# Isolation Levels in PostgreSQL

November 2024

# Phenomena

| | |
|---|---|
| Dirty read | A transaction reads data written by a concurrent uncommitted transaction. |
| Non-repeatable read | A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read). |
| Phantom read | A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction. |
| Serialization anomaly | The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time. |

# Transaction Isolation Levels

| Isolation Level | Dirty Read | Non-repeatable 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

- The default isolation level in pg
- A SELECT query sees only data committed before the query began and never sees either uncommitted data or changes committed during query execution by concurrent transactions.

# Read Committed

Transaction A

```
begin;

update wallets
set balance = balance + 10000
where id = 1;

commit;
```

Transaction B

```
begin;

update wallets
set balance = balance + 20000
where id = 1;

commit;
```

# Read Committed

- If a target row found by a query while executing an UPDATE statement (or DELETE or SELECT FOR UPDATE) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback.

- If the first updater rolls back, then its effects are negated and the second updater can proceed with updating the originally found row.

- If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row.

```python
with transaction.atomic():
    wallet = Wallet.objects.get(id=wallet_id)
    wallet.balance += amount
    wallet.save()
```

- race condition is available
- performs a select and an update query

```python
with transaction.atomic():
    Wallet.objects.get(id=wallet_id).update(
        balance=F("balance") + amount,
    )
```

- no race condition
- performs an update query

```python
with transaction.atomic():
    wallet = Wallet.objects.select_for_update().get(
        id=wallet_id,
    )
    wallet.balance += amount
    wallet.save()
```

- no race condition
- performs a select for update and an update query
- use when explicit row locking is required for business logic or the logic involves multiple complex operations

# Read Committed

| Isolation Level | Dirty Read | Non-repeatable 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 |

# Repeatable Read

- The Repeatable Read isolation level only sees data committed before the transaction began; it never sees either uncommitted data or changes committed by concurrent transactions during the transaction's execution.

- If a target row found by a query while executing an UPDATE statement (or DELETE or SELECT FOR UPDATE) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback.

# Repeatable Read

- If the first updater rolls back, then its effects are negated and the repeatable read transaction can proceed with updating the originally found row.

- If the first updater commits (and actually updated or deleted the row, not just locked it) then the repeatable read transaction will be rolled back with the message *"ERROR:  could not serialize access due to concurrent update"*.

- When an application receives this error message, it should abort the current transaction and retry the whole transaction from the beginning.

# Read Committed

| Isolation Level | Dirty Read | Non-repeatable 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 |

# Serializable

- this isolation level works exactly the same as Repeatable Read except that it also monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transactions.

- If either transaction were running at the Repeatable Read isolation level, both would be allowed to commit; but since there is no serial order of execution consistent with the result, using Serializable transactions will allow one transaction to commit and will roll the other back with this message "*ERROR: could not serialize access due to read/write dependencies among transactions*"

# Serializable

| Isolation Level | Dirty Read | Non-repeatable 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 |

# Resources:

https://www.postgresql.org/docs/current/transaction-iso.html
https://www.youtube.com/watch?v=4EajrPgJAk0