Designing Data-Intensive Applications

Chapter 7: Transactions

Introduction

- This chapter deals with how there are always things that can go wrong
- *Transactions* are a way of simplifying the programming model for applications by providing certain guarantees, so then the application doesn't need to worry about particular failure modes or concurrency issues
- A transaction is a series of operations grouped together as a unit, for which the system provides the certain set of guarantees mentioned above
- The chapter talks about the kinds of guarantees that systems offer, and then enumerates several *race conditions* which can be addressed by these different levels
- Although the concept of transactions applies to distributed databases, the examples in this chapter are all framed from a single-node perspective for the sake of introducing the topic with simpler examples

The Meaning of ACID

- The safety guarantees provided by transactions are often characterized by the well-known acronym ACID, which stands for Atomicity, Consistency, Isolation, and Durability
- Implementations differ, and claims of ACID compliant mean different things, especially for the isolation part
- Atomicity - all writes will either succeed or all will fail. Upon error, any partial writes need to be undone.
- Consistency - in this context, used to mean DB will always be in a consistent state.
    - Users probably interpret this as saying invariant properties will always remain true
    - But preserving invariants is really about how the application blocks operations in transactions
    - So consistency is really an application property, and shouldn't be part of this set of database guarantees
- Isolation - definition is that concurrent transactions won't interfere with each other at all, but that's rarely fully true
    - Not interfering at all would be *serializability*, or acting as if the transactions had run one at a time (in some order), even though they may have actually had some or all parts running concurrently.
    - This is costly, so most systems don't provide full serializability
    - Most of the rest of this chapter is about isolation
- Durability - once a transaction is reported success, any data written won't be lost
    - In practice, this is pretty intuitive, e.g. the data has been written to disk
    - Technically, (multiple) disks can fail, and other storage anomalies mean that data written to disk isn't fully 100.0% safe

Single-Object and Multi-Object Operations

- In ACID, if a transaction performs multiple writes, there are two guarantees provided:
    - If an error occurs partway through, the transaction should be aborted, and any writes made so far need to be rolled back/undone. Writes are all-or-nothing.
    - Concurrent transactions shouldn't interfere with each other. Other transactions will either see all or none of the writes, but not some proper subset.
- Single object writes can need guarantees, e.g writing a large JSON document
- When you have multiple objects in a transaction, there is this added complexity that the system needs to know which operations belong to the same transaction
- The need for multi-object transactions arises even if users don't think they are updating multiple objects
    - Updating denormalized data requires updating multiple places at once
    - Secondary indexes need to be updated when items are inserted/updated/deleted
- The guarantees of transactions means they can safely be retried after an error, but most systems won't do so automatically
    - Many object-relational mapping (ORM) layers don't do this
    - Although safe, sometimes retrying doesn't make sense, e.g. constraint violation

Weak Isolation Levels

- Isolation levels weaker than serializable have been implemented to provide better scalability
- Because most systems do not provide serializability, the author explains, "Rather than blindly relying on tools, we need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them."
- We will list weak isolation levels in increasing order of strength, and list the six race conditions they prevent (in corresponding order of difficulty)
- Note: when a transaction completes successfully, it is said to be *committed*. The term *uncommitted write* refers to a write that's part of a transaction that has not completed yet. Uncommitted writes may eventually be committed, or they may be rolled back.
- Note: this discussion talks about different kinds of locks. If you are not familiar with locks, wikipedia has a short explanation: https://en.wikipedia.org/wiki/Record_locking. Here is deeper coverage: https://www.methodsandtools.com/archive/archive.php?id=83.

Read Uncommitted

- Only mentioned in a footnote is the weakest isolation level where the system only prevents one race condition: dirty writes
- Dirty writes -- One transaction overwrites an uncommitted write on the same object from another transaction
    - The book gives an example of a dirty write situation where selling a car requires one write as to who bought the car, and a second write to create the invoice. The dirty write allows the car to be sold to one person, but the invoice to be sent to the other person.
- Implementation:

    - Usually dirty writes are prevented using row level locks. Exclusive locks are obtained prior to writes. Other transactions that try to write the same object are blocked.
- Read uncommitted is used in analytic workloads where the expectation is that writes by other processes aren't happening
- Read uncommitted provides the fast performance and greatest concurrency for read workloads, because reads don't have any overhead

Read Committed

- In addition to dirty writes, read committed prevents dirty reads
- Dirty reads -- One transaction reads another transaction's writes before they have been committed
    - Dirty reads allow other transactions to see a partial set of writes from this transaction
    - If a dirty read happens and then this transaction is rolled back, the dirty read allows other transactions to see a value that was never actually written
- Implementation:
    - Dirty reads can be prevented by obtaining shared read locks prior to read (assuming writes are using exclusive locks mentioned above), and releasing them immediately after the read. This can hurt performance if slow writes block other transactions.
    - Alternative is to record the value of each object before it is written, and return that old value to all other transactions until the writing transaction completes.
- Read committed is very common because it is easy to implement, provides good concurrency, and is way better than having no transactions.
- But many people do not realize that read committed still allows numerous race conditions

Snapshot Isolation and Repeatable Read

- Not all snapshot isolation implementations are the same. They all prevent read skew in addition to the above. Some also prevent lost updates.
- Snapshot isolation allows a transaction to read from a consistent snapshot of all of the data from a single point in time.
    - Note that some products call snapshot isolation "repeatable read." To add to the confusion, other products mean other things by "repeatable read."
- Snapshot isolation is important for database backups, consistency checks, and long running analytic queries
- Read skew (nonrepeatable reads) -- A transaction sees different parts of the database at different points in time (due to write activity from other users).
    - The book gives an example where a user reads the bank balance from two accounts, where separately a transfer between accounts is happening. It's okay if the user sees both before values or both after values, but read skew is when one account sees the before value and the other sees the after value.
- Implementation:

    - Write locks are still used to prevent dirty writes
    - Snapshot isolation behavior is usually implemented with multi-version concurrency control (MVCC).
        - Each transaction gets a unique, always increasing transaction ID number
        - Every write is tracked with the transaction ID
        - When a transaction runs, it ignores values from transactions that were already in progress or started after it
- Lost updates -- Two transactions concurrently perform a read-modify-write cycle on the same object. One transaction overwrites the other's write without incorporating the other's changes, so data is lost.
- Lost updates situations can be reduced with:
    - Atomic write operations
        - E.g.: UPDATE counters SET value = value + 1 WHERE key = 'foo';
    - Manually setting update locks
        - E.g.: SELECT value FROM counters WHERE key = 'foo' FOR UPDATE;
    - Atomic compare-and-set operations
    - Locks and compare-and-set operations get more complicated with replication because there are multiple copies of the data
- Some implementations of snapshot isolation detect lost updates automatically

Write Skew and Phantoms

- The final two race conditions can only be prevented by full serializability
- Write skew -- A transaction reads something, makes a decision based what it read, and writes a value dependent upon the decision. While the transaction was running, other writes to different objects have made the decision no longer correct.
- Phantom reads -- A transaction reads objects that match some criteria. Another transaction makes a write that creates a new record that affects the search criteria. Snapshot isolation prevents straightforward phantom reads, but phantoms in the context of write skew require special treatment, such as index-range locks.
    - Materializing conflicts is a method that creates a set of locks just for the purpose of isolation, without any real data being stored. Can be error-prone.

Serializability

- Serializable isolation is the only way to guarantee that if a transaction runs correctly when run by itself, it will still run correctly when run concurrently
- The reasons why it's not used are cost and performance

Actual Serial Execution

- Run all transactions on a single thread
- If you have an in-memory database, transactions may be so fast that you can keep up despite having only one thread on one core
- Requiring transactions to be a single stored procedure call can avoid any user/application delays between statements inside a transaction

- You can run on multiple cores if you have each thread operate on a separate partition of the data, but cross-partition transactions must be very limited
- Anti-caching, where you abort the transaction and retry it after all of the data has been loaded into memory, is an interesting strategy that support serial execution

Two-Phase Locking (2PL)

- Typically implemented with locks. Reads require shared locks and writes require exclusive locks. Reads block writes, and writes block reads.
  - Locks, once obtained, must be held until the end of the transaction
  - Predicate locks theoretically could restrict the relevant range of data, but slow to execute
  - Index-range locks are typically used instead
    - If hard to implement exact criteria, a superset can be locked
    - Worst case, an entire table can be locked
- The overhead of locking is one negative, but the real problem is that it is so easy for transactions to get blocked by other transactions, so concurrency can be poor
- It is possible for two (or more) transactions to be caught in a cycle where one is waiting for the other to release a lock, and the other is waiting for the first one to release a lock. This situation will not ever resolve on its own, and is called a *deadlock*.
  - Deadlocks can be detected automatically, and one transaction is killed
  - Note that intent locks can help avoid deadlocks

Serializable Snapshot Isolation (SSI)

- A relatively new algorithm is an *optimistic* concurrency control technique
- Allows concurrency and upon commit detects if any isolation was violated
- Worst case under high contention is lots of transactions getting aborted
- Builds on snapshot isolation MVCC
  - Detects stale MVCC reads which commit prior to this transaction committing
  - Detects writes that affect prior reads