Concurrency

Dr Peadar Grant

November 18, 2024

Contents

1 Transactions S.3

2 ACID properties S.9

3 Transaction isolation S.15

4 Isolation levels S.17

Required reading

- Concurrency control in PostgreSQL manual.
- [Connolly and Begg, 2015, Chapter 22] discusses concurrency in general but focuses first on locking rather than multi-version concurrency control (as PostgreSQL supports).
- [Haerder and Reuter, 1983] introduce the ACID principles.
- You must have access to the shared database server.

1 Transactions

We will assume a transaction to be a “bounded unit of work” [Fowler, 2003].

(We have already met transaction handling as a basic “undo” facility on a practical level.)

1.1 Transaction control statements

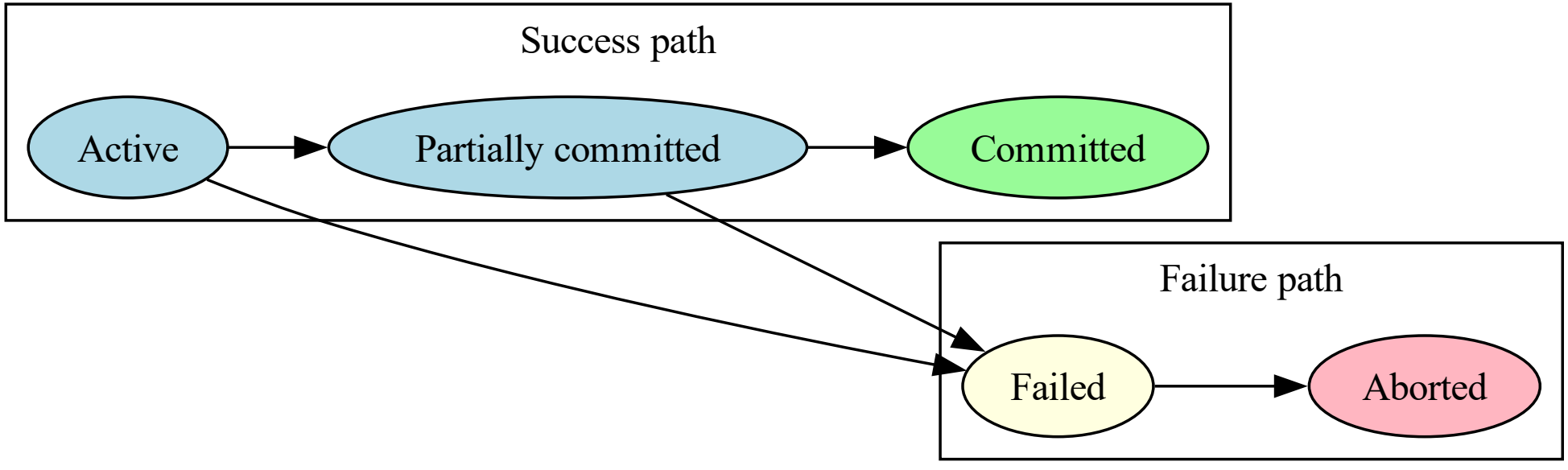
A transaction is identifed to the database by the following SQL.

```
BEGIN; --- starts the transaction

--- statements go here
INSERT ... ;
UPDATE ... ;
DELETE ... ; --- etc

COMMIT; --- finishes the transaction, saving changes
-- or
ROLLBACK; --- finishes the transaction, discards changes
```

1.2 Transaction states

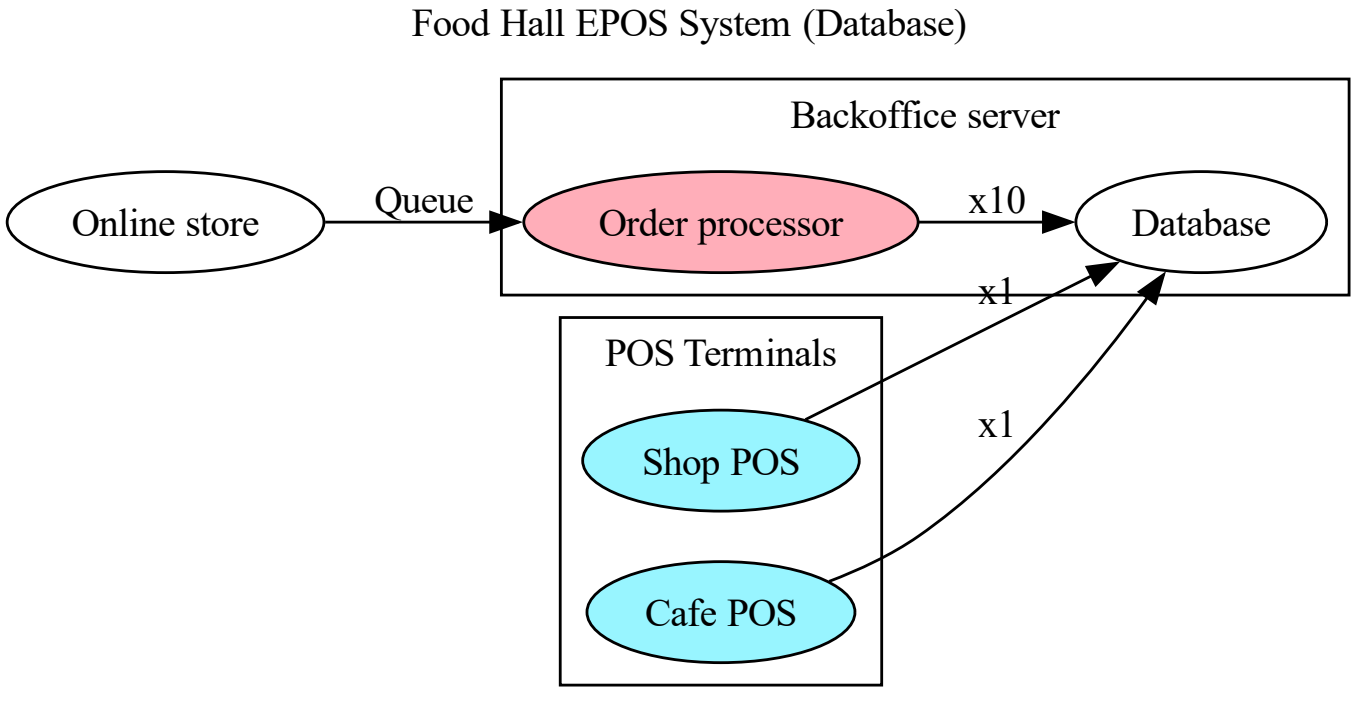


1.3 Savepoints

We have already met the `SAVEPOINT` keyword to do partial rollbacks.

For today's theoretical purposes we will ignore this usage!

1.4 Concurrent access



1.4.1 Key points on clients

- Clients may be homogeneous (all the same) or heterogenous (different).
- Some clients may be co-located on one host.
- Client connection durations may vary:
  - Some clients may hold a connection open when idle (desktop or server-side).
  - Others will open a connection, perform operations and quickly close it.
- Some clients may maintain a connection pool, where multiple connections are held open and used by individual threads.

2 ACID properties

Haerder and Reuter [1983] introduce the so-called ACID properties: **atomicity**, **consistency**, **isolation** and **durability**. We will first look at those relevant to a single connection: atomicity, consistency and durability. Isolation will be treated separately later. Key points to note:

- Individual ACID characteristics are not orthogonal to each other, and depend on each other in a number of ways.
- Although we can have a transaction with one or more operations, we assume for the purposes of this section that a transaction has at least two operations.
- We say define a transaction as committed once the `COMMIT` SQL statement returns, or the commit method/function of the database connectivity library returns when connecting from application layer code.

2.1 Atomicity

Haerder and Reuter [1983] define atomicity of a transaction as:

> It must be of the all-or-nothing type ..., and the user must, whatever happens, know which state he or she is in.

(The word atomic also appears when referring to 1NF. This is a separate usage of the term.)

Thus:

- An atomic database transaction is an indivisible or irreducible set of database operations.
- Either the transaction succeeds as a single unit or it does not.
- Atomicity guarantees that a transaction will not be partially committed.

2.1.1 Sample scenario

Consider a basic transaction in a casino backoffice:

```
-- transfer 20 chips from player 2389 to player 2991
BEGIN;
UPDATE player SET chips = chips - 20 WHERE id=2389;
UPDATE player SET chips = chips + 20 WHERE id=2991;
COMMIT;
```

If the first statement succeeds, issues could subsequently occur:

- The second statement may result in errors.
- Syntactically correct but commercially erroneous instructions may be issued.
- The client-server connection itself may be lost during the transaction.

Regardless of the reasons why, the guarantee of atomicity ensures that the transaction is not partially committed unless the COMMIT statement is given.

2.2 Consistency

Consistency is defined by Haerder and Reuter [1983] to be:

> A transaction reaching its normal end (EOT, end of transaction), thereby committing its results, preserves the consistency of the database. In other words, each successful transaction by definition commits only legal results.

In practical terms, this is usually taken to mean that all database constraints:

- UNIQUE constraints
- Foreign Keys
- Checks
- Triggers (later)

are upheld after the `COMMIT` statement returns.

2.3 Isolation

Isolation requires that [Haerder and Reuter, 1983]:

> Events within a transaction must be hidden from other transactions running concurrently.

This means that no other concurrent database connection should see anything about the current transaction until the `COMMIT` statement returns.

In practical terms, this is the weakest ACID property.

It is often partially relaxed in the interests of performance, section 3.

2.4 Durability

A guarantee of durability requires that [Haerder and Reuter, 1983]:

> Once a transaction has been completed and has committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions.

Generally we assume that the transaction has been stored on disk.

This characteristic also tells us that once committed, a transaction cannot be rolled back.

3 Transaction isolation

In practical terms, isolation is often not perfectly implemented.

The trade-off is a much increased database throughput.

The isolation level determines how strictly that transactions are isolated from each other.

To understand the different isolation levels, we must first see what can happen when isolation isn’t perfect.

3.1 Read-phenomena

A read-phenomenon is said to occur when the data visible to one transaction that has been started does not stay exactly the same until the transaction is finished (committed or rolled back).

This obviously doesn't include changes made by the transaction itself.

Phantom read In the course of a transaction, **two identical queries** are executed, and the **collection of rows** returned by the second query is **different** from the first.

Non-repeatable read During the course of a transaction, a **piece of data is retrieved twice** and the **values differ** between reads.

Dirty read: A transaction is allowed to **read data** that has been **written by another running transaction** and **not yet committed**.

Serialisation anomaly: The result of successfully committing a group of transactions is inconsiste with all possible orderings of running those transactions one at a time.

4 Isolation levels

ANSI SQL defines four standard isolation levels. Each trades increased concurrency against possible read phenomena Table 1.

Isolation level	Dirty read	Non-repeatable read	Phantom read	Serialization anomaly
Read uncommitted	Y	Y	Y	Y
Read committed	N	Y	Y	Y
Repeatable read	N	N	Y	Y
Serializable	N	N	N	N

Table 1: Read phenomena possible in each isolation level

4.1 Minimum requirement

Databases implementing the four isolation levels must guarantee at least the level of isolation specified in the level, that is, none of the prohibited read phenomena must occur.

They can however provide stricter isolation than standards require at any particular level.

Can be difficult to demonstrate certain read phenomena!

See the postgresql manual:
https://www.postgresql.org/docs/current/sql-set-transaction.html

4.2 Setting isolation levels

Isolation levels are set:

```
-- for transaction
SET TRANSACTION ISOLATION LEVEL ... ;
-- for transactions in this session
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL ... ;
```

The explanations below assume PostgreSQL is the database of interest.

4.3 Isolation levels in detail

4.3.1 READ UNCOMMITTED

ANSI standard allows transaction to experience **dirty reads** of as-yet uncommitted changes made by other transactions.

In practice, PostgreSQL treats this the same as `READ COMMITTED`, so dirty reads will never be seen in practice with PostgreSQL.

4.3.2 READ COMMITTED

Read Committed guarantees that:

> A statement can only see rows committed before it began. This is the default.

Read Committed in essence ensures that the state of the database is consistent with respect to a single command. In practice:

- Protects against dirty reads.
- **Phantom reads**, **Non-repeatable reads** and **Serialisation anomalies** possible.
- Any query is working with the committed state of the database plus any changes made thus far in the current transaction.
- Some operations may cause the one transaction to have to wait for the other to end (commit or rollback) before they complete.

4.3.3 REPEATABLE READ

Repeatable Read guarantees that:

> All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

Compared to read committed, it ensures that all statements within a transaction are working with the same consistent view of the database. In practice:

- Protects against dirty reads and non-repeatable reads.
- **Phantom reads** possible within ANSI standard, but will not occur in PostgreSQL.
- There may be serialization failures where an update operation cannot proceed because another transaction has committed a change to data that the operation depends on.

4.3.4 SERIALIZABLE

The PostgreSQL manual defines SERIALIZABLE as:

> The Serializable isolation level provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions; as if transactions had been executed one after another, serially, rather than concurrently.

References

Thomas M Connolly and Carolyn E Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. Pearson, 6th edition, 2015.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4), December 1983.