DDIA Notes Chapter – Transactions

A transaction is where several reads and writes are grouped together into a single logical unit so that they can be executed as one operation. The entire transaction will either succeed (commit) or fail (abort, rollback). Transactions allow for applications to not have to worry about partial failures and safely retry operations as databases provide these safety guarantees.

At times, it is better to weaken transaction guarantees to increase performance or availability. Almost all relational databases and some nonrelational databases support transactions. In a relational database, everything between a `BEGIN TRANSACTION` and `COMMIT` statement is part of the same transaction. Many nonrelational databases do not provide a way to group operations together.

ACID

Transactions provide safety guarantees, also known as ACID: *Atomicity, Consistency, Isolation,* and *Durability*. Databases that are not ACID-compliant are called BASE: *Basically Available, Soft State,* and *Eventual Consistency*.

Atomicity

Atomicity is the ability to abort a transaction on error and undo any writes that have been made so far. ***Abortability*** would have been better to describe the *A* in ACID.

This is not to be confused with atomic operation in multi-threaded programming. An atomic operation is where if one thread executes an atomic operation, another thread could not see half-finished results from that operation.

Atomicity can be implemented using a log such as B-trees for crash recovery.

Consistency

Consistency is the application's perspective that the database is in a "good state." Certain statements about the data (invariants) must always be true. For example, in an accounting system, credits and debits across accounts must be balanced.

The *C* doesn't really belong in ACID. Consistency is a property of the application, whereas atomicity, isolation, and durability are properties of the database.

This is not to be confused with replica consistency and eventual consistency (replication), consistent hashing (partitioning), and consistency in the CAP theorem.

Isolation

Isolation is where each transaction running concurrently with other transactions cannot interfere with each other. When transactions have been committed, the result should be identical to if they were run one after another (serially) even if they were run concurrently.

Durability

Durability is the guarantee that once a transaction has been committed, the data will never be lost. It would safeguard against any hardware faults, database crashes, etc.

Isolation Levels

Read committed isolation and snapshot isolation are weak (nonserializable) isolation levels. Serializable isolation guarantees that transactions have the same effect as if they were run serially (one at a time, without any concurrency). In practice, serializable isolation incurs a significant performance cost. Many databases choose not to pay this cost. A popular comment is, "Use an ACID database if you're handling financial data!" Even many relational databases that are considered ACID use some form of weak isolation.

Read Committed Isolation

Read committed isolation is the most basic level of transaction isolation. It guarantees:

- When reading from a database, only data that has been committed will be returned (no dirty reads).
- When writing to a database, only data that has been committed will be overwritten (no dirty writes).

Dirty Reads

Dirty reads are where one transaction reads another transaction's writes before they have been committed. As the data is in a partially-updated state, it will cause other transactions to make incorrect decisions. Also, if a transaction aborts, other transactions may have read data that will now be rolled back.

Most databases prevent dirty reads by remembering both the old committed value and the new value set by the transaction that currently holds the write lock. While a transaction is ongoing, other transactions that read the object are given the old value. Once a new value is committed, transactions switch over to reading the new

value.

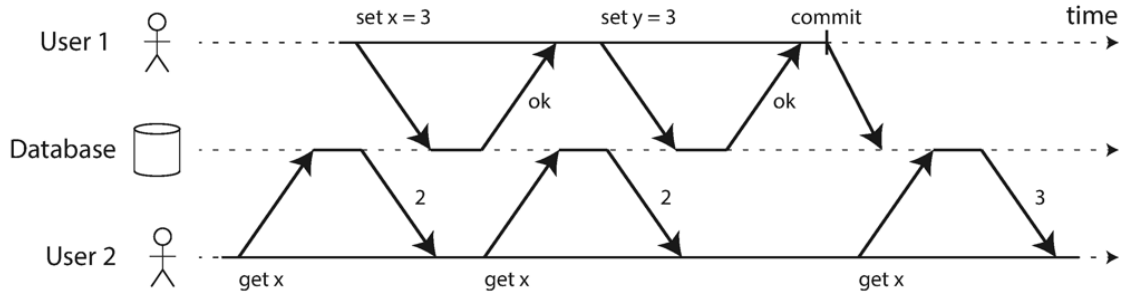

Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

Read locks are rarely used as it harms the response time for read-only transactions. One long write transaction can force many read-only transactions to wait until the write transaction is completed.

Dirty Writes

Dirty writes are where a transaction overwrites uncommitted data of an in-progress transaction. The database needs to delay the second transaction's write until the first transaction's write completes.

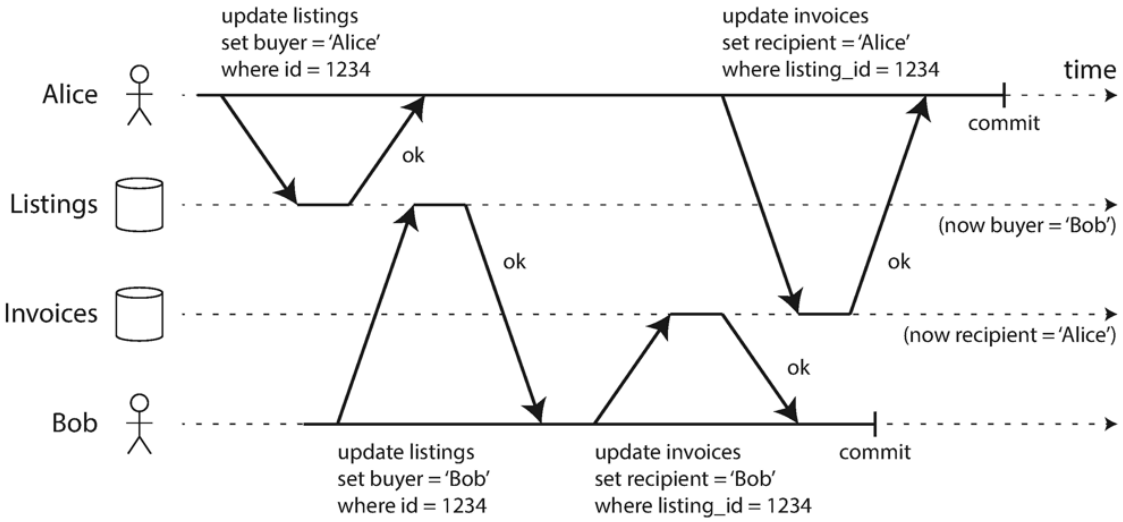

Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Most databases prevent dirty writes by using row-level locks. When a transaction wants to modify an object (row or document), it must first acquire a lock on that

object. The lock is held until the transaction is committed or aborted. Other transactions that want to modify the object must wait until the lock is released.

Read Skew (Nonrepeatable Reads)

Read skew happens when a transaction is writing to multiple objects and in parallel, a second transaction reads the new data in some objects and old data in other objects. This causes the second transaction to see inconsistencies in the database.

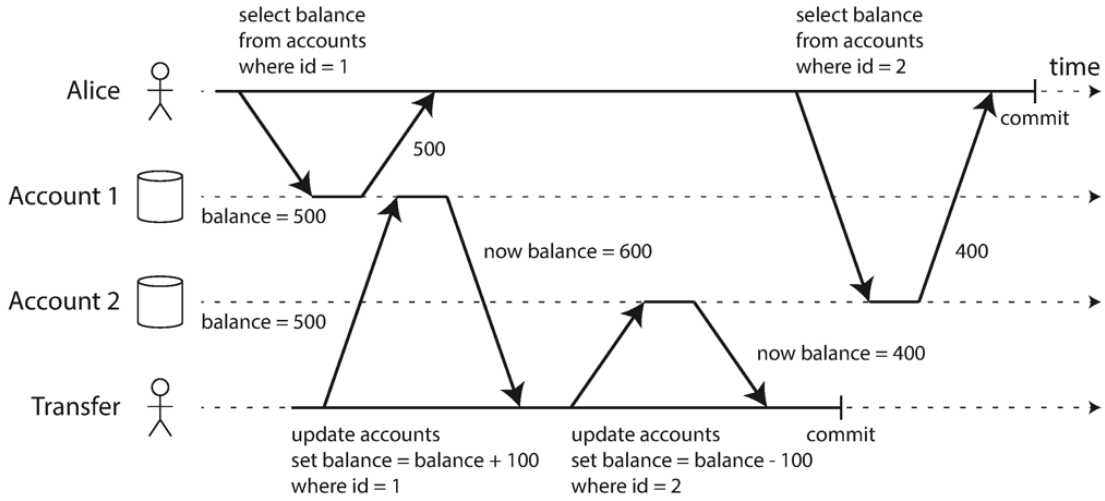

Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

Read committed isolation doesn't prevent read skew (nonrepeatable reads). This is solved by snapshot isolation.

Snapshot Isolation

Snapshot isolation is where a transaction will only read data from a historical database state of when the transaction began. Even if data is changed later by another transaction, each transaction will read old data from its specific snapshot.

Some other databases refer to snapshot isolation with a different name. In Oracle, this is implemented as serializable isolation. In PostgreSQL and MySQL, this is implemented as repeatable read isolation.

Multi-Version Concurrency Control (MVCC)

Snapshot isolation is usually implemented with multi-version concurrency control (MVCC). The database will keep several different committed versions of an object as various in-progress transactions need to read database state at different points in

time. Each transaction is always given a unique, always-incrementing transaction ID.

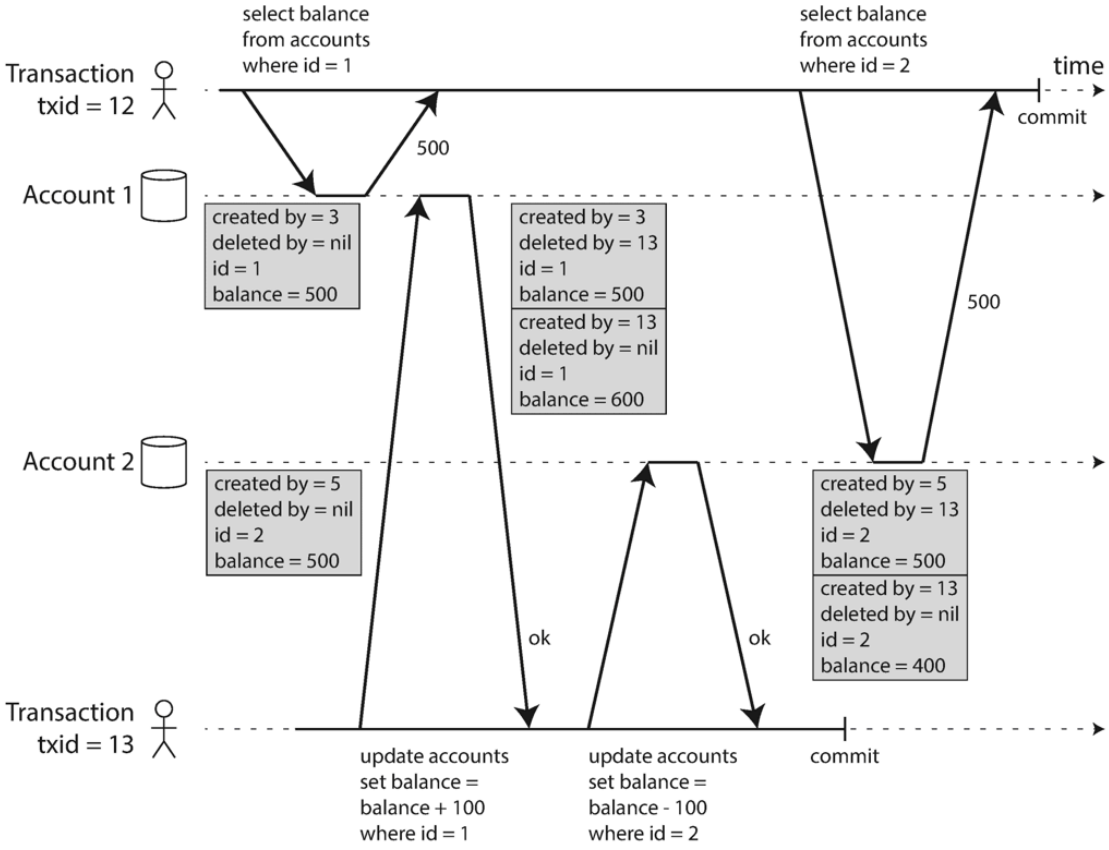

Figure 7-7. Implementing snapshot isolation using multi-version objects.

Modifications

Whenever a write occurs, the data is tagged with the writer's transaction ID.
Whenever a transaction reads from the database, any writes made by transactions with a later transaction ID are ignored.

A garbage collection process will periodically remove old object versions when they are no longer visible to any transactions.

Deletions

Whenever a transaction deletes data, the row isn't actually deleted. Instead, it is marked for deletion by setting the `deleted_by` field to the transaction ID.
A garbage collection process will periodically delete objects marked for deletion when they are no longer visible to any transactions.

Lost Updates

Lost updates are when two transactions concurrently read an object, modify it, and write back the modified object (**read-modify-write** cycle). One of the modifications is lost as the second write does not include the first modification. The later write clobbers the earlier write.

Atomic Update Operations

To prevent lost updates, many databases provide atomic update operations. Applications are expected to use atomic updates instead of read-modify-write cycles. For example, the following is a standard update instruction in relational databases:

```
UPDATE counters SET value = value + 1 WHERE key = 'foo'
```

Atomic updated operations are implemented by acquiring an exclusive lock on the object when it is read until the update has been committed (cursor stability). This prevents other transactions from reading the same object. Relational databases, Redis, and MongoDB provide atomic updates.

Automatic Detection

Another solution is to have a transaction manager that detects lost updates. If one occurs, the transaction manager will abort the offending transaction and force it to retry its read-modify-write cycle. Snapshot isolation needs to be enabled for this check to be implemented efficiently. PostgreSQL, Oracle, and SQL Server snapshot isolation levels automatically detect when a lost update has occurred and abort the offending transaction. MySQL (InnoDB) does not automatically detect lost updates.

Conflict Resolution (Replication)

Replicated databases allow concurrent writes to create several conflicting versions of an object (siblings). These databases use application code or special data structures to resolve and merge these versions afterwards.

Write Skew

Write skew happens when a transaction reads an object, makes a decision based on the data, and writes its decision to the database. However, by the time the decision is committed, another transaction has changed the initially-read object causing the premise of the decision to be false. Snapshot isolation doesn't prevent write skew.

Only serializable isolation prevents write skew.

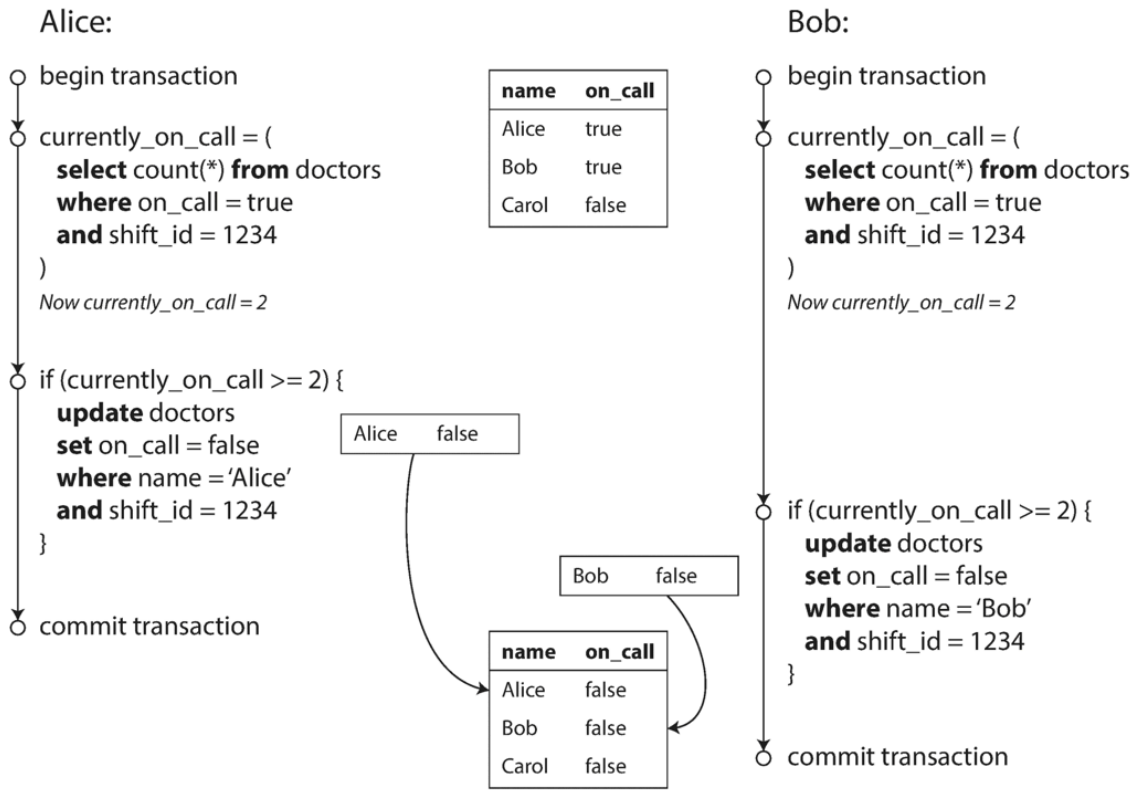

Figure 7-8. Example of write skew causing an application bug.

Phantoms

Phantoms happen when a transaction reads objects that match a search query, then another transaction commits a write that adds new objects that match the search query. In this case, locks aren't effective as the objects do not exist yet. Snapshot serialization prevents standard phantom reads but 2PL's index-ranged locks are required to prevent phantoms and write skew.

Serializable Isolation

Serializable isolation is the strongest isolation level. It guarantees that even if transactions are run in parallel (concurrently), the end result is the same as if they were run one at a time (serially). This isolation level prevents all possible race conditions.

Actual Serial Execution

The simplest way to implement serializable isolation is to remove concurrency entirely. That is, to execute each transaction one at a time, in serial order, on a single

thread. Multi-threaded concurrency was considered essential for good performance but cheaper, large-sized RAM has made single-threaded execution possible. Today, it is feasible to keep an entire active dataset in memory. Transactions in memory execute much faster than from disk. Redis uses this approach. As throughput is limited to a single CPU core, transactions need to be structured differently to make the most of that single thread.

Partitioning data across multiple nodes can allow scaling to multiple CPU cores but require either:

1. Transactions only read and write data from a single partition so that cross-partition coordination will not be necessary.
2. The database system to coordinate transactions across all partitions that the transaction touches.

Key-value data can be partitioned easily. Data with multiple secondary indexes will generally require cross-partition coordination.

Two-Phase Locking (2PL)

In two-phase locking (2PL), several transactions are allowed to concurrently read from the same object as long as no transaction is writing to it. If a transaction wants to read from an object, it must wait for any transaction writing to the object to commit or abort. Also, vice versa is true: if a transaction wants to write to an object, it must wait for all transactions reading from the object to commit or abort. MySQL (InnoDB) and SQL Server use 2PL.

2PL is implemented using shared locks and exclusive locks. The locks are implemented as follows:

- When a transaction wants to read an object, it must acquire a shared lock. It must wait on any existing exclusive lock on the object. Other transactions reading from the same object can also acquire the shared lock simultaneously.
- When a transaction wants to write to an object, it must acquire an exclusive lock. It must wait on any existing lock (shared or exclusive) on the object.
- When a transaction first reads and then writes to an object, it can upgrade its shared lock to an exclusive lock. It must wait on any transactions that have the shared lock on the same object to complete.
- After a transaction acquires a lock, it must hold onto the lock until it commits or aborts.

Deadlocks easily occur when a transaction is stuck waiting for another transaction to release its lock, and vice versa. The database detects deadlocks between transactions

and aborts one so that the other can make progress. The application will retry the aborted transaction.

To protect against phantoms and write skew, databases with 2PL use index-ranged locks (next-key locks). This works similarly to shared or exclusive locks but is applied across a range of indexes versus a single object.

2PL has significantly worse performance with transaction throughput and response times versus weak isolation. This is due to the overhead of acquiring/releasing locks and reduced concurrency.