Bazy danych 2022

Piotr Wieczorek

25 maja 2022

Transactions

```
START TRANSACTION [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ
                    | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE

ROLLBACK;
COMMIT;
```

Transactions

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM sometable;
...
COMMIT;
```

Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

CC in PostgreSQL: Multiversion Concurrency Control (MVCC)

- https://www.postgresql.org/docs/current/transaction-iso.html

CC in PostgreSQL: Multiversion Concurrency Control (MVCC)

- `https://www.postgresql.org/docs/current/transaction-iso.html`
- A Critique of ANSI SQL Isolation Levels. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. ACM-SIGMOD Conference on Management of Data, June 1995.
- Serializable Snapshot Isolation in PostgreSQL. D. Ports and K. Grittner. VLDB Conference, August 2012.
- Serializable isolation for snapshot databases. ACM Trans. Database Syst. 34(4): 20:1-20:42 (2009) Michael J. Cahill, Uwe Röhm, Alan D. Fekete:

CC in PostgreSQL

- dirty read
- nonrepeatable read
- phantom read
- serialization anomaly

Table 13.1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Read Committed

- default isolation level in PostgreSQL

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.
- SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.
- SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- two successive SELECT commands can see different data, even though they are within a single transaction

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.
- SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- two successive SELECT commands can see different data, even though they are within a single transaction
- UPDATE & DELETE commands behave the same as SELECT in terms of searching for target rows, but ...

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.
- SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- two successive SELECT commands can see different data, even though they are within a single transaction
- UPDATE & DELETE commands behave the same as SELECT in terms of searching for target rows, but ...
- if such target row has already been updated (or deleted):

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.
- SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- two successive SELECT commands can see different data, even though they are within a single transaction
- UPDATE & DELETE commands behave the same as SELECT in terms of searching for target rows, but ...
- if such target row has already been updated (or deleted):
  - wait for the first updating transaction to commit or roll back

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.
- SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- two successive SELECT commands can see different data, even though they are within a single transaction
- UPDATE & DELETE commands behave the same as SELECT in terms of searching for target rows, but ...
- if such target row has already been updated (or deleted):
  - wait for the first updating transaction to commit or roll back
  - if it rolls back, then its effects are negated and the second updater can proceed

Read Committed

- default isolation level in PostgreSQL
- a SELECT query sees a snapshot of the database as of the instant the query begins to run.
- SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- two successive SELECT commands can see different data, even though they are within a single transaction
- UPDATE & DELETE commands behave the same as SELECT in terms of searching for target rows, but ...
- if such target row has already been updated (or deleted):
  - wait for the first updating transaction to commit or roll back
  - if it rolls back, then its effects are negated and the second updater can proceed
  - if it commits, ignore the row if it is deleted, otherwise apply its operation to the updated version. Re-evaluate the WHERE clause to see if the updated version of the row still matches.

Read Committed

```
BEGIN;
UPDATE accounts SET balance = balance + 10.00 WHERE acctnum = 11;
UPDATE accounts SET balance = balance - 10.00 WHERE acctnum = 22;
COMMIT;
```

- It is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects of those commands on other rows in the database.

Read Committed

```
BEGIN;
UPDATE accounts SET balance = balance + 10.00 WHERE acctnum = 11;
UPDATE accounts SET balance = balance - 10.00 WHERE acctnum = 22;
COMMIT;
```

- It is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects of those commands on other rows in the database.
- Is is good or bad? What if two such transactions concurrently try to change the balance of account 11?

Read Committed

```
BEGIN;
UPDATE accounts SET balance = balance + 10.00 WHERE acctnum = 11;
UPDATE accounts SET balance = balance - 10.00 WHERE acctnum = 22;
COMMIT;
```

- It is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects of those commands on other rows in the database.
- Is is good or bad? What if two such transactions concurrently try to change the balance of account 11?
- We clearly want the second transaction to start with the updated version of the account's row. Because each command is affecting only a predetermined row, letting it see the updated version of the row does not create any troublesome inconsistency.

Read Committed

Assume `website` is a two-row table with `website.hits` equaling 9 and 10:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

Read Committed

Assume `website` is a two-row table with `website.hits` equaling 9 and 10:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

- DELETE will have no effect even though there is a website.hits = 10 row before and after the UPDATE!

Read Committed

Assume `website` is a two-row table with `website.hits` equaling 9 and 10:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

- DELETE will have no effect even though there is a website.hits = 10 row before and after the UPDATE!

Read Committed mode is adequate for many applications, and this mode is fast and simple to use; however, it is not sufficient for all cases.

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read
- Updates will only find target rows that were committed as of the transaction start time.

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read
- Updates will only find target rows that were committed as of the transaction start time.
- If target is updated (or deleted) by another transaction:

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read
- Updates will only find target rows that were committed as of the transaction start time.
- If target is updated (or deleted) by another transaction:
  - wait for the first updating transaction to commit or roll back

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read
- Updates will only find target rows that were committed as of the transaction start time.
- If target is updated (or deleted) by another transaction:
  - wait for the first updating transaction to commit or roll back
  - if it rolls back, then its effects are negated—can proceed

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read
- Updates will only find target rows that were committed as of the transaction start time.
- If target is updated (or deleted) by another transaction:
  - wait for the first updating transaction to commit or roll back
  - if it rolls back, then its effects are negated—can proceed
  - but if it commits: roll back!

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read
- Updates will only find target rows that were committed as of the transaction start time.
- If target is updated (or deleted) by another transaction:
  - wait for the first updating transaction to commit or roll back
  - if it rolls back, then its effects are negated—can proceed
  - but if it commits: roll back!

Repeatable Reads (Snapshot Isolation)

- Each query sees a snapshot as of the start of the first non-transaction-control statement in the transaction; successive SELECT commands within a single transaction see the same data
- it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.
- Stronger guarantee than is required by the SQL standard: prevents phantom read
- Updates will only find target rows that were committed as of the transaction start time.
- If target is updated (or deleted) by another transaction:
  - wait for the first updating transaction to commit or roll back
  - if it rolls back, then its effects are negated—can proceed
  - but if it commits: roll back!

In Repeatable Reads each transaction sees a completely stable view of the database. However, this view will not necessarily always be consistent with some serial execution (can a summary but not individual changes).

Repeatable Read vs. Serializable: Serialization anomaly

```
-- mytab
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

- Transaction A computes `SELECT SUM(value) FROM mytab WHERE class = 1;`

Repeatable Read vs. Serializable: Serialization anomaly

```
-- mytab
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

- Transaction A computes `SELECT SUM(value) FROM mytab WHERE class = 1;`
- then inserts the result (30) as the value in a new row with class = 2.

Repeatable Read vs. Serializable: Serialization anomaly

```
-- mytab
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

- Transaction A computes `SELECT SUM(value) FROM mytab WHERE class = 1;`
- then inserts the result (30) as the value in a new row with class = 2.
- Concurrently, serializable transaction B computes:
  `SELECT SUM(value) FROM mytab WHERE class = 2;`

Repeatable Read vs. Serializable: Serialization anomaly

```
-- mytab
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

- Transaction A computes `SELECT SUM(value) FROM mytab WHERE class = 1;`
- then inserts the result (30) as the value in a new row with class = 2.
- Concurrently, serializable transaction B computes:
  `SELECT SUM(value) FROM mytab WHERE class = 2;`
- then inserts the result (300) as the value in a new row with class = 1.

Repeatable Read vs. Serializable: Serialization anomaly

```
-- mytab
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

- Transaction A computes `SELECT SUM(value) FROM mytab WHERE class = 1;`
- then inserts the result (30) as the value in a new row with class = 2.
- Concurrently, serializable transaction B computes:
  `SELECT SUM(value) FROM mytab WHERE class = 2;`
- then inserts the result (300) as the value in a new row with class = 1.
- Then both transactions try to commit. What happens?

Repeatable Read vs. Serializable: Serialization anomaly

```
-- mytab
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

- Transaction A computes `SELECT SUM(value) FROM mytab WHERE class = 1;`
- then inserts the result (30) as the value in a new row with class = 2.
- Concurrently, serializable transaction B computes:
  `SELECT SUM(value) FROM mytab WHERE class = 2;`
- then inserts the result (300) as the value in a new row with class = 1.
- Then both transactions try to commit. What happens?
- `ERROR:  could not serialize access due to read/write dependencies...`

Repeatable Read vs. Serializable: Serialization anomaly

```
-- mytab
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

- Transaction A computes `SELECT SUM(value) FROM mytab WHERE class = 1;`
- then inserts the result (30) as the value in a new row with class = 2.
- Concurrently, serializable transaction B computes:
  `SELECT SUM(value) FROM mytab WHERE class = 2;`
- then inserts the result (300) as the value in a new row with class = 1.
- Then both transactions try to commit. What happens?
- `ERROR:  could not serialize access due to read/write dependencies...`
- To guarantee true serializability PostgreSQL uses predicate locking: no deadlocks, just monitoring.

Serializable

```
BEGIN TRANSACTION

UPDATE Duties SET Status = 'reserve'
 WHERE DoctorId = :D
  AND Shift = :S
  AND Status = 'on duty'

SELECT COUNT(DISTINCT DoctorId) INTO tmp
 FROM Duties
WHERE Shift = :S
 AND Status = 'on duty'

IF (tmp = 0) THEN ROLLBACK ELSE COMMIT
```

Serializable: some hints

- Declare transactions as READ ONLY when possible.

Serializable: some hints

- Declare transactions as READ ONLY when possible.
- Don’t put more into a single transaction than needed for integrity purposes.

Serializable: some hints

- Declare transactions as READ ONLY when possible.
- Don’t put more into a single transaction than needed for integrity purposes.
- Don’t leave connections dangling “idle in transaction” longer than necessary.

Serializable: some hints

- Declare transactions as READ ONLY when possible.
- Don’t put more into a single transaction than needed for integrity purposes.
- Don’t leave connections dangling “idle in transaction” longer than necessary.
- More: `https://www.postgresql.org/docs/current/transaction-iso.html`