7 Transactions

Learning Goals:
- Understand what a transaction is
- Be able to explain the ACID principle and its importance, create examples to show the importance of ACID
- Execute, commit, and rollback transactions

Definition:
A transaction is a sequence of data access operations that transfers the database from one consistent state to another consistent state (where the new state is not necessarily different from the old).

Or:
A transaction is a business process that is atomic in the view of the users.

Examples of transactions:
- "Compile a list of all students that take the class DTB-WS00".
- "Set the grade 1.0 in the class DTB-WS00 for the student with matNr 1234".
- "Give all TAs a 10% pay rise".

A transaction can be an entire program, a part of a program, or a single command.
From the point of view of a database, an application program can be thought of as a series of transactions with non-database processing taking place between the database accesses.

Transaction processing systems are systems with large databases and a large number of users who execute transactions in parallel. Examples include travel reservation systems (hotels, flights), credit card processing, stock markets, and supermarket checkout.

A transaction can terminate successfully or reach some state where it is discovered that an error has occurred. In the first case, the transaction is said to have "committed", in the second case it is "aborted".
In the intermediate stages of a transaction, it is accepted that the state of the database can be temporarily inconsistent; but at the end of the transaction a consistent state must be reached. An aborted transaction therefore must be "undone" (any changes it has performed must be reversed), to go back to the previous consistent state. The "undo" is also known as a "roll back".

Syntax for Transactions (DCL Data Control Language):
Pseudocode of a database program:

```
BEGIN_TRANSACTION
   instructions
   ...
if everything ok then COMMIT
else ABORT
END_OF_TRANSACTION
```

Some systems do not provide a begin_transaction statement. Instead, the first SQL command that is executed after the end of a transaction, automatically starts a new transaction. If no begin_transaction, commit or abort are used in a database access session, the entire execution is assumed to be one transaction. It is committed if the sessions ends without problem. In case of an error or failure, it is aborted and rolled back.

How to do it in MySQL:

Excerpt from the MySQL online documentation:
By default, MySQL runs with autocommit mode enabled. This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk.

If you are using transaction-safe tables (like InnoDB or BDB), you can disable auto-commit mode with the following statement:

SET AUTOCOMMIT=0;

You can check whether autocommit mode is on or off by entering
SHOW VARIABLES LIKE 'autocommit';

After disabling autocommit mode by setting the AUTOCOMMIT variable to zero, you must use COMMIT to store your changes to disk or ROLLBACK if you want to ignore the changes you have made since the beginning of your transaction.

If you want to disable autocommit mode for a single series of statements, you can use the START TRANSACTION statement:

Example:

START TRANSACTION;
INSERT INTO Bla VALUES (1,1);
COMMIT;
(or ROLLBACK;)

With START TRANSACTION, autocommit remains disabled until you end the transaction with COMMIT or ROLLBACK. The autocommit mode then reverts to its previous state.

Important features of transactions that must be provided by a DBMS are:

- **A**tomicity
  Either all operations of a transaction are executed or none at all.
  => If the transaction is interrupted by some error, all intermediate changes must be reversed.
- **C**onsistency
  A transaction must transform the database from a consistent state to a consistent state. In case of a distributed transaction: all involved sites agree on the outcome of the transaction.
- **I**solation
  Only after successful completion, partial results of a transaction may be released for usage by other transactions. => Transactions act independently of each other.
- **D**urability
  After the successful termination of a transaction, its results are persistent, i.e. they can only be changed or undone by a new transaction, even if errors or failures occur.

Reason for Isolation:
Assumption:
Transactions A and B are executed concurrently.
A uses a partial result of B and then terminates successfully.
After this, B is aborted => the partial result must be changed back.
=> A would need to be changed as well (-> cascading aborts).
=> Contradiction to Durability
Or otherwise (if no cascading aborts are allowed): contradiction to Consistency

Note: If it is discovered after a commit that the effects of a transaction must be undone, a "compensating transaction" is necessary to do this.

Implementation of the ACID properties:
- Atomicity:
  -> Recovery Manager rolls back in case of error
- Consistency:
  -> Transaction developers must program correctly; integrity constraints may be automatically checked by the DBMS; recovery manager (software, not a person) treats execution interruptions.
- Isolation:
  -> ensured by concurrency control mechanisms
- Durability:
  -> Recovery manager ensures durability after errors (redo if necessary).

Some notes about MySQL:

Excerpts from the MySQL online manual (http://www.mysql.com/doc/en):

„MySQL supports two different kinds of tables: transaction-safe tables (InnoDB and BDB) and not transaction-safe tables (HEAP, ISAM, MERGE, and MyISAM). You can convert tables between different types with the ALTER TABLE statement.

InnoDB provides MySQL with a transaction-safe (ACID compliant) storage engine

with commit, rollback, and crash recovery capabilities. InnoDB does locking on row level and also provides an Oracle-style consistent non-locking read in SELECT statements. These features increase multiuser concurrency and performance. There is no need for lock escalation in InnoDB, because row level locks in InnoDB fit in very small space.

From MySQL version 4.0, InnoDB is enabled by default.

In InnoDB all user activity happens inside transactions. If the autocommit mode is used in MySQL, then each SQL statement forms a single transaction. MySQL always starts a new connection with the autocommit mode switched on.

If the autocommit mode is switched off with SET AUTOCOMMIT = 0, then we can think that a user always has a transaction open. If he issues the SQL COMMIT or ROLLBACK statement, it ends the current transaction, and a new one starts. Both statements will release all InnoDB locks that were set during the current transaction. A COMMIT means that the changes made in the current transaction are made permanent and become visible to other users. A ROLLBACK statement, on the other hand, cancels all modifications made by the current transaction.

If the connection has AUTOCOMMIT = 1, then the user can still perform a multi-statement transaction by starting it with START TRANSACTION or BEGIN and ending it with COMMIT or ROLLBACK.

In terms of the SQL-92 transaction isolation levels, the InnoDB default is REPEATABLE READ. Starting from version 4.0.5, InnoDB offers all 4 different transaction isolation levels described by the SQL-92 standard. You can set the default isolation level for all connections in the [mysqld] section of `my.cnf':

```
transaction-isolation = {READ-UNCOMMITTED | READ-COMMITTED
                        | REPEATABLE-READ | SERIALIZABLE}
```

A user can change the isolation level of a single session or all new incoming connections with the SET TRANSACTION statement. Its syntax is as follows:

```
SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL
                    {READ UNCOMMITTED | READ COMMITTED
                     | REPEATABLE READ | SERIALIZABLE}”
```

Handling of Data Definition Statements in Transactions

Data definition statements like create table, create view, create function, create trigger cannot be rolled back in MySQL. They are also implicitly committed directly after their execution.