INTRODUCTION

Databases usually store very large amounts of data. In the vast majority of applications, data integrity is critical (e.g. banking applications).
This consistency should be maintained when the system is used by many users, as well as when any system failure occurs. One way to maintain data consistency is to use transactional processing.

**A transaction is a logical unit consisting of one or more SQL statements issued by a single user.**

We will use an example to illustrate how the transaction mechanism works.
When a customer transfers money from a savings account to a checking account, he executes a transaction consisting of three parts:

1. debiting the savings account,
2. depositing money in a checking account and
3. registering the transaction in a transaction log:

```
BEGIN TRANSACTION;

UPDATE savings_account
SET balance = balance - 500
WHERE account = 3209;

UPDATE checking_account
SET balance = balance + 500
WHERE account = 3208;

INSERT INTO log VALUES
(log_seq.nextval, '1b',
3209, 3208, 500);

COMMIT;
```

The database must guarantee the execution of all three phases of the transaction - otherwise, the customer would be at risk of losing the money taken. If any of the commands included in the transaction cannot be executed (e.g. due to a hardware failure), then the entire transaction must be canceled. One error means cancellation of the entire sequence of commands.

Changes made by SQL expressions in a transaction can be committed or revoked. After the current transaction is finalized (by committing or revoking it), the next SQL statement will start a new transaction.
When you commit a transaction, all changes made by the SQL commands that are part of it are persisted. Any changes will not be visible to other users' expressions until their parent transaction is approved.

Rolling back a transaction means undoing all changes made by it. After the transaction is recalled, the data remains unchanged, as if the transaction had not been initiated at all.

CONCURRENT ACCESS ANOMALIES

Concurrent access to data (multiple users, multiple transactions) can cause data integrity risks. There are four concurrent access anomalies:

**Lost update**
An anomaly occurs when two transactions simultaneously read and modify the same data - one transaction overwrites the changes made by the other:

| Transakcja 1 | Transakcja 2 |
|---|---|
| … | READ( x ) |
| READ( x ) | … |
| WRITE( x ) | … |
| … | WRITE( x ) |
| COMMIT | … |
| … | COMMIT |

**Dirty read**
A dirty read occurs when the first transaction reads uncommitted changes made by the second transaction. If the second transaction is rolled back, the data read by the first transaction will be invalid because the rollback deleted the input changes. The first transaction will not be aware that the data it reads are incorrect:

| Transakcja 1 | Transakcja 2 |
|---|---|
| READ( x ) | … |
| WRITE( x ) | … |
| … | READ( x ) |
| ROLLBACK | … |
| … | WRITE( x ) |
| … | COMMIT |

**Non-repeatable read, fuzzy read**
Non-repeatable read occurs when you read the same data multiple times in a scope of the same transaction but it gives different results - the data is not protected against modification within the other transaction:

| Transakcja 1 | Transakcja 2 |
|---|---|
| … | READ( x ) |
| READ( x ) | … |
| … | UPDATE/DELETE( x ) |
| … | COMMIT |
| READ( x ) | … |
| COMMIT | … |

**Phantoms** occur when new data added to the database is available in transactions that started before the addition operation. The query results will contain the data added by other transactions that started after the start of a given transaction:

| Transakcja 1 | Transakcja 2 |
|---|---|
| READ( x ) | … |
| … | INSERT( x ) |
| … | COMMIT |
| READ( x ) | … |
| COMMIT | … |

TRANSACTION ISOLATION LEVELS

To increase concurrency, transaction isolation levels are defined, consciously accepting the occurrence of some anomalies. The Update Anomaly can never appear. It does not occur at any level of isolation. According to the ANSI/ISO SQL standard, following isolation levels are defined:

- **Read Uncommitted** - The transaction can read uncommitted data (data changed by another transaction still pending).
- **Read Committed** - The transaction cannot read uncommitted data changed by another transaction still pending.
- **Repeatable Read** - The transaction cannot change the data being read by another transaction.
- **Serializable** - The transaction has exclusive rights to read and write data - other transactions do not have access to the same data. In practice, this means that transactions using the same data are executed sequentially, not concurrently.

The MS SQL Server database implements all the listed transaction isolation levels.

MS SQL Server COMMANDS:

```
SET IMPLICIT_TRANSACTIONS OFF;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
BEGIN TRANSACTION;
COMMIT TRANSACTION;
ROLL BACK TRANSACTION;
```

## THE TASK

1. **Identify 3 tasks (use cases) in the database that require transactional processing (2 points)**
2. **Determine the appropriate isolation level for each of the identified tasks (1 point)**
3. **For one of the selected use cases, create scripts showing the operation of two concurrent processes: (2 points)**
   **a. at the transaction isolation level lower than the identified one (negative effects should be shown),**
   **b. at the identified transaction isolation level (show no negative effects previously present)**

4. **Handing over task IV consists of:**
   **– presenting the task to the Tutor,**
   **– showing and running scripts containing clearly formatted queries.**