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 transfering 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 transfering money between two accounts. We think that we can handle it with just two separete `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 withdrawed 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 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 can involve 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 two `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

Concurrency control

- To fulfil the business requirements the DBMS should protect database consistency *without sacrificing too much performance*
- In a multi-user environment the DBMS has to *interleave the actions of several transactions* but still preserve illusion that each user is executing alone
- The *concurrency control* mechanism is responsible to provide the means to guarantee that a transaction is isolated from the effects of concurrently scheduling other transactions

Concurrently control

- The programmer is responsible of informing the DBMS about the desired *level of isolation*
- If a transaction both reads and writes data, the *isolation level* of the transaction should usually be set as below to avoid anomalies caused by concurrent updates:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

- In this case the isolation level is set as `REPEATABLE READ`. We will soon find out how transactions isolation levels effect the transaction's behavior

Locking

- When our transaction changes (`INSERT` , `UPDATE` , `DELETE`) a row in a database, the DBMS *locks the row exclusively* so that *other concurrent transactions are not allowed to access (read or change) the row as long as the transaction is running*
- That is, the other transactions that want to access the row *must wait* until the transaction ends. This always applies in pure *lock-based concurrency control*

Locking

- If we start your transaction as follows:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
-- ...
```

- The DBMS allows other concurrent transactions to read (`SELECT`) a row that your transaction has already read, but it *prevents the other transactions to change the row as long as your transaction is running*
- That is, other transactions that want to change the row *must wait* until your transaction ends

Locking

- For example, if *user A* starts a transaction which *reads* a row in the `Account` table:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
-- 🔒 this will lock the row from changes
SELECT balance FROM Account WHERE account_id = 1
```

- Then, *user B* starts a transaction which *updates* the locked row in the `Account` table:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
-- 🕐 the row is locked, the transaction has to wait until the user A's transaction ends
UPDATE Account SET balance = 6000 WHERE accountNumber = 1;
```

- The transaction of user B has to wait until the user A transaction ends (`COMMIT` or `ROLLBACK`)

Locking

- Locking resolves concurrency conflicts by *blocking a transaction* (like in the previous example) or *forcing a transaction to abort*
- When transaction is blocked it has to wait for the resource. The default for the maximum waiting time is "forever"
- When two transactions start to *permanently block each other*, the DBMS *forces either one transaction to abort* (rollback). This allows the other transaction to continue normally
- A *deadlock* occurs when two (or more) transactions permanently block each other by each transaction having a lock on a resource which the other transaction is trying to lock *exclusively*

```
BEGIN TRANSACTION
UPDATE Account
SET balance = balance - 100
WHERE account_id = 1 -- 🔒 locking account 1
```

```
BEGIN TRANSACTION
UPDATE Account
SET balance = balance - 50
WHERE account_id = 2 -- 🔒 locking account 2
```

```
UPDATE Account
SET balance = balance - 50
-- 🕐 trying to lock account 2,
-- must wait for transaction B
WHERE account_id = 2
```

```
UPDATE Account
SET balance = balance - 100
-- 🕐 trying to lock account 1,
-- must wait for transaction A
WHERE account_id = 1
```

Example of a deadlock

- The following things happen in the example:
    i. Transaction A locks account 1 *exclusively* and updates it
    ii. Transaction B locks account 2 *exclusively* and updates it
    iii. Transaction A is trying to lock account 2 (must wait, B has already locked the row)
    iv. Transaction B is trying to lock account 1 (must wait, A has already locked the row)
- Transaction B cannot continue until transaction A completes, but transaction A is blocked by transaction B. This cycle is a deadlock

Example of a deadlock

- Both transactions in the deadlock will wait unless the deadlock is broken by the DBMS
- When the DBMS detects a deadlock it chooses one of the transactions as a victim and
  terminates it (rollback) with an error message
- This allows the other transaction to continue normally
- Locks protect our data but *decrease concurrency*. Even without a deadlock a transaction
  can make others wait for a long time

Row-level locking in SQL Server

- Depending on the lock type, when one user has a lock on a row, the lock prevents other
  users from modifying or even reading that row
- The basic lock types are *write lock* and *read lock*
- In SQL Server, to avoid a certain type of deadlock, "*intent to update*" *lock* is also available

Write lock

- If transaction T1 wants to *modify a row* (update, delete, insert), then it has to acquire a *write lock* on the row
- Write locks are *exclusive*, meaning that T1 has to wait until the row is free from all other locks
- When T1 has a write lock on the row, all other transactions that want to access (select or modify) the row have to wait until T1 ends
- A write lock is *never released before the transaction ends*

Read lock

- If transaction T1 wants to *select a row*, then it has to acquire a *read lock* on the row
- If another transaction has a *write lock* on the row, then T1 has to wait until the other
  transaction ends
- If there is no write lock on the row, then T1 gets a read lock on the row
- If the transaction isolation level is set to `READ COMMITTED`, then the read lock is *released immediately after the row is retrieved from the database*
- If the transaction isolation level is set to `REPEATABLE READ` or `SERIALIZABLE`, then the read lock *is not released before the transaction ends*

"Intent to update" lock

- In SQL Server, you can use the non-standard `(UPDLOCK)` locking hint that prevents a certain form of deadlocks that occur when two transactions are first reading the same row and and then updating the row:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT balance FROM Account (UPDLOCK) WHERE account_id = 1
```

- The DBMS allows only one transaction at time to *read the same row* with the `(UPDLOCK)` locking hint
- If another transaction tries to read the same row with the `(UPDLOCK)` locking hint, it has to wait until the first transaction ends

Summary of lock types

Statement	Acquired lock type	Allowed locks on the same row
`UPDATE`, `DELETE`, `INSERT`	Write lock	*None*
`SELECT`	Read lock	Any number of *read locks* and a "*intent to update*" *lock*
`SELECT` with the `(UPDLOCK)` hint	"Intent to update" lock	Any number of *read locks* (but *no* "*intent to update*" *locks*)

Summary

- *Database transactions* provide a way to perform multiple database operations in single independent unit of work which either succeeds or fails as a whole
- Database transactions ensure that the database properly changes states upon a successfully committed transaction
- Changes of failed database transactions are rolled back by the DBMS
- The *concurrency control* mechanism is responsible to provide the means to guarantee that a transaction is isolated from the effects of concurrently scheduling other transactions
- In *lock-based concurrency control* transactions *lock database rows* while reading and modifying a row
- Other transactions that want to access the lock row *must wait* until the transaction ends