Exercise 3 Report

Gruppe 16

Anastasiia Rubanovych Sebastian Funck

May 19, 2020

3.1 Isolation Levels and SQL

a)

- **How can you determine the currently set isolation level?**
  The default isolation level of a database is configured inside the `postgresql.conf` with the attribute `default_transaction_isolation`. It can be queried via `SHOW default_transaction_isolation`.
- **What is the default isolation level of PostgreSQL?**
  The default isolation level of PostgreSQL is `read committed`.
- **How can the isolation level be changed during a session in PostgreSQL?**
  `SET SESSION CHARACTERISTICS AS TRANSACTION { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }`

b)

```
create table OPK (
  ID   int4,
  NAME varchar(64)
)
```

c)

```
insert into OPK (ID, NAME)
    VALUES (1, 'shaggy'), (2, 'fred'),
    (3, 'velma'), (4, 'scooby'), (5, 'daphne');
```

d)

- **Discuss what locks you would expect are held at this point (and before a commit) with `Read Committed`?**
  We would expect 1 shared table lock and 1 row lock to be hold before and 0 table locks and 0 row locks to be hold after the commit. `Cursor Stability (Read Committed)` never holds more than 1 row lock.

- **Discuss what locks you would expect are held at this point (and before a commit) with `Repeatable Read`?**
  We would expect 1 table lock to be hold before and 0 locks after the commit. `Repeatable Read` locks the complete table until commit.

3.2 Lock Conflicts

a)

- **What happens? What is the output of Connection 1?**
  While Connection 1 is executing the query, Connection 2 inserts a row that matches the query of Connection 1. This results in Connection 1 commiting after Connection 2, but will not select the newly inserted item. Therefor the output of Connection 1 is

```
(4, 'scooby')
(5, 'daphne')
```

- **Compare the state before the transactions with the state after the transactions.**
  After the commit of Connection 2 one new row has been added to the OPK table. This happens while Connection 1 is still executing.

- **What can be observed if Connection 1 commits and execute its SQL command again?**
  The output of Connection 1 becomes

```
(4, 'scooby')
(5, 'daphne')
(6, 'scrappy')
```

- **Can we observe a Canonical Synchronization Problem? If yes, explain which one and why it appears.**
  Yes, we can observe a Phantom Read because Connection 1 is querying a range while Connection 2 inserts a record (`6, \`scrappy\``) into the table which would match that queried range.

b)

- **What happens? What is the output of Connection 1?**
  Because Connection 1 is set to isolation mode `Serializable` the scheduler has to ensure that this transactions is executed in a serializable linear order. Therefor Connection 2 has to wait for Transaction 1 to finish.

```
(4, 'scooby')
(5, 'daphne')
```

- **Compare the state before the transactions with the state after the transactions.**
  After the transaction of Connection 1 there are only 5 rows in the data set. Only after connection 2 has been completed, the 6th row is added.

- **What can be observed if Connection 1 commits and execute its SQL command again?**
  The output of Connection 1 stays the same

```
(4, 'scooby')
(5, 'daphne')
(6, 'scrappy')
```

- **Can we observe a Canonical Synchronization Problem? If yes, explain which one and why it appears.**
  By definition of isolation level `Serializable` there can't occour any canonical synchronization problems.

c)

- **In this scenario Connection 2 has to wait until Connection 1 commits. Explain why.**
  `ID=1` is in the context of table `OKP` a range based query, because multiple rows could match this criteria. In addition Connection 1 is in isolation level `Serializable`. This results in Connection 1 holding an R-lock on the complete table `OKP` and Connection 2 waiting for Connection 1 to commit.

- **Discuss, what lock can be potentially encountered on the table OPK? Which Connection do the locks belong to?**
  Connection 1 is likely to hold an R lock on the table `OKP`.

d)

```
create table MPK (
        id int4,
        name varchar(64),
```

```
        CONSTRAINT mpk_pk PRIMARY KEY (id)
);

insert into MPK (ID, NAME)
        VALUES (1, 'shaggy'), (2, 'fred'),
        (3, 'velma'), (4, 'scooby'), (5, 'daphne');
```

e)

With `ID` being a `PRIMARY KEY`, every row is uniqly identifiable by its id. This means the query of Connection 1 is no longer range-query and enables Connection 1 to hold a narrower lock (R-lock on row `ID:3`). On the otherhand Connection 2 now can hold an R-lock on row `ID:3` while Connection 1 is executing and doesn't have to wait for it to finish.