Programming Distributed Systems

11 Consistent transactions

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019

Motivation

Under concurrent data access, race conditions between clients yield undesirable situations:

- Clients may try to modify data at the same time with the danger of overwriting each others changes
- Invariants on the data can be observed to be violated between updates due to interleaving of operations by clients

$\Rightarrow$ Application programmers benefit from stronger guarantees, in particular when modifying multiple objects

What is a transaction?

What is a transaction?

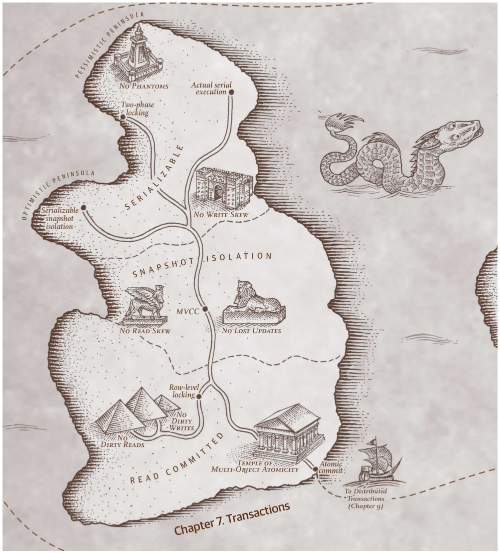
- A transaction groups several reads and writes on different objects together.

- Database literature: ACID
  - Atomicity: Cannot be broken into smaller parts; i.e. appears to execute as one operation
    - Either transaction succeeds to execute (*commit*)
    - Or transaction fails (*abort, rollback*)
  - Consistency: Keeps data invariants
    - Problematic and highly overloaded notion
    - Actually not really guaranteed by the database, but by the app!
  - Isolation: Concurrency semantics
    - Classically: Serializability (more on that later)
  - Durability: Persistence of commits

Why do we need transactions?

We want to simplify the following type of problems:

- To maintain foreign key references in a relational data model
    - E.g. it should not be possible to remove an entry while another update sets a reference to it
- To build safe secondary indexes
    - Index needs to be updated when values change
    - Clients should not observe deviations between data and index



Source: Martin Kleppmann, Designing Data-Intensive Applications, OReilly, 2017[1]

Read committed

Guarantees:

- When reading from the data store, only data that has been committed is visible (no *dirty reads*).
- When writing to the data store, only data that has been committed will be overwritten (no *dirty writes*).

Why are these guarantees useful?

Read committed

Guarantees:

- When reading from the data store, only data that has been committed is visible (no *dirty reads*).
- When writing to the data store, only data that has been committed will be overwritten (no *dirty writes*).

Why are these guarantees useful?

- Dirty reads might violate the atomicity property
  - Only some of the updates from another transaction might be visible
  - When a transaction aborts, its writes will be rolled back $\rightarrow$ What about the transactions that observed its tentative writes?
- Dirty writes are concurrency problems
  - When transactions update multiple values, their updates might overwrite each other in different order on different objects.

Read skew

Alice owns 100 EUR; 50 EUR are on account A, another 50 EUR on account B.

Now, Alice transfers 20 EUR from account A to account B. When checking the account states, account A shows 30 EUR, account B shows 50 EUR.

What happened to her money!?

Lost Updates

- Typical pattern: Read a value, modify it, update it (*read-modify-write* sequence)
- Problem when concurrently updating is that the second write does not observe (read) the changes from the first.
- Consequence: One update overwrites the changes from the concurrent one

How can we prevent this?

Lost Updates

- Typical pattern: Read a value, modify it, update it (*read-modify-write* sequence)
- Problem when concurrently updating is that the second write does not observe (read) the changes from the first.
- Consequence: One update overwrites the changes from the concurrent one

How can we prevent this?

- Conflict-resolution strategies like with CRDTs
- Guarantee atomicity of operations in read-modify-write sequence (e.g. using a lock, operating them in the same process, or using primitives such as compare-and-set; problem: replication)
- Abort if updated value is changed concurrently

Snapshot isolation

Guarantees:

- Transactions read from a consistent snapshot of the data store
- Prevents read skews and lost updates

- Important for long-running read-only operations such as backups, when doing integrity checks on data, or when executing queries
- Supported by a number of popular databases (such as PostgreSQL, MySQL, Oracle, etc.)
- Usually implemented using *multi-version* concurrency control
  - Idea: Readers don't block writers, and writers don't block readers
  - Every write generates a new version
  - Every read obtains the version that corresponds to the respective snapshot

Write Skew

Alica and Bob work in the examination office. To guarantee that student requests can be answered on any day, Alice and Bob cannot go on vacation on the same day.

Alice plans to take a holiday on July 1. She checks Bob's calendar - no entry!

Similarly, Bob plans to take a holiday on the same day. He checks Alice's calendar - no entry!

She adds to her calendar that she will be away on that day. And he adds to his calendar that he will be away on that day.

Serializability

- Guarantees that the result of executing transactions (potentially in parallel) is equivalent to an execution without concurrency
- Prevents write skew (and all the other anomalies mentioned so far)

- Implementation ideas
  - Execute in serial order (feasible only on single node)
  - Two-phase locking (2PL): pessimistic
  - Serializable snapshot isolation (SSI): optimistic
  - Consensus for distributed setting

Serializability

- Guarantees that the result of executing transactions (potentially in parallel) is equivalent to an execution without concurrency
- Prevents write skew (and all the other anomalies mentioned so far)

- Implementation ideas
  - Execute in serial order (feasible only on single node)
  - Two-phase locking (2PL): pessimistic
  - Serializable snapshot isolation (SSI): optimistic
  - Consensus for distributed setting

What is the difference between serializability and linearizability?

Conclusion

- Transactions provide means to operate safely on multiple objects
- Though many programmers are not aware of it, databases provide different isolation levels ($\rightarrow$ Check the default configuration!)
  - Subtle differences
  - No standardized naming
- Trade-off between performance and provided guarantees
- No tool support that tells you which one is the best *for your application*[2]

Further reading

Further reading I

[1] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2016. ISBN: 978-1-4493-7332-0. URL: http://shop.oreilly.com/product/0636920032175.do.

[2] Marc Shapiro und Pierre Sutra. "Database Consistency Models". In: *CoRR* abs/1804.00914 (2018). arXiv: 1804.00914. URL: http://arxiv.org/abs/1804.00914.