Lab: SQL Transaction Isolation Levels (PostgreSQL)

Goal

In this lab you will **observe the effects of different transaction isolation levels** in PostgreSQL. You will work with **two concurrent sessions** to see how isolation levels affect consistency, concurrency, and correctness.

Prerequisites

- PostgreSQL
- Two database sessions (e.g. two terminals using `psql`)
  - **Session A** = Reader
  - **Session B** = Writer

Setup (run once)

Execute the following SQL in **one session**:

```
DROP TABLE IF EXISTS iso_accounts;
DROP TABLE IF EXISTS iso_orders;

CREATE TABLE iso_accounts (
    id      int PRIMARY KEY,
    balance int NOT NULL
);

CREATE TABLE iso_orders (
    id     serial PRIMARY KEY,
    amount int NOT NULL
);

INSERT INTO iso_accounts (id, balance) VALUES (1, 100);
INSERT INTO iso_orders (amount) VALUES (50), (120);
```

Exercise 1 – Non-repeatable read (READ COMMITTED)

Session A

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT balance FROM iso_accounts WHERE id = 1;
-- Expected: 100
```

Session B

```
BEGIN;
UPDATE iso_accounts SET balance = 150 WHERE id = 1;
COMMIT;
```

Session A (again)

```
SELECT balance FROM iso_accounts WHERE id = 1;
-- Expected: 150
COMMIT;
```

Observation

- The same query inside one transaction returns **different values**.
- This is a **non-repeatable read**.

☑ What changed between the two reads?

Exercise 2 – Repeatable read (REPEATABLE READ)

Session A

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

SELECT balance FROM iso_accounts WHERE id = 1;
-- Expected: 150
```

Session B

```
BEGIN;
UPDATE iso_accounts SET balance = 200 WHERE id = 1;
COMMIT;
```

Session A (again)

```
SELECT balance FROM iso_accounts WHERE id = 1;
-- Expected: still 150
COMMIT;
```

Observation

- The value does **not change** within the transaction.
- REPEATABLE READ provides a **stable snapshot**.

☑ Why does Session A not see Session B's update?

Exercise 3 – Phantom read (READ COMMITTED)

Session A

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT COUNT(*) FROM iso_orders WHERE amount >= 100;
-- Expected: 1
```

Session B

```
BEGIN;
INSERT INTO iso_orders (amount) VALUES (130);
COMMIT;
```

Session A (again)

```
SELECT COUNT(*) FROM iso_orders WHERE amount >= 100;
-- Expected: 2
COMMIT;
```

Observation

- The same query returns **more rows**.
- This is a **phantom read**.

☑ What changed: existing rows or the result set?

Exercise 4 – Stable result set (REPEATABLE READ)

Session A

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

SELECT COUNT(*) FROM iso_orders WHERE amount >= 100;
-- Expected: 2
```

Session B

```
BEGIN;
INSERT INTO iso_orders (amount) VALUES (140);
COMMIT;
```

Session A (again)

```
SELECT COUNT(*) FROM iso_orders WHERE amount >= 100;
-- Expected: still 2
COMMIT;
```

Observation

- The result set remains stable inside the transaction.

☑ Why does Session A not see the newly inserted row?

Exercise 5 – SERIALIZABLE and transaction retries

Session A

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT balance FROM iso_accounts WHERE id = 1;
UPDATE iso_accounts SET balance = balance + 10 WHERE id = 1;
```

Session B

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

UPDATE iso_accounts SET balance = balance + 10 WHERE id = 1;
COMMIT;
```

Session A

```
COMMIT;
```

Observation

- One transaction may fail with: `ERROR: could not serialize access due to read/write dependencies among transactions`

☑ Why must applications using SERIALIZABLE be prepared to retry transactions?

Key Takeaways

- Higher isolation levels provide **stronger guarantees**
- Stronger guarantees reduce **concurrency and performance**
- `SERIALIZABLE` is safest, but also **most expensive**
- Choosing an isolation level is a **design decision**

Cleanup (optional)

```
DROP TABLE IF EXISTS iso_accounts;
DROP TABLE IF EXISTS iso_orders;
```