2.0
Transaction Concept

Reading: [Kl], chapter 7

Course content

1. Introduction SQL and NoSQL Databases

2. Relational Databases:
    1. Transactions
    2. Concurrency Control and Consistency

3. Data and Storage Models
    1. Relational (Reference)
    2. Key-Value
    3. Document
    4. Wide-Column / Graph (briefly)

4.Distributed Databases
    1. Replication
    2. Sharding
    3. Distributed Transactions / Consistency in Distributed Databases

knirsch@htw-berlin.de

Transaction Concept

Connecting to the database via PSQL command line tool:

path to Postgres bin filder needs to be in system path.

psql -h localhost -p 5432 -U postgres -d lesson

Command prompt opens:

lesson=#

We schedule a couple of lessons in our database.
What will happen when we insert the following two rows into table lesson?

```
INSERT INTO lesson (t_id, lesson_time, s_username, subjectcode)
            VALUES (14, '2024-04-08 05:22:12.000000', 'Mickey', 'EN');

INSERT INTO lesson (t_id, lesson_time, s_username, subjectcode)
            VALUES (14, '2024-04-08 05:22:12.000000', 'Rose', 'EN');
```

Does the database react as we expect?

eva.knirsch@kiu.edu.ge

Transaction Concept

User Mickey schedules a lesson

```
/* Operation Sequence 1 */
UPDATE student SET s_balance = (s_balance - 1) WHERE s_username = 'Mickey';
/* trigger that checks whether balance > 0 */
UPDATE teacher SET t_payment = (t_payment + 1) WHERE t_id = 1;
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
        VALUES (1, '2023-03-08 05:22:12.000000', 'Mickey', 'EN', 'true');
```

User Rose schedules a lesson:

```
/* Operation Sequence 2 */
UPDATE student SET s_balance = (s_balance - 1) WHERE s_username = 'Rose';
/* trigger that checks whether balance > 0 */
UPDATE teacher SET t_payment = (t_payment + 1) WHERE t_id = 1;
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
        VALUES (1, '2023-03-08 05:22:12.000000', 'Rose', 'EN', 'true');
```

Explain the problem!

Attention: PGAdmin behaves differently. Run these commands in a command line tool.

eva.knirsch@kiu.edu.ge

Transaction Concept – All Or Nothing Principle

In order to keep the database or the business logic consistent, it is often crucial that **all commands** of a sequence run successfully or **no command** runs at all.

A sequence of commands that necessarily needs to run together is called a transaction. The commands of a transaction are seen as one unit: Either all of the commands succeed or none.

A lot of issues can prevent the necessary operations from running successfully:

- Duplicate key error,
- Violation of FK or referential integrity,
- Violation of CHECK or trigger conditions,
- other application errors,
- connection gets lost in the middle of the commands' execution.
- Electricity shuts down after the first write operation.
- Database crashes after first write operation.

eva.knirsch@kiu.edu.ge

Transaction management

Textbook example of a typical transaction - bank transfer - in a banking application:

1. Read the account balance of *A* into the variable *a*: **read**(*A*,*a*);
2. Reduce the account balance by 50: *a*:= *a* - 50;
3. Write the new account balance to the database: **write**(*A*,*a*);
4. Read the account balance of *B* into the variable *b*: **read**(*B*,*b*);
5. Increase the account balance by 50: *b* := *b* + 50;
6. Write the new account balance to the database: **write**(*B*, *b*);

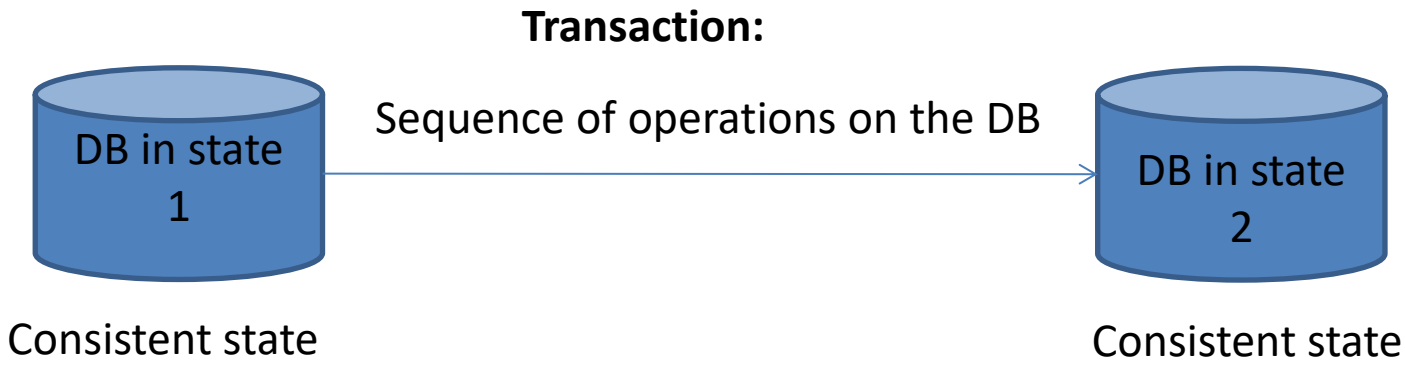
Slide: Kemper

Transaction Concept

The transaction concept is the All-Or-Nothing-Principle: all commands successful or no command.

Definition Transaction:

A transaction is a sequence of (read or write) operations that transfers a database from one consistent state to another consistent, not necessarily different state.



Transaction Concept

[Kl], adapted page 222:

A transaction is a way for an application to group several reads and writes together into a logical unit. Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (**commit**) or it fails (**abort, rollback**). If it fails, the application can safely retry.

With transactions, error handling becomes much simpler for an application, because it doesn't need to worry about partial failure—i.e., the case where some operations succeed and some fail (for whatever reason).

Transactions ….. created with a purpose, namely to ***simplify the programming model*** for applications accessing a database. .... we call these *safety guarantees*.

SQL standard includes support of transactions. So, basically all RDBMS support transactions. In mariadb the storage engine INNODB supports transactions.
NoSQL databases: some NoSQL databases support transactions to some extent. Most NoSQL databases do NOT support transactions.

eva.knirsch@kiu.edu.ge

Transcation Concept

Lets go back to our example and run the sequences as two transactions in two sessions:

```
Begin;
UPDATE student SET s_balance = (s_balance - 1) WHERE s_username = 'Mickey';
UPDATE teacher SET t_payment = (t_payment + 1) WHERE t_id = 14;
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
    VALUES (14, '2023-03-07 05:22:12.000000', 'Mickey', 'EN');
Commit;
```

```
Begin;
UPDATE student SET s_balance = (s_balance - 1) WHERE s_username = 'Rose';
UPDATE teacher SET t_payment = (t_payment + 1) WHERE t_id = 14;
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
    VALUES ('14', '2023-03-07 05:22:12.000000', 'Rose', 'EN');
Commit;
```

What is the result?

eva.knirsch@kiu.edu.ge

Transcation Concept

How do the 2nd transaction end?

T1:
lesson=*# Commit;
COMMIT

T2:
lesson=!# Commit;
ROLLBACK

Database stays consistent. But the application needs to react to the rollback.

Transaction Examples

A transaction is a sequence of operations executed as logical unit on the objects of the database. The sequence of commands must be processed completely or not at all.

Where do you find transactions:

- Commerce activities (trading, buying, selling, money transfer, ...)
- Reservation systems
- Banking
- Warehousing
- basically in all business applications
- and in a lot of other applications

Transaction Commands

- Start Transaction / Begin / Begin of Transaction / BoT
- Commit: Ends a transaction successfully: All writes made by the transaction are durable (persistent).
- Rollback / abort: The transaction is aborted. The DBMS restores the database to exactly the consistent state that existed before the transaction started.
  The DBMS automatically performs a rollback if it could not run all commands of the transaction. That is, the DBMS allows a transaction to fail and performs a rollback on its own.
- Rollback set in application code: Of course, a rollback can also be set in the application code by the application developer for error handling.

SQL standard does not specify how transaction management is to be implemented in the DBMS: That is during the transaction, writes are not yet persistent, after a commit they are persistent. Different databases have their own mechanisms.

ACID Principle

According to Kleppmann, transactions are safety guarantees for the developer. These safety guarantees comprise four characteristics, the so-called ACID characteristics:

Atomicity,

Consistency,

Isolation,

Durability.

Databases that support the ACID characteristics are considered transactional databases.

ACID Principle

Atomicity

A transaction is indivisible.
Either all operations are executed successfully or none is executed. All-or-Nothing-Principle.

Consistency

In the context of ACID this means:
a transaction transfers a database from one consistent state to another (not necessarily different) consistent state, preserving all defined rules (referential integrity, constraints, cascades, triggers, etc.).
Within a transaction, all rules or constraints, placed on the data, need to be kept.

ACID Principle

Durability With successful commit, all changes made within a transaction are made permanent.

How are changes made permanent?

https://www.postgresql.org/docs/current/tutorial-transactions.html

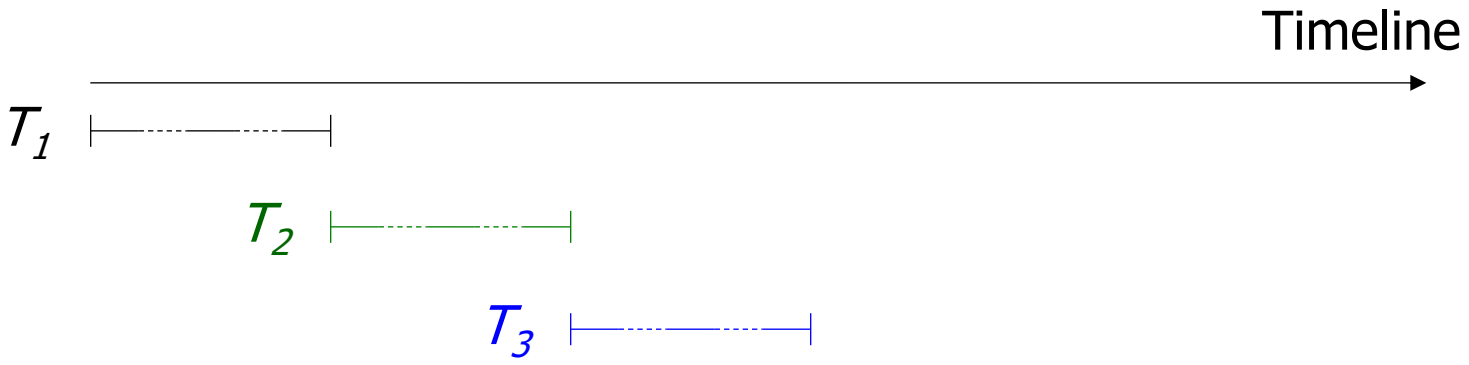
ACID Principle

Isolation Concurrent transactions run as if each transaction had the database completely ‘to itself‘.

[Kl], pager 225:
„ *Isolation* in the sense of ACID means that concurrently executing transactions are isolated from each other: they cannot step on each other’s toes….. The database ensures that when the transactions have committed, the result is the same as if they had run *serially* (one after another), even though in reality they may have run concurrently [10].

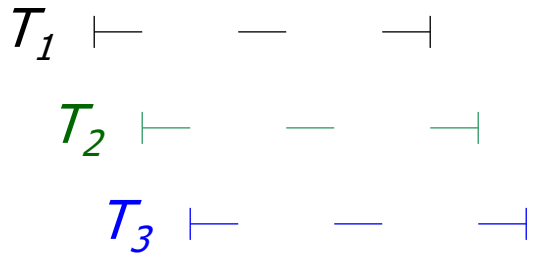
Never believe that your transaction is the only one out there.

Serial Processing of Transactions:



There is no isolation issue in serial processing of operations: a transaction only starts after the previous one completed.

Concurrent processing of transactions (multi-user / multi-application mode)



Adapted from Kemper

Concurrent Transactions

Conflicting goals:

Performance:
Concurrent transactions significantly increase the performance of a system

Consistency:
Concurrent transactions may lead to inconsistency.

Isolation demands that

... the result of concurrent transactions in multi-user mode must be the same as if these transactions had been performed serially in single-user mode.

Transaction Isolation Concept

Example:

```
/* client session 1 – Agency pays teacher 1 */
select t_payment from teacher where t_id = 1;
/* some application code - calculating how much money teacher gets and doing the money transfer */
select pg_sleep(20);
update teacher set t_payment=0 where t_id = 1;
```

Attention: Client PGAdmin behaves differently. Run these commands in a command line tool.

eva.knirsch@kiu.edu.ge

Transaction Isolation Concept

```
/* Client Session 2 – teacher 1 schedules lessons  */
update student set s_balance = s_balance - 3 where s_username= 'Rose';
/* trigger that checks that balance > 0 */
update teacher set t_payment = t_payment + 3 where t_id = 1;
select t_payment from teacher where t_id = 1;
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
        VALUES (1, '2024-03-03 05:22:12.000000', 'Rose', 'EN');
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
        VALUES (1, '2024-03-04 05:22:12.000000', 'Rose', 'EN');
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
        VALUES (1, '2024-03-05 05:22:12.000000', 'Rose', 'EN');
```

What happens when these two sequences run parallel in different sessions – without transaction?

How do we call this phenomenon?

Isolation clearly is violated.

Attention: Client PGAdmin behaves differently. Run these commands in a command line tool.

eva.knirsch@kiu.edu.ge

Transaction Isolation Concept

This time both sequences run **as transactions:**

```
/* client session 1 – Agency pays teacher 1*/
Begin;
select t_payment from teacher where t_id = 1;
/* some application code - calculating how much money teacher gets and doing the money transfer */
select pg_sleep(20);
update teacher set t_payment=0 where t_id = 1;
commit;

/* Client Session 2 – teacher reports back hours  */
Begin;
update teacher set t_payment = t_payment + 3 where t_id = 1;
commit;
```

What do we expect to happen when both sequences run concurrently but as transactions? What happens?

Attention: Client PGAdmin behaves differently. Run these commands in a command line tool.

eva.knirsch@kiu.edu.ge

Concurrent Transactions

Conflicting goals:

Performance
Concurrent transactions significantly increase the performance of a system

Consistency
Concurrent transactions may lead to inconsistency.

Isolation demands that

... the result of concurrent transactions in multi-user mode must be the same as if these transactions had been performed serially in single-user mode.

However,...

Transaction Isolation Levels

- Isolation in ACID exists in different degrees – called isolation levels.
- The strongest isolation level - serializable - guaranetees that the results of concurrently executing transactions are equivalent to the results of executing these transactions in serial order.
- Database systems also offer weaker isolation levels in transactions in order to boost performance.
- An application developer may decide to work with a weaker isolation level either if inconsistencies can be tolerated or if the application itself takes care of consistency issues.
- A database has an isolation level set as default for transactions. The default isolation level usually is a weaker isolation level.
- A higher isolation level needs to be specified, if needed, at the beginning of the transaction.
- Different relational databases support different isolation levels, and the same isolation level can mean different isolation in different databases.

eva.knirsch@kiu.edu.ge

Concurrent Write Operations

Example – this time both sequence run **as transactions with isolation level repeatable read**

```
/* client session 1 – Agency pays teacher */
Begin isolation level repeatable read;
select t_payment from teacher where t_id = 1;
/* some application code */
select pg_sleep(20);
update teacher set t_payment=0 where t_id = 1;
commit;

/* Client Session 2 – teacher reports back hours  */
Begin isolation level repeatable read;
update teacher set t_payment = t_payment + 3 where t_id = 1;
commit;
```

What happens?

Attention: Client PGAdmin behaves differently. Run these commands in a command line tool.

eva.knirsch@kiu.edu.ge

Transaction Isolation

T2 commits
Teacher counter is incremented

T1 errors out with:
ERROR: could not serialize access due to concurrent update

Transaction 1 is rolled back.
lesson=!# Commit;
ROLLBACK

Teacher account balance is NOT reset to 0.

Concurrent Write Transactions Postgres

We run the 2 sequences of again; again in default isolation level. The only difference is that T2 now also does some application code – simulated by select pg_sleep() – **see red line**

```
/* client session 1 – Agency pays teacher 1*/
Begin;
select t_payment from teacher where t_id = 1;
/* some application code - calculating how much money teacher gets and doing the money transfer */
select pg_sleep(20);
update teacher set t_payment=0 where t_id = 1;
commit;

/* Client Session 2 – teacher reports back hours  */
Begin;
update teacher set t_payment = t_payment + 3 where t_id = 1;
SELECT t_payment FROM teacher WHERE t_id = 1;
select pg_sleep(20);
commit;
```


eva.knirsch@kiu.edu.ge

Concurrent Write Transactions Postgres

T1 waits – does not do its write operation - until T2 commits

T1 does not overwrite data written by T2 – but not committed yet.

T2 commits → T1 continues, which leads to Lost Update

eva.knirsch@kiu.edu.ge

Concurent Write Operations Postgres: Dirty Write and Lost Update Anomalies

Isolation Level	Dirty Write Anomaly	Lost Update Anomaly
	overwrite uncommitted data	overwrite committed data
Read Committed	does not happen (is prevented)	happpens (is not prevented)
Repeatable Read	does not happen (is prevented)	does not happen (is prevented)

Lost Update Anomaly

- Two transcactions concurrently perform a read-modify-write cycle.
- One transaction updates data (one or more rows) and commits.
- The second transaction updates the same data and overwrites the committed update without noticing / respecting the update of the first transaction.
- → Update of the first committed transaction is lost.

Concurrent Write Transactions Postgres

We run the 2 sequences of again; this time again in isolation level repeatable read. We know that T1 waits for T2 to commit before it writes. This time, however, T2 rolls back.

– **see red line**

```
/* client session 1 – Agency pays teacher 1*/
Begin;
select t_payment from teacher where t_id = 1;
/* some application code - calculating how much money teacher gets and doing the money transfer */
select pg_sleep(20);
update teacher set t_payment=0 where t_id = 1;
commit;

/* Client Session 2 – teacher reports back hours  */
Begin;
update teacher set t_payment = t_payment + 3 where t_id = 1;
SELECT t_payment FROM teacher WHERE t_id = 1;
select pg_sleep(20);
rollback;
```

Explain the result!

eva.knirsch@kiu.edu.ge

eva.knirsch@kiu.edu.ge

Assignments 3

PostGreSQL Transaction Example

Create the following table:

```
CREATE TABLE public.test
(
    "ID" serial,
    name character varying(30),
    PRIMARY KEY ("ID")
);

ALTER TABLE IF EXISTS public.test
    OWNER to postgres;
```


eva.knirsch@kiu.edu.ge

PostGreSQL Transaction Example

Run the following command sequence, in **PSQL**:

```
START TRANSACTION;
        INSERT into test values (1,'Miller');
        INSERT into test values (2,'Doe');
        ROLLBACK;
START TRANSACTION;
        INSERT into test values (3,'Neuer');
        INSERT into test  values (4,'Herrmann');
        COMMIT;
        INSERT into test values (5,'Niemann');
        INSERT into test  values (6,'Smith');
        rollback;
```

Which rows are in the table after running the sequence?
Explain!

eva.knirsch@kiu.edu.ge

PostGreSQL Transaction Example

- delete the data rows out of table test.
- run the same command sequence in PGAdmin4:

```
START TRANSACTION;
    INSERT into test values (1,'Miller');
    INSERT into test values (2,'Doe');
    ROLLBACK;
    START TRANSACTION;
    INSERT into test values (3,'Neuer');
    INSERT into test  values (4,'Herrmann');
    COMMIT;
    INSERT into test values (5,'Niemann');
    INSERT into test  values (6,'Smith');
    rollback;
```

https://www.postgresql.org/docs/current/tutorial-transactions.html

Some client libraries issue BEGIN and COMMIT commands automatically, so that you might get the effect of transaction blocks without asking. Check the documentation for the interface you are using.

Which rows are in the table after running the sequence?

Never use PGAdmin for transaction tests!

eva.knirsch@kiu.edu.ge

PostgreSQL Autocommit

Most RDBMS know two modes for transaction commits. The modes are controlled by the **autocommit** command.

- MariaDB: show variables like '%autocommit'
- PostgreSQL: \echo :AUTOCOMMIT

```
lesson=# \echo :AUTOCOMMIT
lesson=# \set AUTOCOMMIT off | on
```

In mariadb INNODB and PostgreSQL the default setting is: Autocommit = ON

By default (without BEGIN), PostgreSQL executes transactions in "autocommit" mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done).
https://www.postgresql.org/docs/16/sql-begin.html

eva.knirsch@kiu.edu.ge

PostGreSQL Transaction Example

- Delete the data rows out of table test.
- Set autocommit OFF in command line
- Run the same command sequence again on command line

```
START TRANSACTION;
    INSERT into test values (1,'Miller');
    INSERT into test values (2,'Doe');
    ROLLBACK;
    START TRANSACTION;
    INSERT into test values (3,'Neuer');
    INSERT into test  values (4,'Herrmann');
    COMMIT;
    INSERT into test values (5,'Niemann');
    INSERT into test  values (6,'Smith');
    rollback;
```

Which rows are in the table after running the same sequence in PSQL but with autocommit off?

eva.knirsch@kiu.edu.ge

PostGreSQL Transaction Example

- Delete the data rows out of table test.
- Run the same command sequence again on command line – but without the last rollback command!

```
START TRANSACTION;
    INSERT into test values (1,'Miller');
    INSERT into test values (2,'Doe');
    ROLLBACK;
    START TRANSACTION;
    INSERT into test values (3,'Neuer');
    INSERT into test  values (4,'Herrmann');
    COMMIT;
    INSERT into test values (5,'Niemann');
    INSERT into test  values (6,'Smith');
```

- Check that transaction is still active:
  SELECT * FROM pg_stat_activity WHERE state IN ('active', 'idle in transaction');
- end transaction manually by typing command in PSQL: commit;
- Check, which rows are in the table

eva.knirsch@kiu.edu.ge

PostgreSQL Autocommit

On the server side, PostgreSQL operates in autocommit mode.

On the client side, autocommit mode can be turned off in two ways:

1. Explicitly: By setting autocommit OFF
2. Implicitly: By starting a transaction with a BEGIN statement

If you disable autocommit, Postgres understands all commands as implicit transactions – independent of whether you explicitly start a transaction or not.

But: PostgreSQL will wait for a commit / rollback to finish a transaction.
If that command does not come, the session will remain in the state “idle in transaction”. This means that locks may not be released.

eva.knirsch@kiu.edu.ge

Transactions on Different Rows

- Open 2 sessions in PSQL (command line sessions)
- Run two concurrent transactions that operate on the same table (teacher) but **different rows.**
- Run the transactions in default isolation level

T1: reads out t_payment counter of **teacher 1**, then sleeps for some seconds, then sets payment counter to 0

T2: starts right after T1, increments t_payment counter of **teacher 2** by 3, then commits

- Do both transactions run right through and commit?
- Change isolation level to repeatable read.
- Do both transactions run through and commit?
- What conclusions do you draw regarding PostgreSQL isolation mechanism?

eva.knirsch@kiu.edu.ge

Transactions on Different Columns

- Open 2 sessions in PSQL (command line sessions)
- Run two concurrent transactions that operate on the same table (teacher) on the same table (teacher) and on **the same row** but on **different columns**.
- Run the transactions in **default isolation level**

T1: reads out t_payment counter of teacher 2 , then does some application logic, then resets the counter of teacher 2 to 0

T2 updates t_education of teacher 2, sets t_education = 'Bachelor'

- Do both transactions run right through and commit?
- Can both transactions eventually commit?

eva.knirsch@kiu.edu.ge

Transactions on Different Columns

- Change the isolation level to repeatable read:

T1: reads out t_payment counter of teacher 2 , then does some application logic, then resets the counter of teacher 2 to 0

T2 updates t_education of teacher 2, sets t_education = 'Bachelor'

- Do both transactions run right through and commit?
- Can both transactions eventually commit?

- Let T2 rollback instead of commit. What effect does this have on T1?

eva.knirsch@kiu.edu.ge

Concurrent Write Transactions

Run the 2 sequences again, in default isolation level. The only difference is that T2 now also does some application code – simulated by select pg_sleep() – **see red line**

```
/* client session 1 – Agency pays teacher 1*/
Begin;
select t_payment from teacher where t_id = 1;
/* some application code - calculating how much money teacher gets and doing the money transfer */
select pg_sleep(20);
update teacher set t_payment=0 where t_id = 1;
commit;

/* Client Session 2 – teacher reports back hours  */
Begin;
update teacher set t_payment = t_payment + 3 where t_id = 1;
SELECT t_payment FROM teacher WHERE t_id = 1;
select pg_sleep(20);
commit;
```

Explain the execution.

eva.knirsch@kiu.edu.ge

Isolation Levels – feel free to use AI for your answer.

1. Which different isolation levels does the SQL standard have?

2. What anomalies (consistency problems) do the different levels prevent?

3. Which isolation levels does Postgres offer?

4. Which isolation levels does MySQL / MariaDB offer?

eva.knirsch@kiu.edu.ge