

Database transactions

- The learning objectives for this week are:
 - Knowing what a *database transaction* is and *why it has a crucial role in reliable software systems*
 - Knowing the difference between *business transaction*, *user transaction* and *database transaction*
 - Knowing what a *SQL transaction* is and *what are the SQL statements that the programmer needs to use to control SQL transactions*
 - Knowing what is meant by *concurrency control*
 - Knowing what a *deadlock* is and *how to demonstrate it*

Introduction to database transactions

- Let's imagine that we are responsible for designing database that needs to store bank account balances
- The design should support a basic user transaction of transferring money between two accounts. That is, withdrawing money from one account and depositing it to another
- We come up with the following `Account` table to match the requirements:

```
CREATE TABLE Account (  
    account_id INTEGER NOT NULL IDENTITY,  
    balance INTEGER NOT NULL,  
  
    CONSTRAINT PK_Account PRIMARY KEY (account_id)  
),
```

Introduction to database transactions

- Then we consider the user transaction of transferring money between two accounts. We think that we can handle it with just two separate `UPDATE` statements:

```
UPDATE account SET balance = balance + 100 WHERE account_id = 2
UPDATE account SET balance = balance - 100 WHERE account_id = 1
```

- The DBMS will execute these two statements separately one-by-one. Assuming that *both* of these statements are executed successfully the transfer is successful
- But what if there's a failure (for example system failure or disk crash) in the DBMS and the second statement is never executed:

```
UPDATE account SET balance = balance + 100 WHERE account_id = 2
-- ✨ There's a failure in the DBMS
```

Introduction to database transactions

- In this case, the second statement would never be executed, meaning that the money is deposited to one account but never withdrawn to the other one
- What we would have wanted is that either *both of the statements succeed* or *in case of a failure the database is restored to its previous state*
- This behavior can be achieved with *database transactions*
- A database transaction is a unit or sequence of work that is performed on a database
- All operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state

Business, user and database transactions

- 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 *database transactions* (SQL transactions)
- A database transaction involves multiple database operations, such as retrieving, creating or updating data in the database

The autocommit mode

- In the autocommit mode each individual SQL statement submitted for execution is treated as a *single transaction*
- If a statement *completes successfully*, it is *committed*
- If a statement *encounters any error*, it is *rolled back*
- That is, no changes can be rolled back after the statement has been successfully executed
- This is the default transaction management mode in SQL Server, MariaDB, and PostgreSQL

```
INSERT INTO Account (balance) VALUES (0) -- automatically committed
UPDATE Account SET balance = 100 WHERE account_id = 1 -- automatically committed
```

The autocommit mode

- Let's see what's the consistency problem mentioned in the previous bank account example. Let's assume that the total balance of the bank is 900 euros:

account_id	balance
1	300
2	600

- Now, if we start to execute a bank transfer (in the autocommit mode) to transfer *100 euros* from *account 1* to *account 2*:

```
UPDATE account SET balance = balance + 100 WHERE account_id = 2
-- 🌟 There's a failure in the DBMS
```

- Unfortunately, due to a system failure the second update is not executed

The autocommit mode

- That is, we end up with a situation where the database is *not in a consistent state* any more
- Now the total balance retrieved from the database is 1000 euros, even though it should be 900 euros:

account_id	balance
1	300
2	700

Database transactions

- Transactions have four standard properties, usually referred to by the acronym *ACID*
- *Atomicity* ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state
- *Consistency* ensures that the database properly changes states upon a successfully committed transaction
- *Isolation* enables transactions to operate independently of and transparent to each other
- *Durability* ensures that the result or effect of a committed transaction persists in case of a system failure


SQL transactions

- When the application logic needs to execute a sequence of SQL statements in an atomic fashion, then the statements need to be grouped as a logical unit of work called *SQL transaction*
- The following SQL statements can be used for starting and ending SQL transactions:
 - `BEGIN TRANSACTION` : starts a new transaction (explicit transaction start in SQL Server)
 - `COMMIT` : terminates the transaction and *make all changes permanent*
 - `ROLLBACK` : terminates the transaction and *rolls back all changes* (restores the database to the previous state)

SQL transactions

- In the bank transfer example we could have the following transaction:

```
BEGIN TRANSACTION -- start a new transaction
UPDATE account SET balance = balance + 100 WHERE account_id = 2
UPDATE account SET balance = balance - 100 WHERE account_id = 1
COMMIT -- make all changes permanent
```

-  Nothing will happen until we either `COMMIT` or `ROLLBACK` the transaction
- From the programmer's viewpoint, the advantage of the rollback statement is that it *undoes all database changes made during the current transaction*
- That is, there is no need for executing any of reverse operations to cancel the already completed insert, delete, and update operations

SQL transactions

- The `COMMIT` statement *can fail*, whereas the `ROLLBACK` statement *cannot ever fail*
- If a program crash or any system failure occurs before a transaction commits, the DBMS automatically rolls back all the changes made by the transaction
- Let's revisit the bank transfer example, but this time we will have the `UPDATE` statements *inside a transaction*:

```
BEGIN TRANSACTION
UPDATE account SET balance = balance + 100 WHERE account_id = 2
UPDATE account SET balance = balance - 100 WHERE account_id = 1
COMMIT
```

SQL transactions

- Initially, the total balance of the bank is 900 euros:

account_id	balance
1	300
2	600

- Now, if we execute a bank transfer in an *SQL transaction* to transfer *100 euros* from *account 1* to *account 2*:

```
BEGIN TRANSACTION
UPDATE account SET balance = balance + 100 WHERE account_id = 2
-- 🌟 Failure
```

SQL transactions

- Due to the failure the second `UPDATE` and the `COMMIT` statement are not executed
- If the client application failed or if the client computer went down or was restarted, this also broke the database connection. The *DBMS rolls back the transaction* when the network notifies it of the break
- If the failure was on the DBMS side then the *DBMS automatically rolls back all uncommitted transactions* when it goes through the standard recovery process when it is restarting

SQL transactions

- That is, we always end up with a situation where the database is in a *consistent state*
- The DBMS has automatically *undone the first update* and the total balance retrieved from the database is 900:

account_id	balance
1	300
2	600

Transaction modes

- *Transaction mode* determines how transactions are started. By default, transactions are managed at the database connection level
- A database connection operates either in *explicit transactions mode* or *implicit transactions mode*
- In the *explicit transactions mode*, a transaction must be started explicitly by executing the `BEGIN TRANSACTION` statement
- The explicit transactions mode is the default mode in SQL Server
- In the *implicit transactions mode*, transactions are *not* started explicitly by executing the `BEGIN TRANSACTION` statement
- Instead, a transaction begins *implicitly* with the first executable SQL statement after the database connection is opened or after the previous transaction is terminated

