Exercise 3

Nasrul Huda, Aldo Sula

May 14, 2025

1 3. Synchronization with Locking Protocols

1.1 3.1. Isolation Levels

a) The current isolation level is **READ COMMITTED**.The isolation levels supported by the PostgreSQL are:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

b) Sample table named sheet3 with columns id and name is created. Sample data is inserted into the table.

c) The autocommit is set to off with the command

```
\set AUTOCOMMIT off;
```

Query one row from table sheet3 and find out the currently held locks:

Result of querying a row and checking locks:

```
SELECT * FROM sheet3 WHERE id = 1;
```

```
 id | name
----+------
  1 | John
```

```
SELECT relation::regclass, mode, granted
FROM pg_locks
WHERE relation::regclass = 'sheet3'::regclass;
```

The result shows no locks on the sheet3 table. This is because in READ COMMITTED isolation level, PostgreSQL acquires row-level read locks during query execution but releases them immediately after the statement completes. Since our SELECT query finished before we checked for locks, any locks that were held had already been released.

d) The transaction is started with the command

```
-- Transaction 1
BEGIN;
SELECT * FROM sheet3 WHERE id = 1;
COMMIT;
```

– Serializable Transaction

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM sheet3 WHERE id = 1;
COMMIT;
```

Steps repeated for the serializable transaction:

The locks held by the transaction are:

```
relation |       mode        | granted
---------+-------------------+---------
sheet3   | AccessShareLock   | t
sheet3   | SIReadLock        | t
(2 rows)
```

Locks held by the transaction:

- Serializable Read Lock

Serializable transaction acquires a row-level read lock on the sheet3 table and holds it until the transaction is committed.

1.2 3.2 Lock Conflicts

a) First connection with isolation level **READ COMMITTED** and **AUTOCOMMIT off**

```
SELECT * FROM sheet3 WHERE id > 3;
```

```
id |  name
---+--------
 4 | Nasrul
 5 | Raj
 6 | Aldo
 7 | Alia
 8 | Ayesha
(5 rows)
```

Second connection with isolation level **READ COMMITTED** and **AUTOCOMMIT on**

Inserts a new row into the sheet3 table;

```
INSERT INTO sheet3 (id, name) VALUES (9, 'Bill');
```

First connection repeating the query;

```
SELECT * FROM sheet3 WHERE id > 3;
```

```
 id |  name
----+--------
 4  | Nasrul
 5  | Raj
 6  | Aldo
 7  | Alia
 8  | Ayesha
 9  | Bill
(6 rows)
```

The new row is visible. Even though the first connected has not yet committed, when the SELECT query is run again in the same transaction, the new row is visible. This is because the second connection has committed the new row and the first connection is in READ COMMITTED isolation level.

b) First connection's isolation level is set to **REPEATABLE READ** with **AUTOCOMMIT off**

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

Repeats the query:

```
SELECT * FROM sheet3 WHERE id > 3;
```

```
 id |  name
----+--------
  4 | Nasrul
  5 | Raj
  6 | Aldo
  7 | Alia
  8 | Ayesha
  9 | Bill
(6 rows)
```

Whereas in the second connection, a new row is inserted at the same time:

```
INSERT INTO sheet3 (id, name) VALUES (10, 'Chris');
```

Repeats the query in the first connection:

```
SELECT * FROM sheet3 WHERE id > 3;
```

```
 id |   name
----+--------
  4 | Nasrul
  5 | Raj
  6 | Aldo
  7 | Alia
  8 | Ayesha
  9 | Bill
(6 rows)
```

The new row is not visible to the first connection. This is because the first connection is in REPEATABLE READ isolation level and the second is in READ COMMITTED isolation level. The second connection has committed the new row but the first connection has not yet committed.

REPEATABLE READ takes a snapshot of the database at the start of the transaction, not at the start of each statement (unlike READ COMMITTED)

The locks held before the commit of the first transaction are:

```
SELECT relation::regclass, mode, granted
FROM pg_locks
WHERE relation::regclass = 'sheet3'::regclass;
```

```
 relation | mode | granted
----------+------+---------
sheet3    | AccessShareLock | t
sheet3    | SIReadLock      | t
(2 rows)
```

After the commit of the first transaction, the newly inserted row is visible in the first transaction.

```
 id |   name
----+--------
  4 | Nasrul
  5 | Raj
  6 | Aldo
  7 | Alia
  8 | Ayesha
  9 | Bill
 10 | Chris
(7 rows)
```

Would the same thing happen if PostgreSQL was using strict 2PL?

No, it would be different

Why? Here is what will happen:

- When Session 1 reads `id > 3`, it would **lock that range**.
- Then, if Session 2 tries to insert a row like `id = 5`, it would have to **wait**.
- Session 2 can't continue until Session 1 is done (commits or rolls back).

In PostgreSQL, this doesn't happen. Session 2 is allowed to insert right away, and Session 1 doesn't see the new row because it's using an old snapshot.

So, **2PL would block**, but PostgreSQL's method just avoids the problem by hiding the new data until the first transaction finishes.

c) Update one row in the first connection with isolation level **REPEATABLE READ** and **AUTOCOMMIT off**

```
UPDATE sheet3 SET name = 'Brock' WHERE id = 1;
```

Haven't committed the transaction yet. While in the second connection, another row is updated with id 2:

```
UPDATE sheet3 SET name = 'Shayne' WHERE id = 2;
```

And again in the second connection, the same row with id = 1 is updated with name = 'Shayne':

```
UPDATE sheet3 SET name = 'Shayne' WHERE id = 1;
```

The transaction will now be blocked, because the first connection has not yet committed.

The isolation level does not matter for this particular blocking behavior. Whether using READ COMMITTED or REPEATABLE READ, PostgreSQL always prevents concurrent updates to the same row (this prevents "lost updates").

Only after connection 1 commits or rolls back, releasing its lock on row id=1, will connection 2's update proceed. At that point, connection 2's update will overwrite connection 1's changes to that row.

d) When does PostgreSQL abort or rollback a transaction?

We tested different situations using two connections with different isolation levels. Some actions caused a rollback or abort. Below are the main cases, with explanations on why PostgreSQL chooses to abort in each case.

(a) **Serializable Read/Write Conflict**

When two transactions use `SERIALIZABLE` isolation, PostgreSQL tries to make sure the final result looks like the transactions ran one after the other. If one transaction reads a row and the other updates it, PostgreSQL cannot guarantee that order anymore. So, it aborts one of them.

Example:

- Session 1 reads `id = 1`.
- Session 2 updates `id = 1` and commits.
- Session 1 tries to update the same row and commit.
- **Why it fails:** PostgreSQL detects that both transactions changed the same data and can no longer keep the isolation. It aborts Session 1 to avoid incorrect results.

(b) **Serializable Phantom Insert**

In `SERIALIZABLE`, PostgreSQL tracks ranges of data read by a transaction. If another transaction inserts a new row in that range, it creates a conflict (called a "phantom"). PostgreSQL may abort one of the transactions to prevent this.

Example:

- Session 1 selects `id > 3`.
- Session 2 inserts `id = 5` and commits.
- Session 1 inserts `id = 6` and tries to commit.
- **Why it fails:** The insert in Session 2 creates a new row in the range Session 1 had read. PostgreSQL sees this as a conflict and rolls back Session 1.

(c) **Serializable vs Read Committed Range Conflict**

When one transaction uses `SERIALIZABLE` and reads a range of rows, and another (with lower isolation) writes to that range, the `SERIALIZABLE` transaction might fail at commit.

Example:

- Session 1 selects `id > 10`.
- Session 2 (in `READ COMMITTED`) inserts `id = 11` and commits.
- Session 1 tries to insert `id = 12` and commit.
- **Why it fails:** The write from Session 2 breaks the serializability of Session 1's view. PostgreSQL aborts Session 1 to keep the transactions in a valid order.

(d) **Deadlock**

A deadlock happens when two transactions each hold a lock and try to get the one the other is holding. PostgreSQL will notice this situation and abort one of the transactions to break the loop.

Example:

- Session 1 updates `id = 1`.
- Session 2 updates `id = 2`.
- Session 1 then tries to update `id = 2`, while Session 2 tries to update `id = 1`.
- **Why it fails:** Each session is waiting on the other. PostgreSQL aborts one to avoid an endless wait.

(e) **Serializable vs Repeatable Read Read/Write Conflict**

This conflict happens when a `SERIALIZABLE` transaction reads a row that a `REPEATABLE READ` transaction later modifies and commits. If the `SERIALIZABLE` transaction then tries to update that row too, PostgreSQL may detect an unsafe dependency and abort it.

Example:

- Session 1 uses `SERIALIZABLE` and reads `id = 2`.
- Session 2 uses `REPEATABLE READ` and updates `id = 2`, then commits.
- Session 1 then tries to update `id = 2` and commit.
- **Why it fails:** PostgreSQL sees that the data read by Session 1 was also changed by another transaction. This could lead to non-serial behavior, so PostgreSQL aborts Session 1.