	Lehrveranstaltung	Databases and Information Systems 2020		
	Aufgabenzettel	3		
	STiNE-Gruppe 14	Simon Weidmann, Aram Yesildeniz		
	Ausgabe	12. Mai 2020	Abgabe	19. Mai 2020

### 3.1 Isolation Levels and SQL

a)

*How can you determine the currently set isolation level?*

```
SHOW TRANSACTION ISOLATION LEVEL;
```

*What is the default isolation level of PostgreSQL?*

Read Committed is the default isolation level in PostgreSQL.

*How can the isolation level be changed during a session in PostgreSQL?*

For current transaction:

```
SET TRANSACTION ISOLATION LEVEL
{ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED };
```


For default transaction characteristics:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
{ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED };
```

b)

```
CREATE TABLE public."OPK"
(
    "ID" integer,
    "NAME" text
);
```

```
ALTER TABLE public."OPK"
    OWNER to postgres;
```

	Lehrveranstaltung	<b>Databases and Information Systems 2020</b>		
	Aufgabenzettel	<b>3</b>		
	STiNE-Gruppe 14	<b>Simon Weidmann, Aram Yesildeniz</b>		
	Ausgabe	<b>12. Mai 2020</b>	Abgabe	<b>19. Mai 2020</b>

c)

```
INSERT INTO public."OPK"(
    "ID", "NAME")
VALUES (1, 'shaggy');
```

```
INSERT INTO public."OPK"(
    "ID", "NAME")
VALUES (2, 'fred');
```

```
INSERT INTO public."OPK"(
    "ID", "NAME")
VALUES (3, 'velma');
```

```
INSERT INTO public."OPK"(
    "ID", "NAME")
VALUES (4, 'scooby');
```

```
INSERT INTO public."OPK"(
    "ID", "NAME")
VALUES (5, 'daphne');
```

d)


### Read Committed

Read Committed is an isolation level. Under read committed isolation, the user is only allowed to read committed data. It is guaranteed that the query returns only data which was committed at the time of the read. If a transaction needs to read a row that has been modified by an incomplete transaction in another session, the transaction waits until the first transaction completes (either commits or rolls back). Therefore read committed isolation prevents Dirty Read.

Read committed holds exclusive locks during the whole transaction. Shared locks on the other hand are acquired as late as possible and released as soon as possible. Shared locks are being released at the end of each statement. Therefore, Phantom Reads and Non-Repeatable Reads can occur.

In DB2, the read committed isolation level is called *Cursor Stability* and is the default isolation level.

Example: In the example, the transaction would hold exclusive locks for all write or update queries. Since the transaction will only read data, no exclusive lock will be hold. For the SELECT query, the transaction will hold a shared lock for the OPK table where ID is equal to 3. It will only read the data, if there are no uncommitted changes. After the SELECT statement and before the commit, the transaction will release the shared lock. After the commit, no locks are hold by the query.

	Lehrveranstaltung	<b>Databases and Information Systems 2020</b>		
	Aufgabenzettel	<b>3</b>		
	STiNE-Gruppe 14	<b>Simon Weidmann, Aram Yesildeniz</b>		
	Ausgabe	<b>12. Mai 2020</b>	Abgabe	<b>19. Mai 2020</b>

e)

### Repeatable Read

Repeatable Read guarantees that no item that has been selected the first time can be modified or deleted until the commit. The transaction can repeat the same query, and no rows that have been read by the transaction will have been updated or deleted.

Repeatable Read disallows Dirty Reads, Lost Updates and Non-Repeatable Reads, but Phantom Reads can occur. This means that the set of rows that is returned by two consecutive select queries in a transaction can differ. This happens if another transaction adds or removes rows from the same table.

Under Repeatable Read isolation level, shared locks are placed on all data read by each statement in the transaction and are held until the transaction completes. This prevents other transactions from modifying any rows that have been read by the current transaction. Shared locks are held to the end of a transaction instead of being released at the end of each statement.

Example: By the start of the transaction, a shared lock will be held for the row where ID = 3. The shared lock will only be released after the commit. This means that no other transaction can modify the data from the selected row. During the transaction, other queries that add or remove rows to the table may still occur.

## 3.2 Lock Conflicts

a)


- What happens? What is the output of Connection 1?

Repeatable read ensures that dirty reads, lost updates and non-repeatable reads are not possible. Phantom reads can still occur. Therefore, the select statement from T1 will lock the items and data from the selected rows can not be modified. The output of Connection 1 is:

```
dis=# select * from "OPK" where "ID" > 3;
 ID |  NAME
----+-----
  4 | scooby
  5 | daphne
(2 rows)
```

- Compare the state before the transactions with the state after the transactions.

After both transactions, the table has an additional row:

	Lehrveranstaltung	Databases and Information Systems 2020		
	Aufgabenzettel	3		
	STiNE-Gruppe 14	Simon Weidmann, Aram Yesildeniz		
	Ausgabe	12. Mai 2020	Abgabe	19. Mai 2020

```
dis=# select * from "OPK";
 ID |  NAME
----+-----
  1 | shaggy
  2 | fred
  3 | velma
  4 | scooby
  5 | daphne
  6 | scrappy
(6 rows)
```

- What can be observed if Connection 1 commits and execute its SQL command again?

The row added by T2 is visible:

```
 ID |  NAME
----+-----
  4 | scooby
  5 | daphne
  6 | scrappy
(3 rows)
```

- Can we observe a Canonical Synchronization Problem? If yes, explain which one and why it appears.

Phantom read occurs. This means that although T1 locks the data selected by the query, in parallel, rows can be added to the table. The set of rows that are satisfying the condition by T1 has changed.

With postgresSQL, if T1 executes the select statement several times before a commit, and T2 adds a row in parallel, the data returned will not change and the added row is not visible for T1.


From the docs: *REPEATABLE READ: All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.*

b)

Within the Serializable isolation level, all anomalies are prohibited and concurrent transactions run as they would run sequentially one by one in order.

- What can be observed now? What is the output of Connection 1?

The created row from T2 will be added to the table. If we SELECT in T1 again before T1 committed and after T2 committed, the added row is not visible. If we run T1 after it already committed again, the output is:

	Lehrveranstaltung	<b>Databases and Information Systems 2020</b>		
	Aufgabenzettel	<b>3</b>		
	STiNE-Gruppe 14	<b>Simon Weidmann, Aram Yesildeniz</b>		
	Ausgabe	<b>12. Mai 2020</b>	Abgabe	<b>19. Mai 2020</b>

```

ID | NAME
----+-----
 4 | scooby
 5 | daphne
 6 | scrappy
(3 rows)

```

- Compare the state before the transactions with the state after the transactions.

After both transactions, the table has an additional row:

```

dis=# select * from "OPK";
ID | NAME
----+-----
 1 | shaggy
 2 | fred
 3 | velma
 4 | scooby
 5 | daphne
 6 | scrappy
(6 rows)

```

- Can we observe a Canonical Synchronization Problems? If yes, explain which one and why it appears.

No Canonical Synchronization Problems can be observed.. This is expected as the isolation level Serializable is defined by this behaviour.

c)

- In this scenario Connection 2 has to wait until Connection 1 commits. Explain why.

T2 does not have to wait (?)


- Discuss, what lock can be potentially encountered on the table OPK? Which Connection do the locks belong to?

d)

```

CREATE TABLE public."MPK"
(
  "ID" integer,
  "NAME" text,
  PRIMARY KEY ("ID")
);

```

	Lehrveranstaltung	<b>Databases and Information Systems 2020</b>		
	Aufgabenzettel	<b>3</b>		
	STiNE-Gruppe 14	<b>Simon Weidmann, Aram Yesildeniz</b>		
	Ausgabe	<b>12. Mai 2020</b>	Abgabe	<b>19. Mai 2020</b>

```
ALTER TABLE public."MPK"
  OWNER to postgres;
```

```
INSERT INTO public."MPK"(
  "ID", "NAME")
VALUES (1, 'shaggy');
```

```
INSERT INTO public."MPK"(
  "ID", "NAME")
VALUES (2, 'fred');
```

```
INSERT INTO public."MPK"(
  "ID", "NAME")
VALUES (3, 'velma');
```

```
INSERT INTO public."MPK"(
  "ID", "NAME")
VALUES (4, 'scooby');
```

```
INSERT INTO public."MPK"(
  "ID", "NAME")
VALUES (5, 'daphne');
```

e)