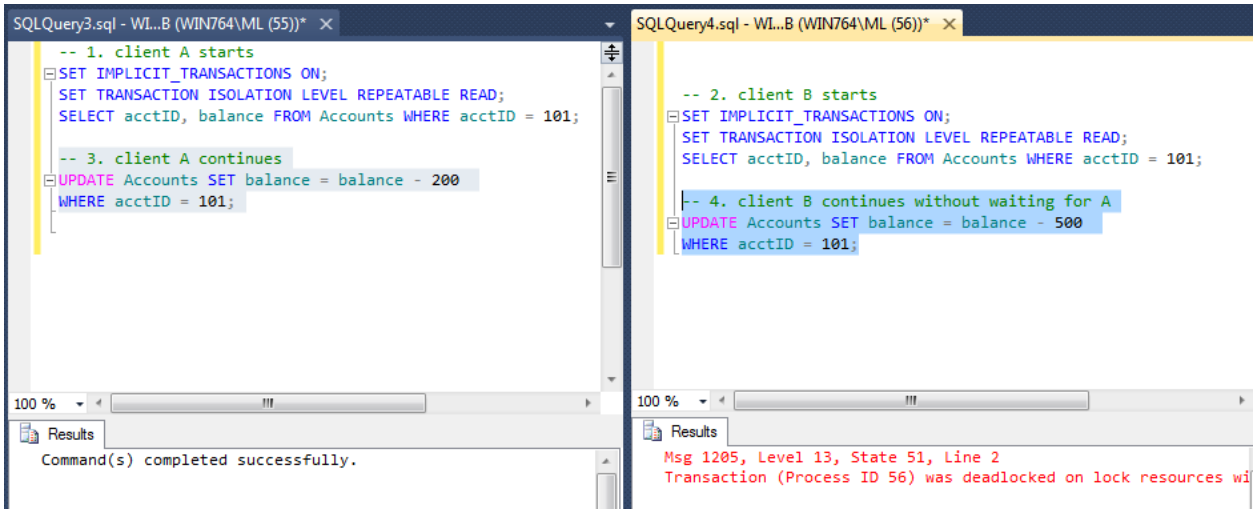
```
-----------------------------------------------------------------
-- Exercise 2.2   "Lost Update Problem" fixed by locks
-----------------------------------------------------------------
--      Competition on a single resource
--      using SELECT .. UPDATE scenarios both client A and B
--      tries to withdraw amounts from the same account.
```

```
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
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 IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

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



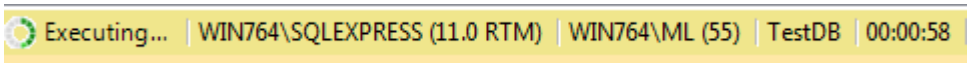
... waiting for B ...

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



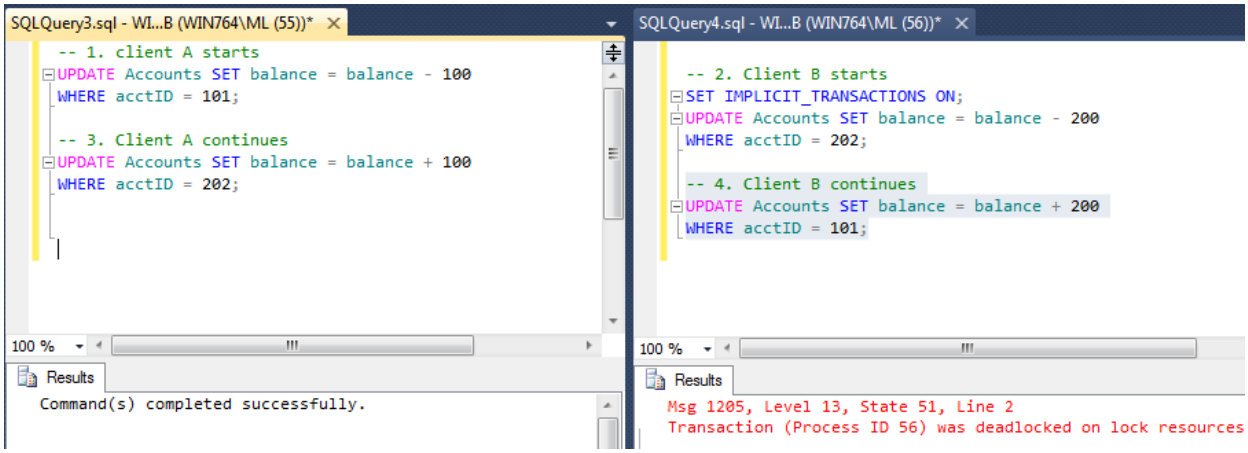

```
-- 5. The client which survived will commit
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID      balance
----------- -----------
101         800

COMMIT;
```

```
-----------------------------------------------------------------
-- Exercise 2.3  Competition on two resources in different order
--               using UPDATE-UPDATE scenarios
-----------------------------------------------------------------
-- 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 IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 202;

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

COMMIT;
```



... waiting for B ...

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




```
-- 5. Client A continues if it can ...
COMMIT;
```

In the exercises 2.4 – 2.7, we will experiment **concurrency anomalies** i.e. data reliability risks known by the ISO SQL standard. Can we identify those? How can we fix the anomalies?

```
-----------------------------------------------------------------
-- Exercise 2.4   Dirty Read?
-----------------------------------------------------------------
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
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 IMPLICIT_TRANSACTIONS ON;

UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100
COMMIT;


-- 3. Client A continues
ROLLBACK;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

COMMIT;
```

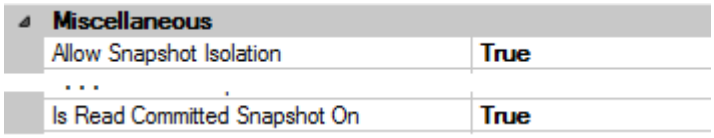
```
-----------------------------------------------------------------
-- Exercise 2.5   Non-repeatable Read?
-----------------------------------------------------------------
-- In non-repeatable read anomaly some rows read in the current transaction
-- may not appear in the resultset if the read operation would be repeated
-- before end of the transaction.

-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
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 IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

--  Listing accounts having balance > 500 euros
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
COMMIT;


-- 3. Client A continues
-- Can we still see the same accounts as in step 1?
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
202         2500

COMMIT;
```

```
-----------------------------------------------------------------
-- Exercise 2.6   Insert Phantom?
-----------------------------------------------------------------
-- Insert phantoms are rows inserted by concurrent transactions and
-- which the current might see before the end of the transaction.
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
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 TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Accounts having balance > 1000 euros
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Accounts (acctID,balance) VALUES (303,3000);
COMMIT;


-- 3. Client A continues
-- Let's see the results
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
202         2000
303         3000

COMMIT;
```

Question

- How could we prevent the phantoms?

```
-------------------------------------------------------------
-- SNAPSHOT STUDIES
-------------------------------------------------------------
-- The database needs to be configured to support SNAPSHOT isolation.
-- For this we create a new database

CREATE DATABASE SnapsDB;

-- to be configured to support snapshots as follows
```

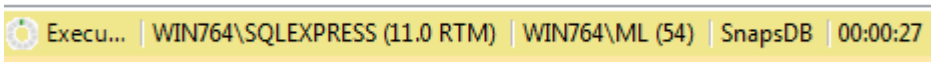



```
-- Then both client A and B are switched to use SnapsDB as follows:

USE SnapsDB;


-------------------------------------------------------------------
-- Exercise 2.7   A Snapshot study with different kinds of Phantoms
-------------------------------------------------------------------
USE SnapsDB;

-- 0. Setup the test: recreate the table T with five rows
DROP TABLE T;
GO

SET IMPLICIT_TRANSACTIONS ON;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), 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;

-- 1. Client A starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT ;

SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           first                          1
3           third                          1
5           to be or not to be             1
```

```
-- 2. Client B starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

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;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           fourth                         2
6           Insert Phantom                 1


-- 3. Client A continues
-- Let's repeat the query and try some updates
SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           first                          1
3           third                          1
5           to be or not to be             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 B' WHERE id = 1;

SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
3           update by A inside snapshot    1
5           to be or not to be             1
7           inserted by A                  1
```

```
-- 3.5 Client C in a new session starts and executes a query
USE SnapsDB;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           update by A after 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 row(s) affected)
```

```
-- 4. Client B continues
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           fourth                         2
6           Insert Phantom                 1


-- 5. Client A continues
SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
3           update by A inside snapshot    1
5           to be or not to be             1
7           inserted by A                  1

UPDATE T SET s = 'update after delete?' WHERE id = 5;
```



... waiting for B ...

```
-- 6. Client B continues without waiting for A
COMMIT;

-- 7. Client A continues
Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use
snapshot isolation to access table 'dbo.T' directly or indirectly in database
'SnapsDB' to update, delete, or insert the row that has been modified or
deleted by another transaction. Retry the transaction or change the isolation
level for the update/delete statement.

-- 8. Client B continues
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1

(5 row(s) affected)
```

Question

- Explain how the experiment proceeded. Explain the different results of steps 3.5 and 4.

--

For more advanced topics on SQL Server transactions see

Kalen Delaney (2012), “SQL Server Concurrency, Locking, Blocking and Row Versioning”

ISBN 978-1-906434-90-8

Appendix 2 Transactions in Java Programming

We can experiment with SQL transactions as available in the SQL dialect and the services of the DBMS product to be used just using interactive SQL in SQL-client tools, so called SQL Editors. However, for applications the data access is programmed using some data access API. To give a short example of writing SQL transactions using some API, we present the Bank Transfer example implemented in a Java program using JDBC as the data access API. We assume that the reader of this appendix is already familiar with Java programming and the JDBC API, and Figure A2.1 serves only as a visual overview helping to memorize the major objects, methods, and their interaction.

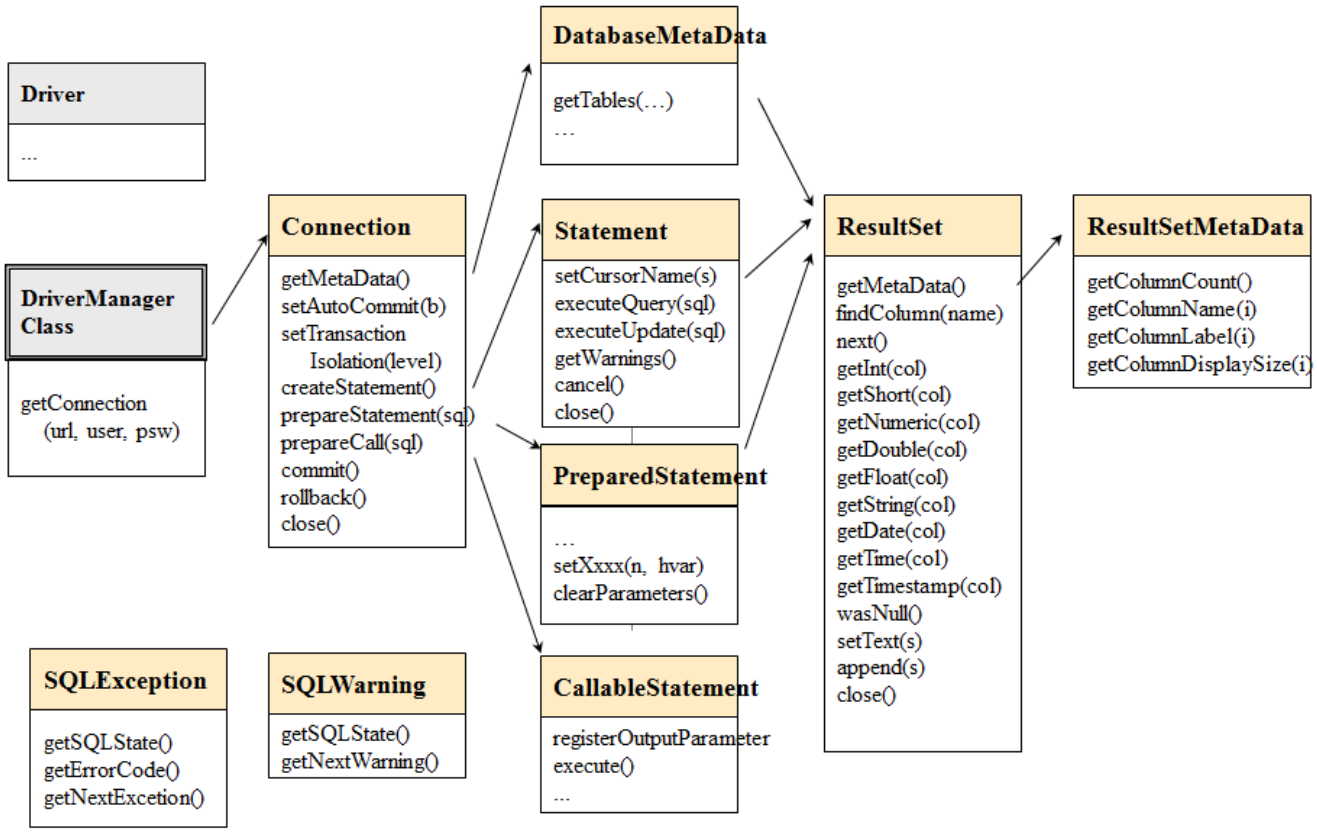


Figure A2.1 A simplified overview of JDBC API

Example Java code: BankTransfer

The program BankTransfer transfers an amount of 100 euros from one bank account (fromAcct) to another bank account (toAcct). The driver name, the URL of the database, username, user's password, fromAcct and toAcct parameters are read from the command line parameters of the program.

For test arrangement, the user should start two Command Windows on Windows platforms (or Terminal windows on Linux platforms) and run the program concurrently in both windows so that the fromAcct of the first is the toAcct of the other and vice versa, see the scripts below.

After updating the fromAcct the program waits for the ENTER key to continue so that test runs get synchronized and we can test the concurrency conflict. The source program demonstrates also the **ReTryer data access pattern**.

Listing A2.1 The BankTransfer Java code using JDBC

```
/*  DBTechNet Concurrency Lab 15.5.2008 Martti Laiho
Save the java program as BankTransfer.java and compile as follows
  javac BankTransfer.java
See BankTransferScript.txt for the test scripts applied to SQL Server, Oracle and DB2
Updates:
2.0 2008-05-26 ML preventing rollback by application after SQL Server deadlock
2.1 2012-09-24 ML restructured for presenting the Retry Wrapper block
2.2 2012-11-04 ML exception on non-existing accounts
2.3 2014-03-09 ML TransferTransaction returns 1 for retry, 0 for OK, < 0 for error
**********************************************************/
import java.io.*;
import java.sql.*;
public class BankTransfer {
   public static void main (String args[]) throws Exception
   {
      System.out.println("BankTransfer version 2.3");
      if (args.length != 6) {
         System.out.println("Usage:" +
         "BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%");
         System.exit(-1);
      }
      java.sql.Connection conn = null;
      boolean sqlServer = false;
      int counter = 0;
      int retry = 0;
      String URL = args[1];
      String user = args[2];
      String password = args[3];
      int amount = 100;
      int fromAcct = Integer.parseInt(args[4]);
      int toAcct = Integer.parseInt(args[5]);
      // SQL Server's explicit transactions will require special treatment
      if (URL.substring(5,14).equals("sqlserver")) {
        sqlServer = true;
      }
            // register the JDBC driver and open connection
      try {
         Class.forName(args[0]);
         conn = java.sql.DriverManager.getConnection(URL,user,password);
      }
      catch (SQLException ex) {
           System.out.println("URL: " + URL);
           System.out.println("** Connection failure: " + ex.getMessage() +
                        "\n SQLSTATE: " + ex.getSQLState() +
                        " SQLcode: " + ex.getErrorCode());
           System.exit(-1);
      }
```

```
      do {
      // Retry wrapper block of TransaferTransaction ----------
         if (counter++ > 0) {
            System.out.println("retry #" + counter);
            if (sqlServer) {
               conn.close();
               System.out.println("Connection closed");
               conn = java.sql.DriverManager.getConnection(URL,user,password);
               conn.setAutoCommit(true);
            }
         }
         retry = TransferTransaction (conn, fromAcct, toAcct, amount, sqlServer);
         if (retry == 1) {
            long pause = (long) (Math.random () * 1000); // max 1 sec.
            System.out.println("Waiting for "+pause+ " mseconds before retry"); // just for testing
            Thread.sleep(pause);
         } else
         if (retry < 0) System.out.println (" Error code: " + retry + ", cannot re-try.");
      } while (retry == 1 && counter < 10);  // max 10 retries
      // end of the Retry wrapper block ------------------------------

      conn.close();
      System.out.println("\n End of Program. ");
   }

   static int TransferTransaction (Connection conn,
         int fromAcct, int toAcct, int amount,
         boolean sqlServer
         )
         throws Exception {
         String SQLState = "*****";
         String errMsg = "";
         int retry =  0;
         try {
            conn.setAutoCommit(false);   // transaction begins
            conn.setTransactionIsolation(
                Connection.TRANSACTION_SERIALIZABLE);
            errMsg = "";
            retry = 0; //"N";
            // a parameterized SQL command
            PreparedStatement pstmt1 = conn.prepareStatement(
              "UPDATE Accounts SET balance = balance + ? WHERE acctID = ?");
            // setting the parameter values
            pstmt1.setInt(1, -amount);  // how much money to withdraw
            pstmt1.setInt(2, fromAcct); // from which account
            int count1 = pstmt1.executeUpdate();
            if (count1 != 1) throw new Exception ("Account "+fromAcct + " is missing!");

            // ******  interactive pause just for concurrency testing ******
            // In the following we arrange the transaction to wait
            // until the user presses ENTER key so that another client
            // can proceed with a conflicting transaction.
            // This is just for concurrency testing, so don't apply this
```

```
            // user interaction in real applications!!!
              System.out.print("\nPress ENTER to continue ...");
              BufferedReader reader = new BufferedReader(
                      new InputStreamReader(System.in));
              String s = reader.readLine();
            // ****** end of waiting ****************************

              pstmt1.setInt(1, amount);  // how much money to add
              pstmt1.setInt(2, toAcct);  // to which account
              int count2 = pstmt1.executeUpdate();
              if (count2 != 1) throw new Exception  ("Account "+toAcct + " is missing!");
              System.out.print("committing ..");
              conn.commit();  // end of transaction
              pstmt1.close();
            }
            catch (SQLException ex) {
              try {
                errMsg = "\nSQLException:";
                while (ex != null) {
                    SQLState = ex.getSQLState();
                    // is it a concurrency conflict?
                    if ((SQLState.equals("40001")   // Solid, DB2, SQL Server,...
                      || SQLState.equals("61000")   // Oracle ORA-00060: deadlock detected
                      || SQLState.equals("72000"))) // Oracle ORA-08177: can't serialize access
                        retry = 1; //"Y";
                    errMsg = errMsg + "SQLState: "+ SQLState;
                    errMsg = errMsg + ", Message:  "+ ex.getMessage();
                    errMsg = errMsg + ", Vendor:   "+ ex.getErrorCode() + "\n";
                    ex = ex.getNextException();
                }
                // SQL Server does not allow rollback after deadlock !
                if (sqlServer == false) {
                   conn.rollback();  // explicit rollback needed for Oracle
                              // and the extra rollback does not harm DB2
                }
                // println for testing purposes
                System.out.println("** Database error: " + errMsg);
              }
              catch (Exception e) { // In case of possible problems in SQLException handling
                System.out.println("SQLException handling error: " + e);
                conn.rollback();  // Current transaction is rolled back
                retry = -1; // This is reserved for potential exception handling
              }
            } // SQLException
            catch (Exception e) {
              System.out.println("Some java error: " + e);
              conn.rollback();  // Current transaction is rolled back also in this case
              retry = -1; // This is reserved for potential other exception handling
            } // other exceptions
            finally { return retry; }
        }
    }
```

The scripts in Listing A2.2 can be used for testing the program on a Windows workstation in two parallel Command Prompt windows. The scripts assume that DBMS product is SQL Server Express, the database is called "TestDB", and the JDBC driver is stored in the subdirectory "jdbc-drivers" of the current directory of the program.

Listing A2.2 Scripts for experimenting with BankTransfer on Windows

```
rem Script for the first window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set
URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=101
set toAcct=202
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%


rem Script for the second window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set
URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=202
set toAcct=101
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%
```

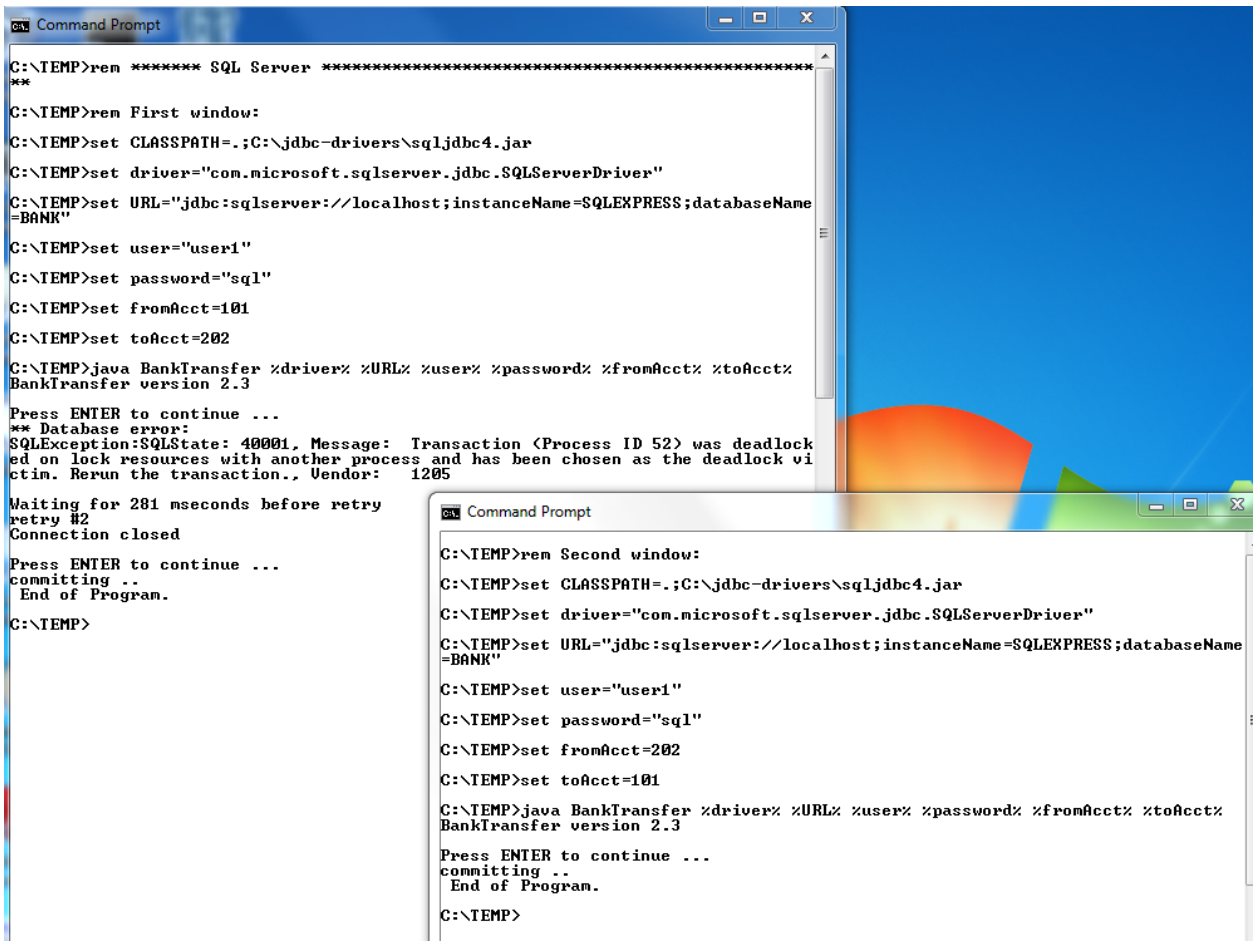


Figure A2.1 Sample tests of BankTransfer on Windows

The scripts for other DBMS products and Linux platform can be easily modified from the scripts shown in listing A2.2.

Listing A2.3 Scripts for experimenting with BankTransfer using MySQL on the Linux platform of DebianDB where the BankTransfer.java code has been stored and compiled in the Transactions directory of user student and the JDBC drivers have been installed in directory /opt/jdbc-drivers as follows

Scripts for MySQL on Linux:

```
# Creating directory /opt/jdbc-drivers for JDBC drivers
cd /opt
mkdir jdbc-drivers
chmod +r+r+r jdbc-drivers
# copying the MySQL jdbc driver to /opt/jdb-drivers
cd /opt/jdbc-drivers
cp $HOME/mysql-connector-java-5.1.23-bin.jar
# allow read access to the driver to everyone
chmod +r+r+r mysql-connector-java-5.1.23-bin.jar


#******* MySQL/InnoDB ************************************************

# First window:
cd $HOME/Transactions
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/TestDB
export user=user1
export password=sql
export fromAcct=101
export toAcct=202
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password
$fromAcct $toAcct


# Second window:
cd $HOME/Transactions
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/TestDB
export user=user1
export password=sql
export fromAcct=202
export toAcct=101
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password
$fromAcct $toAcct
#*********************************************************************
```

Appendix 3 Transactions and Database Recovery

Figure A3.1 presents the "big picture" of transaction processing internals for a typical database server. The tutorial slides,"Basics of SQL Transactions" available at

http://www.dbtechnet.org/papers/BasicsOfSqlTransactions.pdf

present in more detail the architectures of some mainstream DBMS products, including the management of database files and that of transaction logs (e.g. snapshots of the MS SQL Server transaction log); the management of the bufferpool (data cache) in memory to keep disk I/O at minimum for increased performance; the way SQL transactions are served; the reliability of COMMIT operations; and the implementation of ROLLBACK operations.

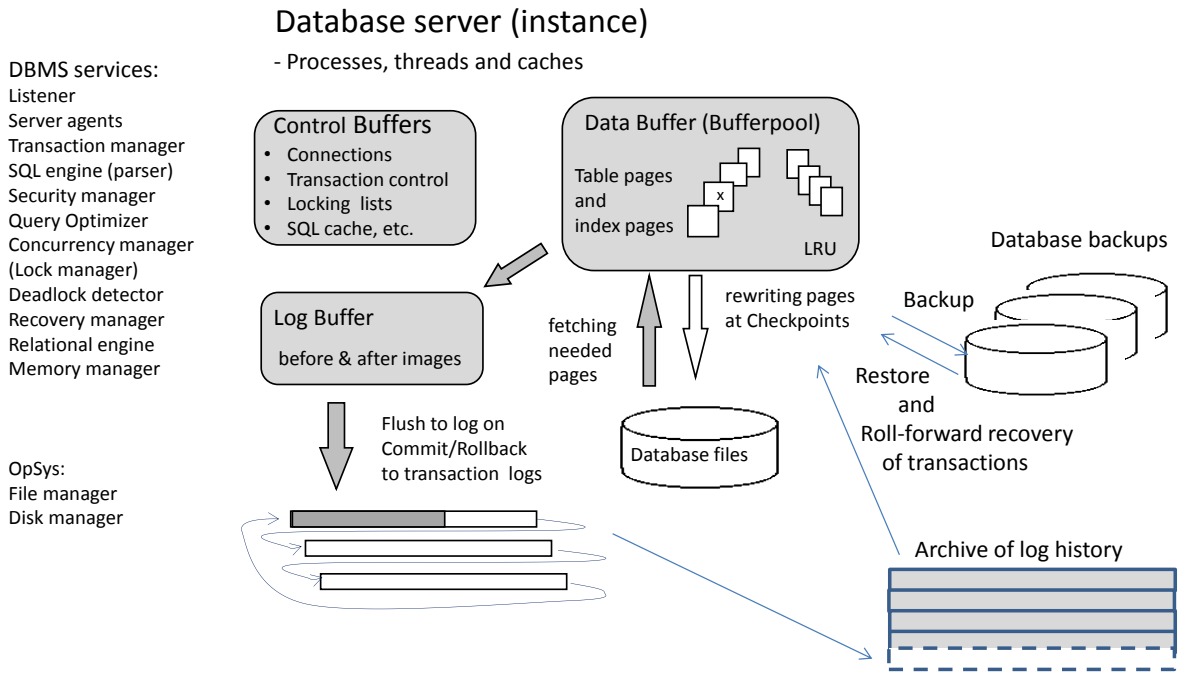


Figure A3.1 A generic overview of a Database Server

In the following, we describe how DBMS can recover the database up to the latest committed transaction in case of power failures and soft crashes of the server. In case of hardware failures, the database can be recovered using the backup of the database and the saved transaction log history since the database backup.

(See also: **http://www.dbtechnet.org/papers/RdbmsRecovery.ppt**)

When an SQL transaction starts the database server will give it a unique transaction id number, and for every action in the transaction it will write log records into the transaction log file. For every processed row the log record contains the transaction id and original contents of the row as "before image" and the contents of the row after the operation as "after image". In case of INSERT commands the before image parts are empty, and in case of DELETE commands the after image parts are empty. Also for COMMIT and ROLLBACK operations, the corresponding log records will be written into the log, and as part of this, all the log records of the transaction will be written into the transaction log file on a disk. The control from the COMMIT operation will return to the client only after the commit record has been written to the disk.

From time to time the database server will make a CHECKPOINT operation in which serving the clients will stop; all transaction log records from the log cache will be written to the transaction log file and all the updated data pages (marked by a "dirty bit") in data cache (bufferpool) will be written in their original places in the data files of the database, and the "dirty bits" of those pages will be cleared in the data cache. In the transaction log, a group of checkpoint records will be written including a list of transaction id numbers of the currently running transactions. Eventually, serving the clients continues.

> **Note**: In a managed shutdown of the server, there should not be any active SQL sessions running in the server, so that as the last transaction log operation the server writes an empty checkpoint record indicating a clean shutdown.

Figure A3.2 presents a history in a transaction log before the database instance fails down or the whole server computer stops, for example, due to some power failure. All the files on the disk are readable, but the contents in the bufferpool has been lost. The situation is called a **Soft Crash**.

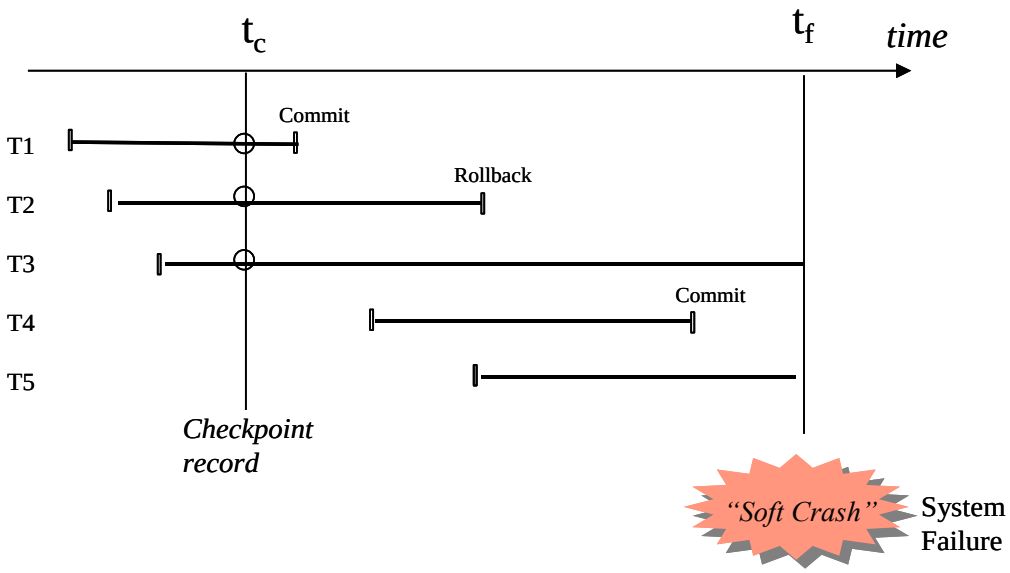


Figure A3.2 A history in transaction log before a soft crash

When the server is restarted, it will first find the last checkpoint record in the transaction log. If the checkpoint record is empty, it indicates a clean shutdown and the server is ready to serve the incoming clients. Otherwise, the server starts a rollback recovery procedure as follows: First, all the transaction numbers listed in the checkpoint record will be copied to the UNDO list of transactions to be rolled back, and the REDO list of the id numbers of the transaction to be redone into the database is set empty. The server scans then the transaction log from the checkpoint to the end of the log enlisting all new transaction ids to the UNDO list and moving the ids of committed transactions from the UNDO list to the REDO list. After that, the server proceeds backwards in the log writing to the database the before images of all the rows of transactions listed in the UNDO list, and then proceeds forward from the checkpoint record writing the after images of all the rows of transactions listed in the REDO list. After that, the database has been recovered into the level of the last committed transaction before the soft crash, and the server can start serving the clients (see figure A3.3).

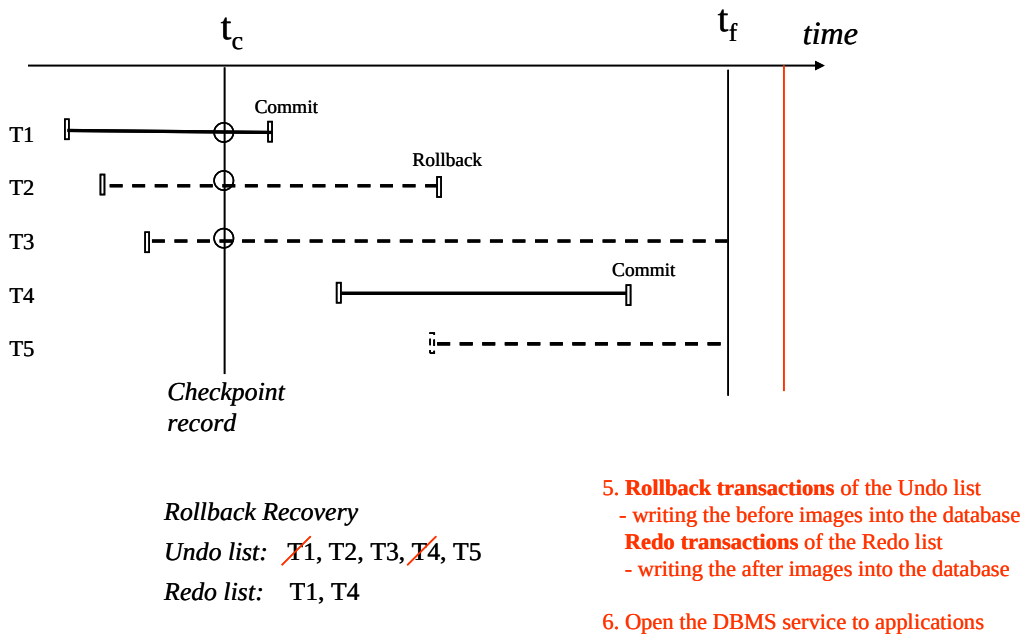


Figure A3.3 Rollback recovery of the database

Note: Apart from the simplified recovery procedure described above, most modern DBMS products use the ARIES protocol in which, between the checkpoints, when there is no load in the database, a "lazy writer" thread synchronizes the dirty pages from the data cache back to data files. The synchronized pages marked by LSN marks of ARIES can then be omitted in the rollback recovery, and this will make the rollback recovery faster.

INDEX

Reader comments:

"*That is an amazing work of yours. It is perfect elearning material, together with online tools. I would never believe something like this is possible in connection with transaction management. Debian platform is welcome*".
- Ferenc Kruzslicz

''This is the only publication we know to have both solid theoretical foundation and practical exercises with so many different database products."