TDT4225
Chapter 7 – Transactions

Svein Erik Bratsberg
Department of Computer Science (IDI), NTNU

Transactions are necessary because

- The database software or hardware may fail at any time
- The application may crash at any time
- Interruptions in the network cause apps to be lost from db
- Several clients may have concurrent, conflicting writes
- A client may read data that doesn’t make sense due to partial updates
- Race conditions between clients can cause surprising bugs

Transactions

- Used to group several reads and writes together into a logical unit
- Commits or aborts as a unit
- May be retried in case of abort
- Used to simplify the programming model for applications accessing a database
- Some applications don’t need transactions
- Concepts to learn: Read committed, snapshot isolation, and serializability.

History of transactions

- Started with System R (IBM) in the 70s
- Supported almost without change since then in SQL databases (MySQL, PostgreSQL, Oracle, SQL Server, DB2)
- NoSQL skipped transactions
  - Partitioning / Sharding
  - Replication
  - CRUD
- NewSQL added transactions to NoSQL (Spanner, CockroachDB, YugabyteDB, TiDB)

ACID – properties of a transaction

Transaction: A sequence of operations on DB which are

- **A** – atomic: completely run or not
- **C** – consistency: (primary key, references, check, etc)
- **I** – isolation: does not notice other transactions.
- **D** – durability: nothing lost after commit.

A transaction is usually a logical operation or task.

Kleppmann: The high-level idea is sound, but the devil is in the details. Today, when a system claims to be “ACID compliant,” it’s unclear what guarantees you can actually expect. ACID has unfortunately become mostly a marketing term.

- BASE is «not ACID»

Atomicity

- Atomic refers to something that cannot be broken down into smaller parts.
- In ACID, atomicity is not about concurrency
- ACID atomicity describes what happens if a client wants to make several writes (and reads)
- Atomicity simplifies the problem of half-done operations: If a transaction was aborted, the application can be sure that it didn’t change anything, so it can safely be retried.
- *Kleppmann*: Abortability is a better word.

Consistency

- Consistency is an overloaded word.
- In ACID, consistency refers to an application-specific notion of the database being in a “good state.”
- Consistency depends on the application’s notion of invariants
- Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application. The application may rely on the database’s atomicity and isolation properties in order to achieve consistency, but it’s not up to the database alone. Thus, the letter C doesn’t really belong in ACID. (Kleppmann)

Isolation

- Isolation in ACID means that concurrently executing transactions are isolated from each other.
- Serializability: Each transaction can pretend that it is the only transaction running on the entire database.
- The database ensures that when the transactions have committed, the result is the same as if they had run serially

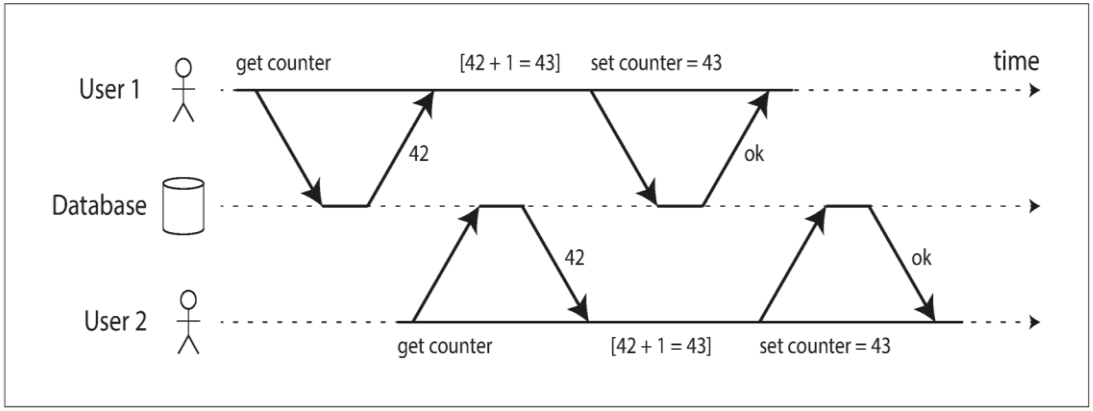

Figure 7-1. A race condition between two clients concurrently incrementing a counter.

Durability

- Durability is the promise that once a transaction has committed successfully, any data it has written will not be forgotten.
- Usually done by WAL – Write Ahead Logging (force-log-at-commit). Some oldtype databases forces data to disk.
- Force/Steal classification (Theo Harder)
- Force data or force redo log
- Steal slot in buffer (force undo log) or no-steal

Single-Object and Multi-Object Operations

- Multiple writes could be rolled back (atomicity)
- Concurrent transactions may see all writes or none (isolation)
- SELECT COUNT(*) FROM emails WHERE recipient_id = 2 AND unread_flag = true
- Slow, so use a cached counter: Anomaly may appear

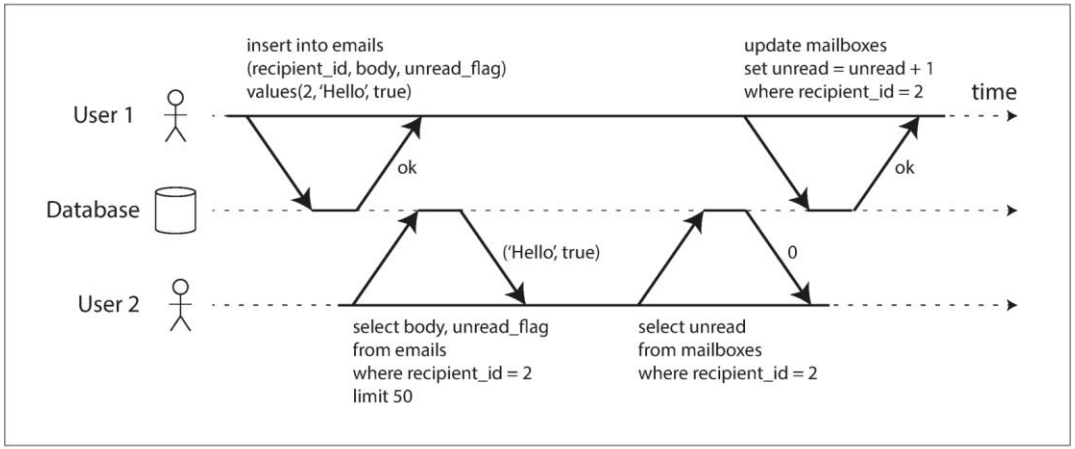

Figure 7-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a "dirty read").

Transactions and Atomicity

- If an error occurs somewhere over the course of the transaction: Abort.
- NoSQL databases usually don’t support this

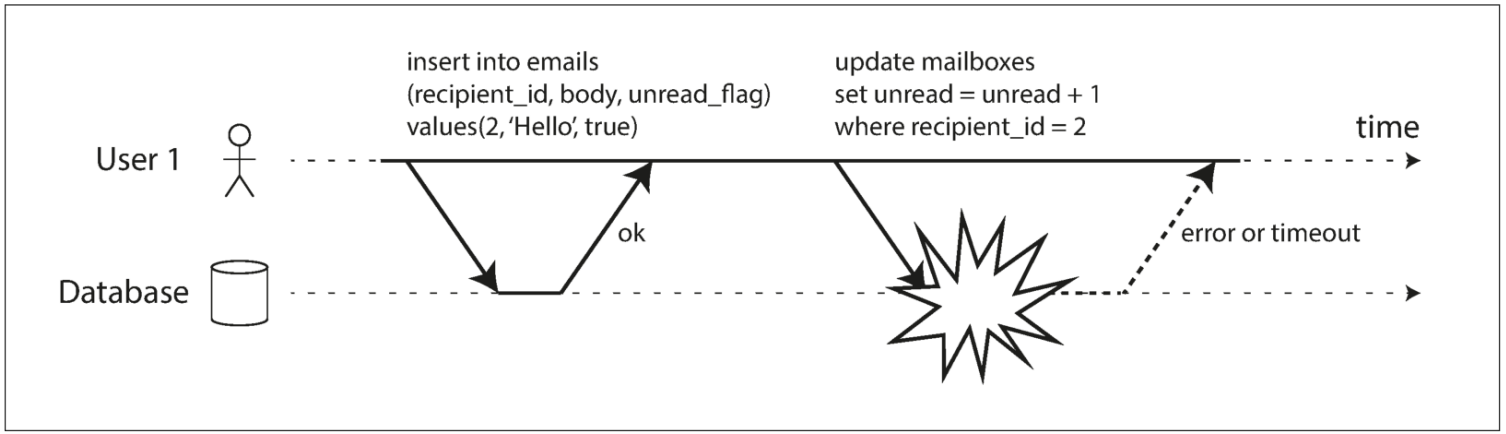

Figure 7-3. Atomicity ensures that if an error occurs any prior writes from that transaction are undone, to avoid an inconsistent state.

Single Object Writes

- All databases try to make single object writes atomic.
- E.g. writing large JSON strings may fail halfway
- Use Logging & concurrency control
- Support increment instead of using read-modify-write cycle (some DBs do)
- Compare-and-set and other single-object operations have been dubbed “lightweight transactions” or even “ACID” for marketing purposes
- A transaction is usually understood as a mechanism for grouping multiple operations on multiple objects into one unit of execution.

The need for multi-object transactions

- Rows may contain foreign keys, thus need to update several rows in different tables
- Document databases using normalization may have the same problem, documents referencing other documents
- Secondary indexes require multiple writes belonging to the same logical operation
- Operations belonging to the same logical task (of course)

Handling errors and aborts

- *Leaderless replication*: “the database will do as much as it can, and if it runs into an error, it won’t undo something it has already done”—so it’s the application’s responsibility to recover from errors”
- Retries are not always good:
  - The transaction succeeded, but the network failed
  - Overload: retrying the transaction will make the problem worse
  - It is only worth retrying after transient errors (deadlocks, concurrency, etc)
  - If the transaction also has side effects outside of the database, those side effects may happen even if the transaction is aborted
  - If the client process fails while retrying, any data it was trying to write to the database is lost

Weak Isolation Levels (1)

- Concurrency issues (race conditions) when one transaction reads data that is concurrently modified by another transaction
- Concurrency bugs are hard to find by testing
- Thus, transaction isolation: Transactions should feel that they are the sole users running.
- We need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them.
- Isolation levels and their problems and properties follow …

Read Committed

1. When reading from the database, you will only see data that has been committed (no *dirty reads*).
2. When writing to the database, you will only overwrite data that has been committed (no *dirty writes*).

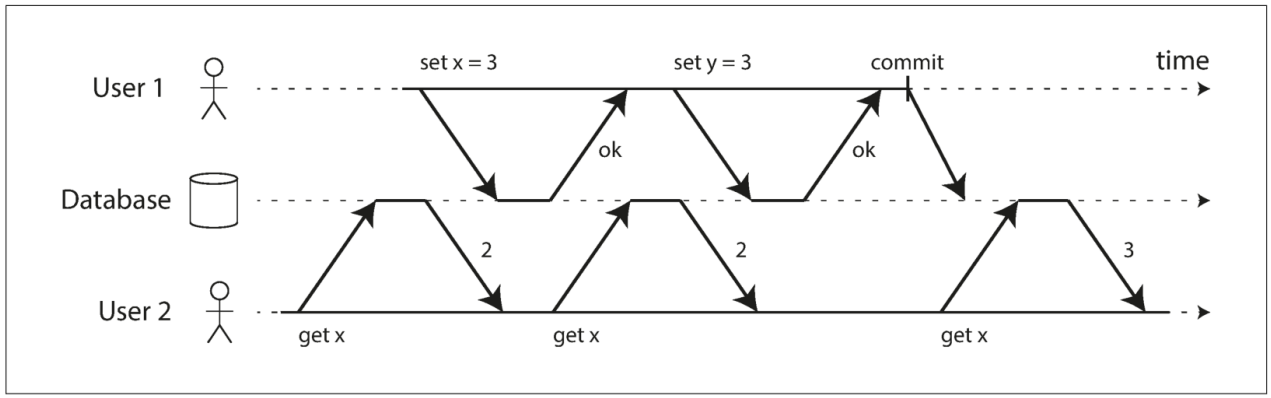

Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

Read Committed (2)

- If a transaction needs to update several objects, a dirty read means that another transaction may see some of the updates but not others
- If the database allows dirty reads, that means a transaction may see data that is later rolled back

No Dirty Writes

- The earlier write is part of a transaction that has not yet committed, so the later write overwrites an uncommitted value.

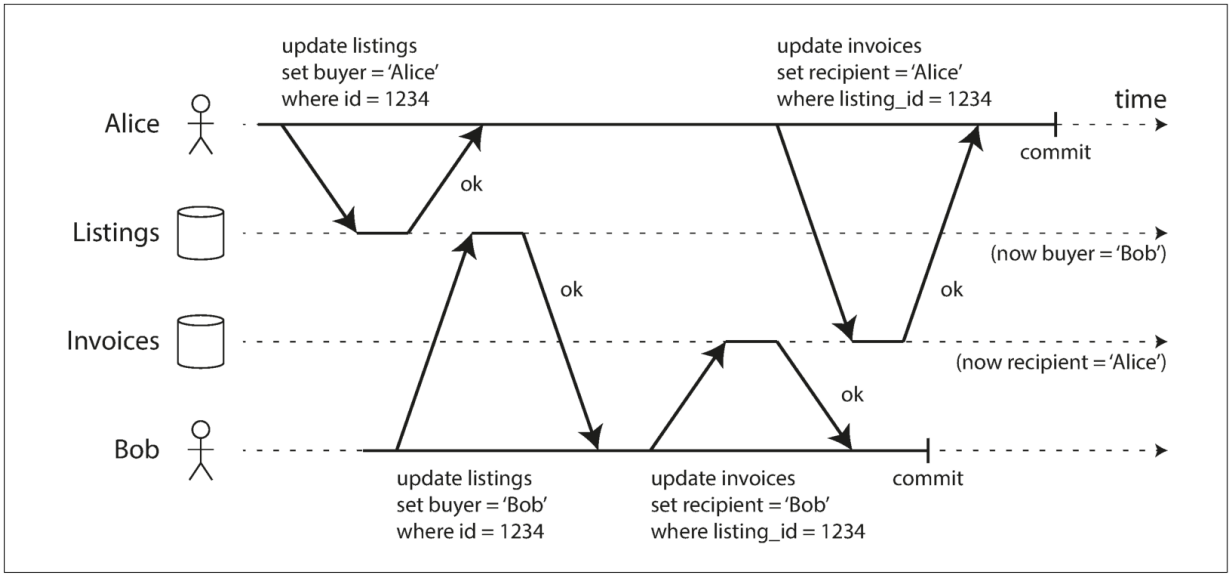

Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Read Committed

- **Read committed** is a very popular isolation level. It is the default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL, …
- Most databases prevent dirty reads by keeping old values for writes until the transactions commit. Read transactions may read the old value.
- Only when the new value is committed do transactions switch over to reading the new value.
- To keep single record locks would cost too much, since one writer may cause multiple readers to wait

Snapshot Isolation and Repeatable Read

- Problems with Read Committed: This anomaly is called a nonrepeatable read or read skew

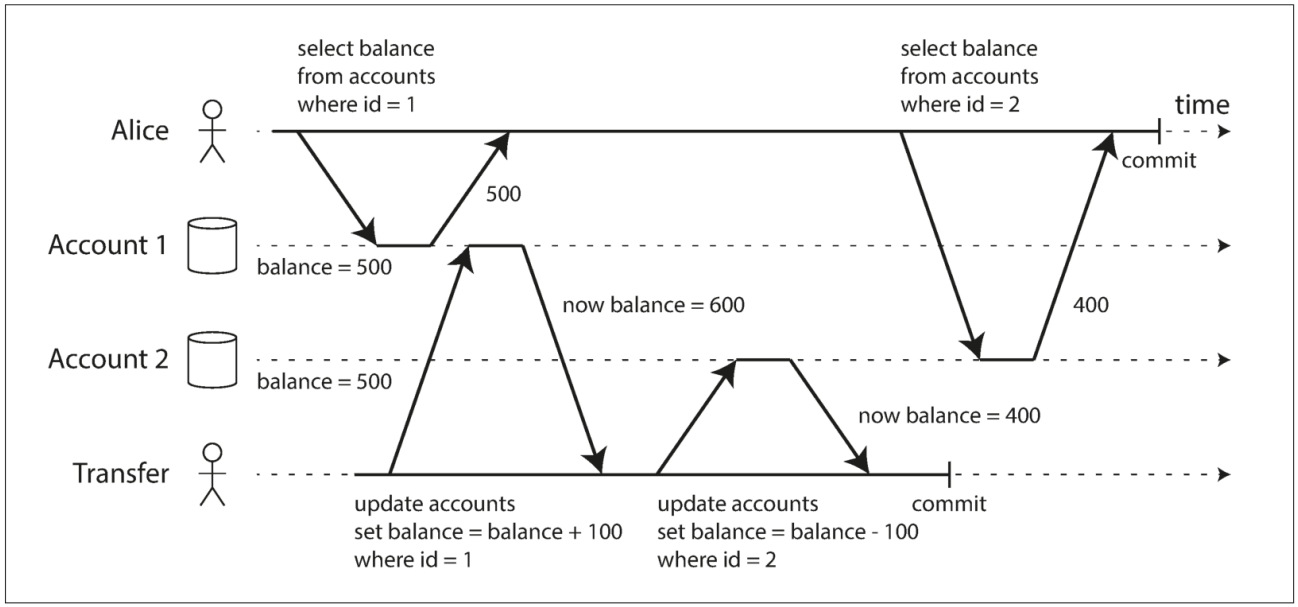

Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

Snapshot Isolation and Repeatable Read (2)

- Snapshot isolation is the most common solution to this problem. Supported by PostgreSQL, MySQL/InnoDB, Oracle, SQL Server, and others.
- Use write locks, but reads do not require locks
- Readers never block writers, writers never block readers
- Multiversion concurrency control (MVCC). Store several versions of an updated record. One for each snapshot.
- Storage engines that support snapshot isolation typically use MVCC for their read committed isolation level.

PostgresSQL's MVCC

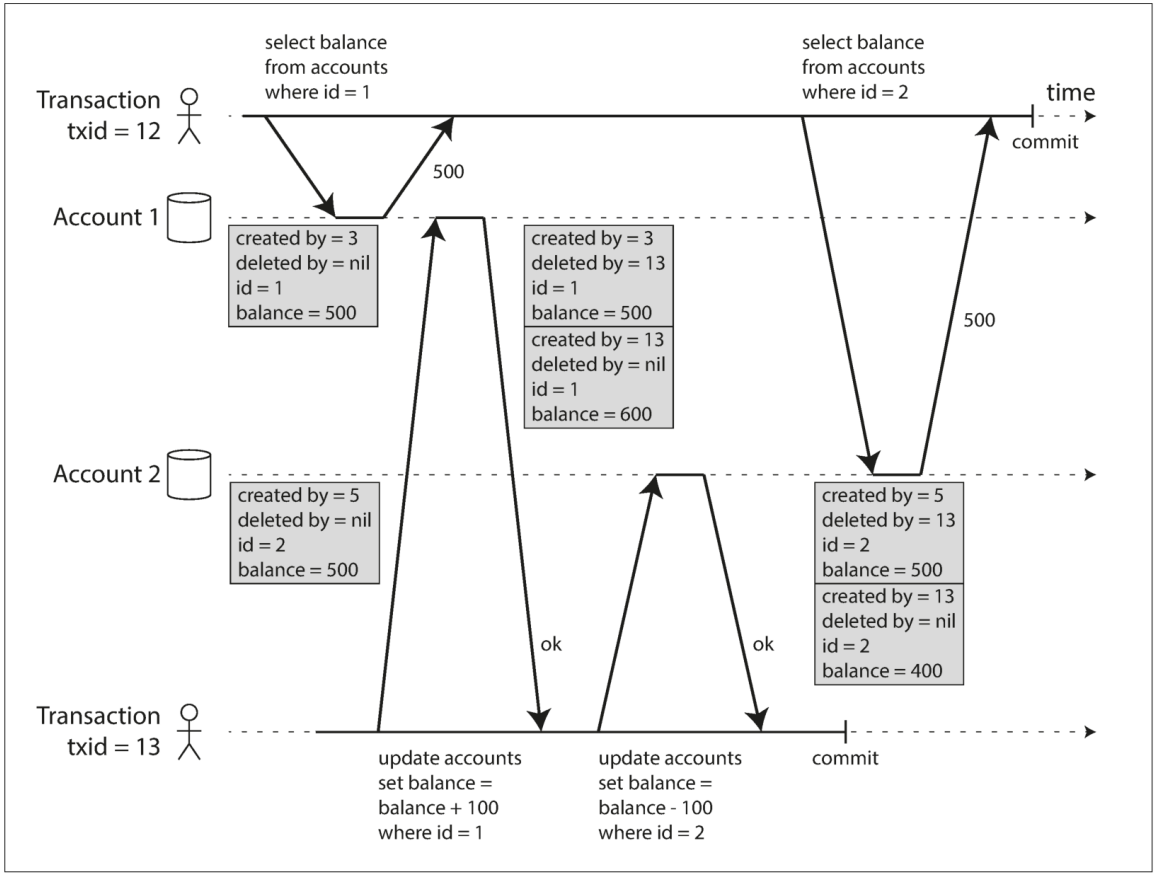

Figure 7-7. Implementing snapshot isolation using multi-version objects.

Visibility rules for observing a consistent snapshot

- TransactionIds are used to decide visibility of objects
  1. Existing, active transactions' writes are not visible to a new transaction
  2. Writes by aborted transactions are ignored
  3. Writes by newer transactions are not visible
  4. All other writes are visible
- Alternative rules (skipped)

MVCC, Indexes and COW-B+-trees

- Should the index point to all versions of an object?
- PostgresSQL has optimizations when the object versions are in the same page (avoids updates to index)
- CouchDB, Datomic, and LMDB: Copy-on-write B+-trees.
- All writes make copies of pages which are propagated up in the tree.
- And a new root is visible at commit
- Creates need for compaction and garbage collection
- BTRFS – File system om Linux based on copy-on-write B+-trees.

Repeatable Read and the SQL «standard»

- Real confusion on use of Repeatable Read in SQL databases
- Oracle: calls it Serializable
- PostgresSQL and MySQL calls snapshot isolation for Repeatable Read
- IBM DB2 uses «Repeatable Read» to refer to Serializability
- SQL standard based on System R’s use of isolation levels, and snapshot isolation wasn’t implemented at that time

Preventing Lost Updates

- Snapshot isolation: what a read-only transaction can see in the presence of concurrent writes
- Atomic write (update) operations:

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

- Implemented by exclusive lock
- Object-relational mapping frameworks make it easy to accidentally write code that performs unsafe read-modify-write cycles instead of using atomic operations provided by the database

Explicit locking

- SELECT FOR UPDATE provides explicit locking

Example 7-1. Explicitly locking rows to prevent lost updates

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE; ❶

-- Check whether move is valid, then update the position
-- of the piece that was returned by the previous SELECT.
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

- PostgreSQL's repeatable read, Oracle's serializable, and SQL Server's snapshot isolation levels automatically detect when a lost update has occurred and abort the offending transaction.

Compare-and-Set

- Used by some databases without transactions, often called *lightweight transactions (LWT)*
- The purpose of this operation is to avoid lost updates by allowing an update to happen only if the value has not changed since you last read it.

```
-- This may or may not be safe, depending on the database implementation
UPDATE wiki_pages SET content = 'new content'
  WHERE id = 1234 AND content = 'old content';
```

- If the content has changed and no longer matches 'old content' , this update will have no effect, so you need to check whether the update took effect and retry.

Conflict Resolution and Replication

- In multi-leader replication you may have conflicts when parallel updates happen
- Use application code to resolve conflicts
- You need to detect the conflicts and this must be provided by the database system using vector clocks or version vectors
- Last Write Wins (LWW) is the default solution used in many database systems

Write Skew and Phantoms

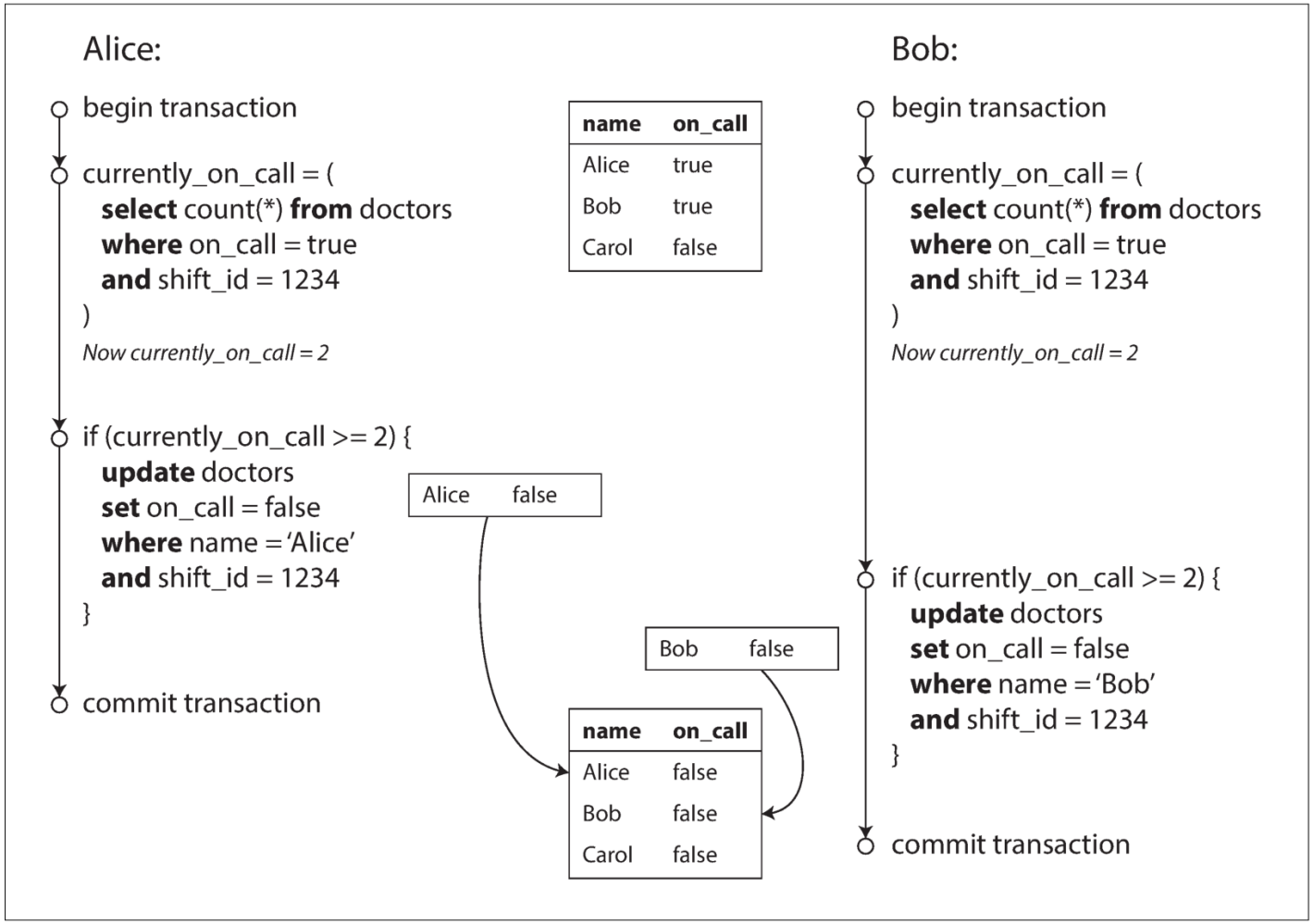

Figure 7-8. Example of write skew causing an application bug.

Write Skew and Phantoms (2)

- Write skew can occur if two transactions read the same objects, and then update some of those objects
- *Serializable isolation le*vel should solve this (treated later)
- Explicit locking:

```
BEGIN TRANSACTION;

SELECT * FROM doctors
  WHERE on_call = true
  AND shift_id = 1234 FOR UPDATE; ❶

UPDATE doctors
  SET on_call = false
  WHERE name = 'Alice'
  AND shift_id = 1234;

COMMIT;
```

As before, `FOR UPDATE` tells the database to lock all rows returned by this query.

Write Skew and Phantoms (3)

- Pattern for problem:

1. A SELECT query checks some conditions
2. Depending on the result of the first query, the application code decides how to continue
3. If the application decides to go ahead, it makes a write to the database and commits the transaction.

- The effect, where a write in one transaction changes the result of a search query in another transaction, is called a phantom
- E.g. insert a new row into a table being counted by another transaction

Serializability (1)

- Isolation levels are hard to understand, and inconsistently implemented in different databases.
- If you look at your application code, it's difficult to tell whether it is safe to run at a particular isolation level.
- There are no good tools to help us detect race conditions.
- **Serializable isolation** is usually regarded as the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, serially, without any concurrency.

Serializability (2)

- How do databases provide serializability?

1. Literally executing transactions in a serial order
2. Two-phase locking
3. Optimistic concurrency control techniques

- Why run serially? (VoltDB/H-Store, Redis, Datomic)

1. RAM became cheap enough that for many use cases is now feasible to keep the entire active dataset in memory
2. OLTP transactions are usually short and only make a small number of reads and writes

Serializability (3)

- Each transaction is through one HTTP request.
- Executed by a stored procedure
- No interaction

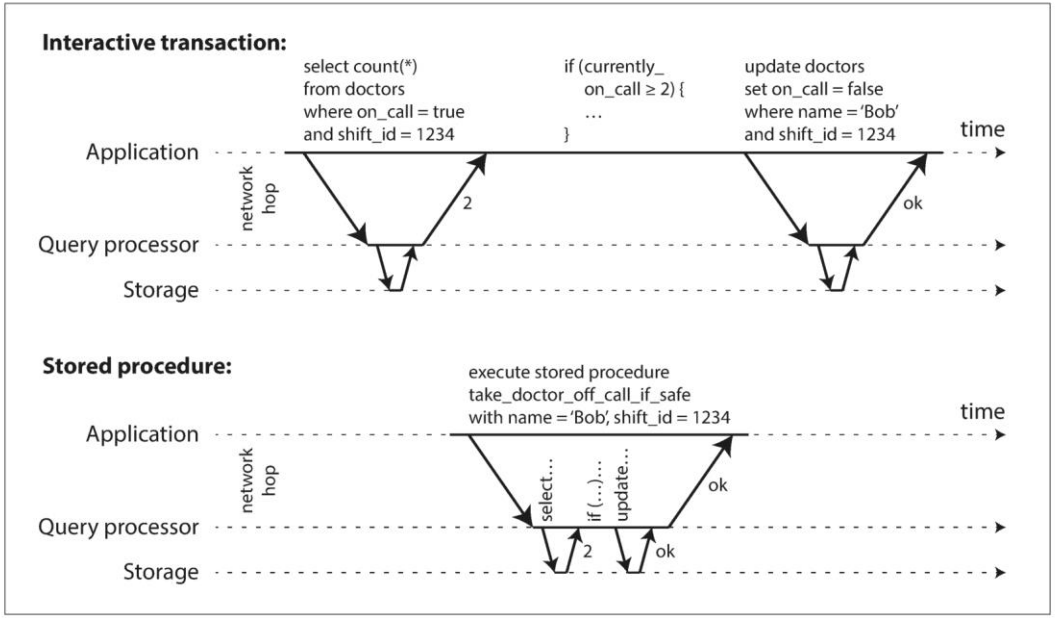

Figure 7-9. The difference between an interactive transaction and a stored procedure (using the example transaction of Figure 7-8).

Pros and cons of stored procedures

- Each database vendor has its own language for stored procedures
- Code running in a database is difficult to manage
- A database is often much more performance-sensitive than an application server
- Modern implementations of stored procedures have abandoned PL/SQL and use existing general-purpose programming languages instead (Java, Clojure, Lua, ..)
- VoltDB also uses stored procedures for replication
- Difficult to support this efficiently using partitioning. Geo-partitioning may be an exception.

Summary of serial execution

- Every transaction must be small and fast.
- The active dataset must fit in memory.
- Write throughput must be low enough to be handled on a single CPU core.
- Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.

Two-phase locking (2PL)

- Several transactions are allowed to concurrently read the same object as long as nobody is writing to it.
- If transaction A has read an object and transaction B wants to write to that object, B must wait until A commits or aborts before it can continue.
- If transaction A has written an object and transaction B wants to read that object, B must wait until A commits or aborts before it can continue.
- 2PL is used by the serializable isolation level in MySQL (InnoDB) and SQL Server, and the repeatable read isolation level in DB2

Two-Phase Locking (2PL) (2)

- T wants to read an object: T must acquire the lock in shared mode. Several transactions are allowed to hold the lock in shared mode simultaneously.
- T wants to write to an object: T must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time.
- T first reads and then writes an object, it may upgrade its shared lock to an exclusive lock.
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort).

Performance of two-phase locking

- Overhead of acquiring and releasing all those locks
- Reduced concurrency
- It may take just one slow transaction, or one transaction that accesses a lot of data and acquires many locks, to cause the rest of the system to grind to a halt.
- Deadlocks, abortions and retries

Predicate locks

- Locking a search condition, so that you can have a consistent read

```
SELECT * FROM bookings
  WHERE room_id = 123 AND
    end_time   > '2018-01-01 12:00' AND
    start_time < '2018-01-01 13:00';
```

- Transaction A must acquire a shared-mode predicate lock on the conditions of the query.
- If transaction B currently has an exclusive lock matching those conditions, A must wait until B releases its lock.
- Predicate locks apply even to objects that do not yet exist

Index-range locks

- Also called next-key locking
- Simplified approximation of predicate locking
- Lock index entries to the data set you are searching
- Shared locks
- E.g. room_id or start_time / end_time in the room booking example

Serializable Snapshot Isolation (SSI)

- 2PL is a pessimistic concurrency control
- SSI is an optimistic concurrency control
- Detecting stale MVCC reads

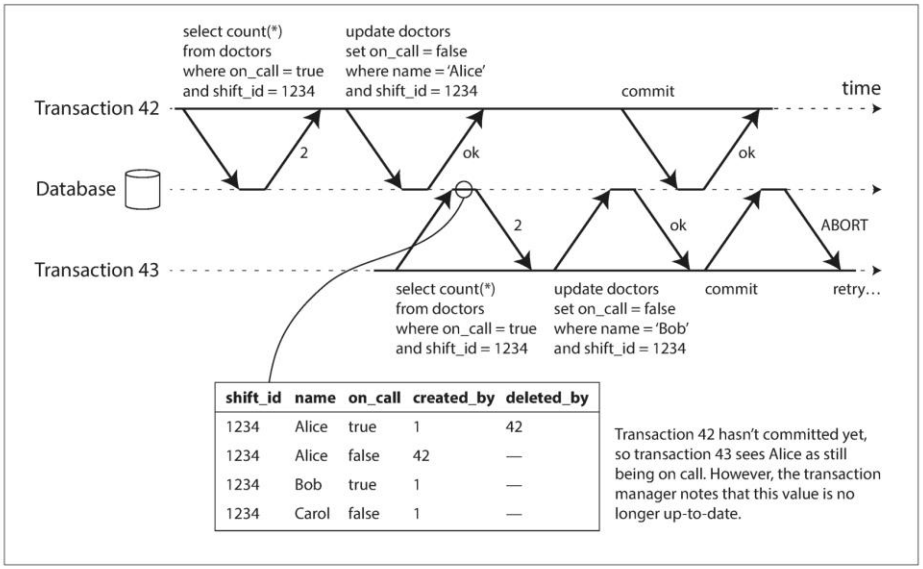

Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

Serializable Snapshot Isolation (SSI) (2)

- Detecting writes that affect prior reads

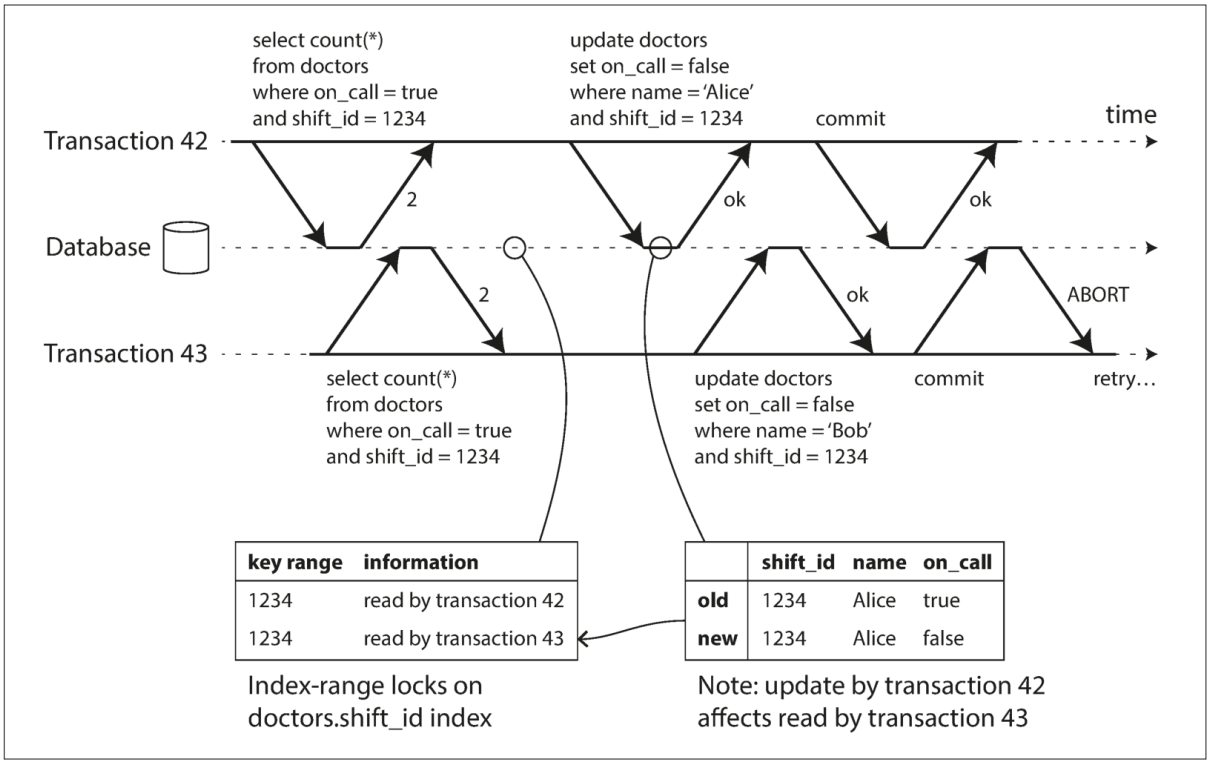

Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

SSI – Advantages and performance

- Compared to two-phase locking, the big advantage of serializable snapshot isolation is that one transaction doesn’t need to block waiting for locks held by another transaction.
- Compared to serial execution, serializable snapshot isolation is not limited to the throughput of a single CPU core
- FoundationDB distributes the detection of serialization conflicts across multiple machines

Summary

- Transactions are abstractions to simplify error handling
- Isolation levels: read committed, snapshot isolation (repeatable read), and serializable
- Problems
  - Dirty reads
  - Dirty writes
  - Read skew (non-repeatable reads)
  - Lost updates
  - Write skew
  - Phantom reads

Summary (2)

- Solutions to serializable

1. Serial execution
2. Two-Phase locking
3. Serializable snapshot isolation (SSI)