Isolation and Recovery

Database Administration
Lab Guide 5

2024/2025

Consider an improved version of the simplified invoice processing system (`skelbench-v2`), adding basic inventory management (*italics* denote changes i.r.t. the previous version):

Client: Id, Name, Address, Data.

Product: Id, Description, Price, Ref, Data, *Stock*, *Min*, *Max*.

Invoice: Id, ~~*ProductId*~~, ClientId, ~~*Price*~~, Data.

***InvoiceLine*:** *Id*, *InvoiceId*, *ProductId*, *Price*.

***Orders*:** *Id*, *ProductId*, *Items*, *Supplier*, *Data*.

Evaluate the impact of the isolation level and logging configuration in the performance of a transactional workload. Unless otherwise stated, execute the workload with a low scale (`-s 9`) and multiple clients (`-c 16`).

Steps

1. Check the database error log while the benchmark is running (check the appendix for more information). Observe the errors reported.
2. Use automatically generated primary keys (with `SERIAL`[1]) for the *Invoice* table. Update both the table creation and the `addInvoice` statement. Re-run the benchmark and observe the log again.
3. Observe the sporadic deadlock error in the log. Update the `sell()` transaction to fix it.
4. To reduce the number of collisions in the `order()` transaction, optimize the `getLowStockProduct` statement to select a random low-stock product instead of the one with the smallest id.
5. Re-run the benchmark with different isolation levels[2] (update the last line of the `Workload` constructor). Test with `TRANSACTION_READ_COMMITTED`, `TRANSACTION_REPEATABLE_READ`, and `TRANSACTION_SERIALIZABLE`. For each one, observe the database log.

[1] `https://www.postgresql.org/docs/17/datatype-numeric.html#DATATYPE-SERIAL`

[2] `https://www.postgresql.org/docs/17/transaction-iso.html`

6. For each isolation level, run the benchmark for 1, 2, 4, 8, 16 client threads. Plot response time vs. throughput and rollbacks vs. throughput.

7. Examine each transaction and determine the minimum isolation required for correctness.[3] At the start of each transaction, set the isolation level to the target (i.e., `c.setTransactionIsolation(...);`). Re-plot the scalability curves.

8. Evaluate the performance with different WAL and checkpoint configurations.[4] [5] Consider longer runtimes to collect more accurate results (e.g., 3 minutes).

Questions

1. What is the impact of the isolation level?

2. What concurrency hot-spots are responsible for observed rollbacks?

3. What strategies can be used to overcome each of these hot-spots?

4. If there was no `check (stock > 0)` constraint, and instead the `sell()` transaction first read the stock and, if greater than 0, decrement it, what would be the minimum isolation required?

Learning Outcomes Recognize the impact of different isolation levels and underlying concurrency control mechanisms in performance and scalability. Evaluate the impact of recovery mechanisms in performance.

[3] To help in this task, you can open two `psql` sessions side by side with different isolations and test queries. Check the appendix for more information.

[4] `https://www.postgresql.org/docs/17/runtime-config-wal.html`

[5] `https://www.postgresql.org/docs/17/wal-configuration.html`

Accessing Postgres' error log

- Using Docker:

```
docker logs postgres --tail 100 -f
```

- Using a native installation (the log location depends on the OS; on Linux, it is often found at `/var/log/postgresql/postgresql-17-main.log`):

```
tail -n 100 -f /var/log/postgresql/postgresql-17-main.log
```

Transactions in PSQL

- Setting the isolation level at the start of each transaction:

```
-- start
testdb=# begin;
BEGIN

-- set the target isolation, e.g.
testdb=*# set transaction isolation level repeatable read;
SET

-- execute queries
...

-- commit
testdb=*# commit;
COMMIT
```

- Alternatively, we can set the isolation for every transaction executed in that session.

```
-- e.g.
testdb=# set default_transaction_isolation to "repeatable read";

-- execute the transaction using default_transaction_isolation
testdb=# begin;
...
```