
PESSIMISTIC PENINSULA
No Phantoms
Actual serial execution
Two-phase locking
SERIALIZABLE
OPTIMISTIC PENINSULA
Serializable snapshot isolation
No Write Skew
SNAPSHOT ISOLATION
MVCC
No Read Skew
No Lost Updates
Row-level locking
No Dirty Writes
No Dirty Reads
READ COMMITTED
Temple of Multi-Object Atomicity
Atomic commit
To Distributed Transactions (Chapter 9)

Chapter 7 Transactions

- A transaction is a way for an application to group several reads and writes together into a logical unit.
- Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (commit) or it fails (abort, rollback).
- If it fails, the application can safely retry.
- There is no partial failure.
- Transaction aims to simplify the programming model for applications accessing a database. By using transactions, the application is free to ignore certain potential error scenarios and concurrency issues, because the database takes care of them instead (**safety guarantees**).

The Slippery Concept of a Transaction

The Meaning of ACID

- The safety guarantees provided by transactions are known as ACID, Atomicity, Consistency, Isolation, Durability. For fault-tolerance mechanisms in databases.
- The implementation of ACID in one database can be different from another database.
- Systems that do not meet ACID criteria are called BASE, Basically Available, Soft state, Eventual consistency. More vague definition than ACID.

Atomicity

- Atomicity/abortability: The ability to abort a transaction on error and have all writes from that transaction discarded is the defining feature of ACID atomicity.
- Atomicity describes what happens if a client wants to make several writes, but a fault occurs after some of the writes have been processed, e.g, a process crashes, a network connection is interrupted, a disk becomes full, or some integrity constraint is violated. If the writes are grouped together into an atomic transaction, and the transaction cannot be completed (committed) due to a fault, then the transaction is aborted and the database must discard or undo any writes it has made so far in that transaction.
- Atomicity simplifies this problem: if a transaction was aborted, the application can be sure that it didn't change anything, so it can safely be retried.

Consistency

In the context of ACID, consistency refers to an application-specific notion of the database being in a "good state."

You have certain statements about your data that must always be true.
Foreign key constraints? / Primary Key Constraints / Unique constraints
The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

Isolation

Isolation means concurrently executing transactions are isolated from each other.
The classic database textbooks formalize isolation as serializability, meaning each transaction pretends it is the only transaction running on the entire db. The db ensures when the transactions are committed, the result is the same as they had run serially(one after another).

Durability

Durability is the promise that once a transaction has been committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or database crashes.
To achieve this durability guarantee, write to permanent storage such as hard drive or SSD, use write-ahead log, or wait until writes or replications are complete before commit.

Single-Object and Multi-Object Operations

In ACID, atomicity and isolation describe what the database should do if a client makes several writes within the same transaction.
There should be no dirty reads and dirty writes between different transactions.
Multi-object transactions require some way of determining which read and write operations belong to the same transaction. In relational databases, this is typically done based on the client's TCP connection to the database server. Nonrelational databases don’t have transaction semantics.

Single-object writes

Storage engines should provide atomicity and isolation on the level of a single object on one node. Atomicity can be implemented using a log for crash recovery, and isolation can be implemented using a lock on each object, or other operations like compare-and-set.
These single-object operations are useful, as they prevent lost updates when several clients concurrently write to the same object.

The need for multi-object transactions

Cases need writes to multi-objects:

- In a relational data model, a row in one table has a **foreign key** reference to a row in another table. (Similarly, in a graph-like data model, a vertex has edges to other vertices.)
- In a document data model, document databases lacking join functionality also encourage **denormalization**. When denormalized information needs to be updated, you need to update several documents in one go. Transactions are very useful in this situation to prevent denormalized data from going out of sync.
- In databases with **secondary indexes(Secondary Indexing in Databases - GeeksforGeeks)**, the indexes also need to be updated every time you change a value. These indexes are different database objects from a transaction point of view.

Handling errors and aborts

A key feature of a transaction is that it can be aborted and safely retried if an error occurred. ACID databases are based on this philosophy: if the database is in danger of violating its guarantee of atomicity, isolation, or durability, it would rather abandon the transaction entirely than allow it to remain half-finished.
Object-Relational Mapping(ORM) frameworks like ActiveRecord and Django don't retry abort transactions, they usually error the results in an exception and the user gets error messages.
Problems with retrying aborted transactions:

- If the transaction actually succeeded, but the network failed while the server tried to acknowledge the successful commit to the client, then retrying the transaction causes it to be performed twice—unless you have an **additional application-level deduplication mechanism** in place.
- If the error is due to **overload**, retrying the transaction will make the problem **worse**, not better. To avoid such feedback cycles, you can limit the number of retries, use **exponential backoff**, and handle overload-related errors differently from other errors.
- It is only worth retrying after **transient errors** (for example due to **deadlock, isolation violation, temporary network interruptions, and failover**); after a **permanent error** (e.g., constraint violation) a retry would be **pointless**.
- If the transaction also has **side effects outside of the database**, those side effects may happen even if the transaction is aborted. For example, if you're sending an email, you wouldn't want to send the email again every time you retry the transaction. If you want to make sure that several different systems either commit or abort together, two-phase commit can help.
- If the **client process fails** while retrying, any data it was trying to write to the database is **lost**.

Weak Isolation Levels

Concurrency issues/Race conditions only happen when two transactions simultaneously modify the same data, or one transaction reads data which is concurrently modified by another transaction.

Read Committed

The most basic **level of transaction isolation** is read committed. It makes two **guarantees**:

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

No dirty reads

Dirty read: If a transaction has written some data to the database but has not yet committed or aborted, other transactions have visibility to that uncommitted data.
Transactions running at the **read committed isolation level** must **prevent** dirty reads.

No dirty writes

Dirty writes: If two transactions concurrently try to update the same object in a database, the first write has not yet been committed, but the later write overwrites an uncommitted value.
Transactions running at the **read committed isolation level** must **prevent** dirty writes, usually by delaying the second write until the first write's transaction has committed or aborted.

This isolation level avoids certain kinds of concurrency problems, e.g., If transactions update multiple objects, dirty writes can lead to a bad outcome.
However, read committed does not prevent the race condition between two counter increments. In this case, the second write happens after the first transaction has been committed, so it's not a dirty write. It's still incorrect, but for a different reason - Lost Updates

Implementing read committed

Databases **prevent dirty writes** by using **row-level locks**: when a transaction wants to modify a particular object (row or document), it must first **acquire a lock on that object**. It must then **hold that lock until** the transaction is **committed or aborted**. **Only one** transaction can hold the lock for any given object; if another transaction wants to write to the same object, it must **wait** until the first transaction is committed or aborted before it can **acquire** the lock and continue. This locking is done automatically by databases in read committed mode (or stronger isolation levels).

Prevent dirty reads:

1. One option would be to **use the same lock**, and to require any transaction that wants to read an object to acquire the lock and then release it again after reading. This would ensure that a read couldn't happen while an object has a dirty, uncommitted value. **However**, the approach of requiring read locks does not work well in practice, because one **long-running write transaction** can force many read-only transactions to wait until the long-running transaction has completed. **This harms the response time of read-only transactions and is bad for operability**: a slowdown in one part of an

application can have a knock-on effect in a completely different part of the application, due to waiting for locks.

2. For that reason, most databases prevent dirty reads:
   For every object that is written, the database remembers both the **old committed value** and the **new value** set by the transaction that currently holds the write lock. While the transaction is ongoing, any other transactions that **read the object are given the old value**. Only when the new value is committed do transactions switch over to reading the new value.

Snapshot Isolation and Repeatable Read

Many concurrency issues cannot be resolved by isolation level read committed, for example:

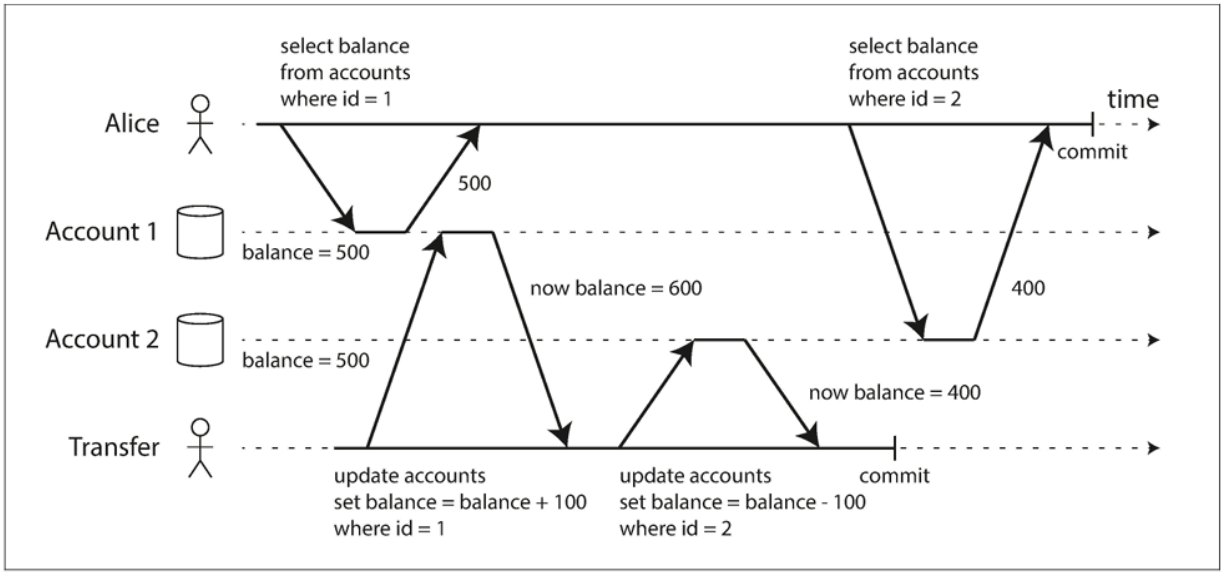


Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

This is **nonrepeatable read** or **read skew**, it is a **timing anomaly** over here.

Some situations cannot tolerate such **temporary inconsistency**:

- **Backups**
  Backup requires making a copy of the entire database, it may take hours on a large database. During backup, writes will continue to be made to the database. Thus, some parts of the backup contain an older version of the data, and other parts contain a newer version. If restored from such a backup, the inconsistencies become permanent.
- **Analytic queries and integrity checks**
  When running a query that scans over large parts of the database, which are common in analytics, or may be part of a periodic integrity check that everything is in order (monitoring for data corruption). These queries are likely to return nonsensical results if they observe parts of the database at different points in time.

Snapshot isolation is the most common solution to this problem. **Each transaction reads from a consistent snapshot of the database**. The transaction sees all the data that was committed in the database **at the start of the transaction**.
Snapshot isolation is good for **long-running, read-only queries** such as **backups** and **analytics.**

Implementing snapshot isolation

Prevent dirty writes: Use **write locks** like read committed isolation. A transaction that makes a write can block another transaction that writes to the same object.
Reads do not require any locks.
A **key principle** of snapshot isolation is **readers never block writers, and writers never block readers**.
Database can handle long-running read queries on a consistent snapshot while processing writes normally.

Databases keep several different committed versions of an object. This is called **multiversion concurrency control (MVCC).**

If a database only needed to provide read committed isolation, but not snapshot isolation, it would be sufficient to keep two versions of an object: the committed version and the overwritten-but-not-yet-committed version. However, storage engines that support snapshot isolation typically use MVCC for their read committed isolation level as well. A typical approach is that read committed uses a separate snapshot for each query, while snapshot isolation uses the same snapshot for an entire transaction.
Figure 7-7 illustrates how MVCC-based snapshot isolation is implemented in PostgreSQL (other implementations are similar). When a transaction is started, it is given a unique, always-increasing vii transaction ID (txid). Whenever a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer.
Each row in a table has a created_by field, containing the ID of the transaction that inserted this row into the table. Moreover, each row has a deleted_by field, which is initially empty. If a transaction deletes a row, the row isn't actually deleted from the database, but it is marked for deletion by setting the deleted_by field to the ID of the transaction that requested the deletion. At some later time, when it is certain that no transaction can any longer access the deleted data, a garbage collection process in the database removes any rows marked for deletion and frees their space.

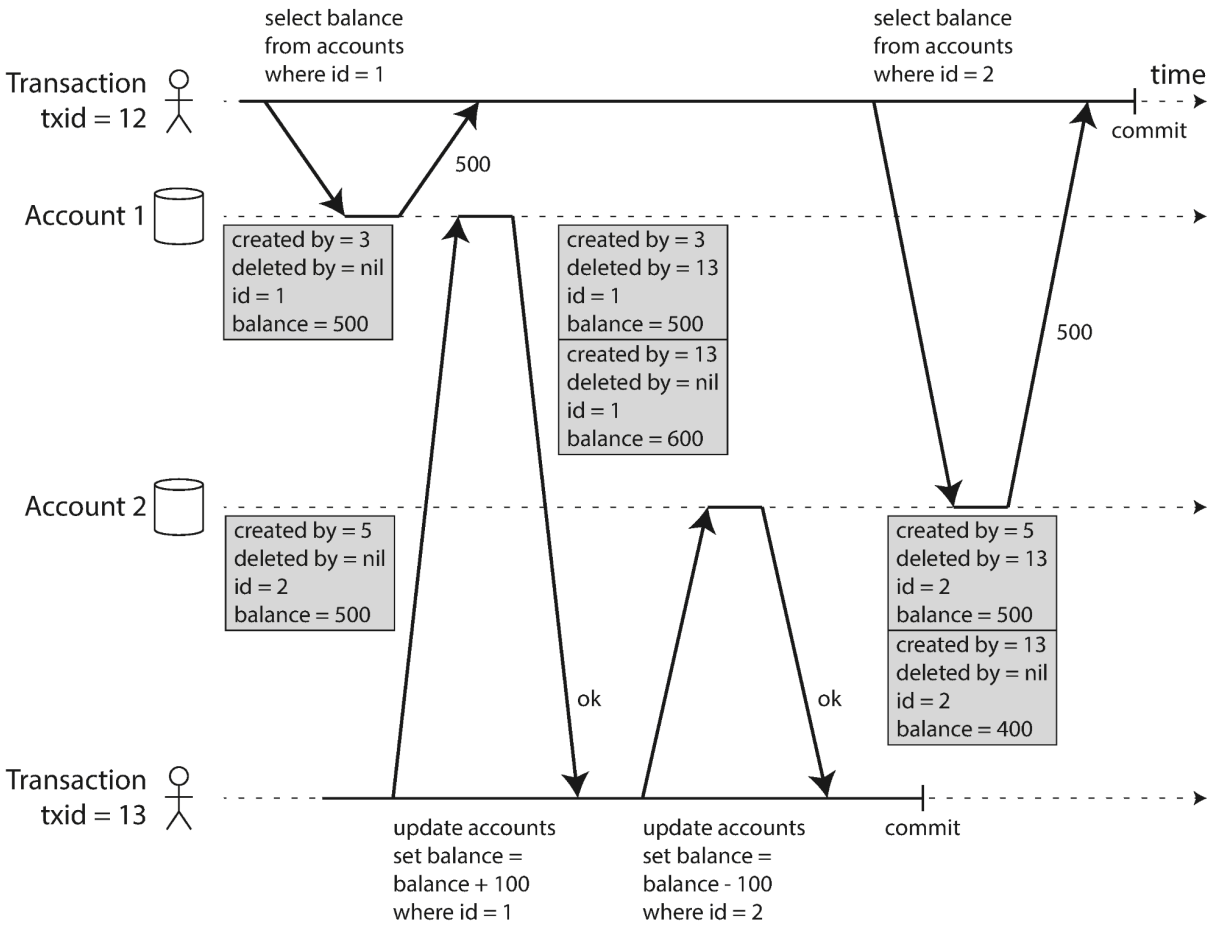


Figure 7-7. Implementing snapshot isolation using multi-version objects.

Visibility rules for observing a consistent snapshot

When a transaction reads from the database, transaction IDs are used to decide which objects it can see and which are invisible. By carefully defining visibility rules, the database can present a consistent snapshot of the database to the application.

How it works:

1. At the start of each transaction, the database makes a list of **all the other transactions** that are **in progress** (not yet committed or aborted) at that time. Any writes that those transactions have made are ignored, even if the transactions subsequently commit.
2. Any writes made by **aborted** transactions are ignored.
3. Any writes made by transactions with **a later transaction ID** are ignored.
4. All other writes are visible to the application's queries.

Put another way, an object is visible if both of the following conditions are true:

- At the time when the reader's transaction started, the transaction that created the object had already committed.
- The object is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.

A long-running transaction may continue using a snapshot for a long time, continuing to read values that (from other transactions' point of view) have long been overwritten or deleted. By never updating values in place but instead creating a new version every time a value is changed, the database can provide a consistent snapshot while incurring only a small overhead.

Indexes and snapshot isolation

Implementation details determine the performance of multiversion concurrency control.
How do indexes work in a multi-version database?

One option is to have the index simply point to all versions of an object and require an index query to filter out any object versions that are not visible to the current transaction. When garbage collection removes old object versions that are no longer visible to any transaction, the corresponding index entries can also be removed.

PostgreSQL has optimizations for avoiding index updates if different versions of the same object can fit on the same page.

Another approach is used in CouchDB, Datomic, and LMDB. Although they also use B-trees, they use an **append-only/copy-on-write** variant that does not overwrite pages of the tree when they are updated, but instead **creates a new copy of each modified page**. Parent pages, up to the root of the tree, are copied and updated to point to the new versions of their child pages. Any pages that are not affected by a write do not need to be copied, and **remain immutable**.

With append-only B-trees, every write transaction (or batch of transactions) creates a new B-tree root, and a particular root is a consistent snapshot of the database at the point in time when it was created. There is no need to filter out objects based on transaction IDs because subsequent writes cannot modify an existing B-tree; they can only create new tree roots. Need a background process for compaction and garbage collection.

Repeatable read and naming confusion

Snapshot isolation is a useful **isolation level**, especially for **read-only transactions**.

Many names: **serializable** in Oracle, **repeatable read** in PostgreSQL and MySQL.

Preventing Lost Updates

The read committed and snapshot isolation levels are primarily about the guarantees of a read-only transaction in the presence of concurrent writes. Only No dirty writes elaborated one type of two transactions writing concurrently.
Other types of conflict: lost update problem

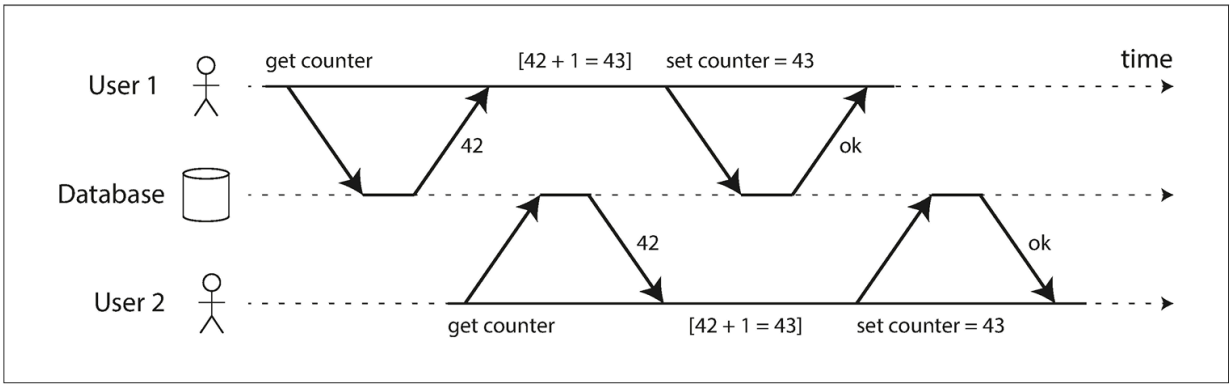


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

The lost update problem can occur **if an application reads some value from the database, modifies it, and writes back the modified value (a read-modify-write cycle). If two transactions do this concurrently, one of the modifications can be lost, because the second write does not include the first modification.**

Scenarios:

- Incrementing a counter or updating an account balance
- Making a local change to a complex value, e.g., adding an element to a list within a JSON document (requires parsing the document, making the change, and writing back the modified document)
- Two users editing a wiki page at the same time, where each user saves their changes by sending the entire page contents to the server, overwriting whatever is currently in the database

Atomic write operations

Many databases provide atomic update operations, which remove the need to implement read-modify-write **cycles** in **application code**. They are usually the **best solution** if your **code can be expressed in terms of those operations.**

For example, the following instruction is concurrency-safe in most relational databases:

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

Similarly, document databases such as MongoDB provide atomic operations for making local modifications to a part of a JSON document, and Redis provides atomic operations for modifying data structures such as priority queues.

Not all writes can easily be expressed in terms of atomic operations—for example, **updates to a wiki page involve arbitrary text editing**.

Atomic operations are usually implemented by **taking an exclusive lock on the object when it is read so that no other transaction can read it until the update has been applied**. This technique is sometimes known as **cursor stability**. Another option is to simply force all atomic operations to be executed on a **single thread**.

Explicit locking

Another option for preventing lost updates, if the database's built-in atomic operations don't provide the necessary functionality, is for the application to **explicitly lock objects that are going to be updated**. Then the application can perform a read-modify-write cycle, and if any other transaction tries to concurrently read the same object, it is forced to wait until the first read-modify-write cycle has completed.

For example, in a **multiplayer game**, several players move the same figure concurrently. In this case, an atomic operation may not be sufficient, because the application also needs to ensure that a player's move abides by the rules of the game, which involves some logic that you cannot sensibly implement as a database query. Instead, you may **use a lock to prevent two players from concurrently moving the same piece**.

Example: Explicitly locking rows to prevent lost updates

```
BEGIN TRANSACTION;

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

-- 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;
```

The FOR UPDATE clause indicates that the database lock on all rows returned by this query.

Automatically detecting lost updates

Atomic operations and locks are ways of preventing lost updates by **forcing the read-modify-write cycles to happen sequentially**. An alternative is to **allow them to execute in parallel and, if the transaction manager detects a lost update, abort the transaction and force it to retry its read-modify-write cycle.**

An **advantage** of this approach is that databases can perform this check efficiently in conjunction with **snapshot isolation**.

Lost update detection is a great feature, because it doesn't require application code to use any special database features—you may forget to use a lock or an atomic operation and thus introduce a bug, but lost update detection happens automatically and is thus less error-prone.

Compare-and-set

In databases that don't provide transactions, you sometimes find an atomic compare-and-set operation (previously mentioned in "Single-object writes"). 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. If the current value does not match what you previously read, the update has no effect, and the read-modify-write cycle must be retried.

For example, to prevent two users concurrently updating the same wiki page, you might try something like this, expecting the update to occur only if the content of the page hasn't changed since the user started editing 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 if necessary. However, if the database allows the WHERE clause to read from an old snapshot, this statement may not prevent lost updates, because the condition may be true even though another concurrent write is occurring. Check whether your database's compare-and-set operation is safe before relying on it.

Conflict resolution and replication

In **replicated databases**, preventing lost updates takes on another dimension: since they have copies of the data on multiple nodes, and the data can potentially be modified concurrently on different nodes, some additional steps need to be taken to prevent lost updates.

Locks and compare-and-set operations assume that there is a **single up-to-date** copy of the data. However, databases with **multi-leader or leaderless replication** usually allow several writes to happen concurrently and replicate them asynchronously, so they cannot guarantee that there is a single up-to-date copy of the data. Thus, techniques based on locks or compare-and-set do not apply in this context.

Instead, as discussed in "Detecting Concurrent Writes", a common approach in such replicated databases is to **allow concurrent writes to create several conflicting versions of a value** (also known as siblings), and to **use application code or special data structures** to resolve and merge these versions after the fact.

Atomic operations can work well in a replicated context, especially if they are commutative (i.e., you can apply them in a different order on different replicas, and still get the same result). For example, incrementing a counter or adding an element to a set are commutative operations.

On the other hand, the **last write wins (LWW) conflict resolution method is prone to lost updates**. Unfortunately, LWW is the **default** in many **replicated databases**.

Write Skew and Phantoms

Two kinds of race conditions, dirty writes and lost updates, can occur when different transactions concurrently try to write to the same objects.
However, there are more potential race conditions, which have more subtler conflicts.
Example:
A hospital has at least one doctor on-call, doctors can give up their shifts when at least one colleague remains on call. When Alice and Bob are both feeling unwell and request leave at same time. Since the database is using snapshot isolation, both reads return 2 doctors. Eventually at least one doctor on call policy is violated.

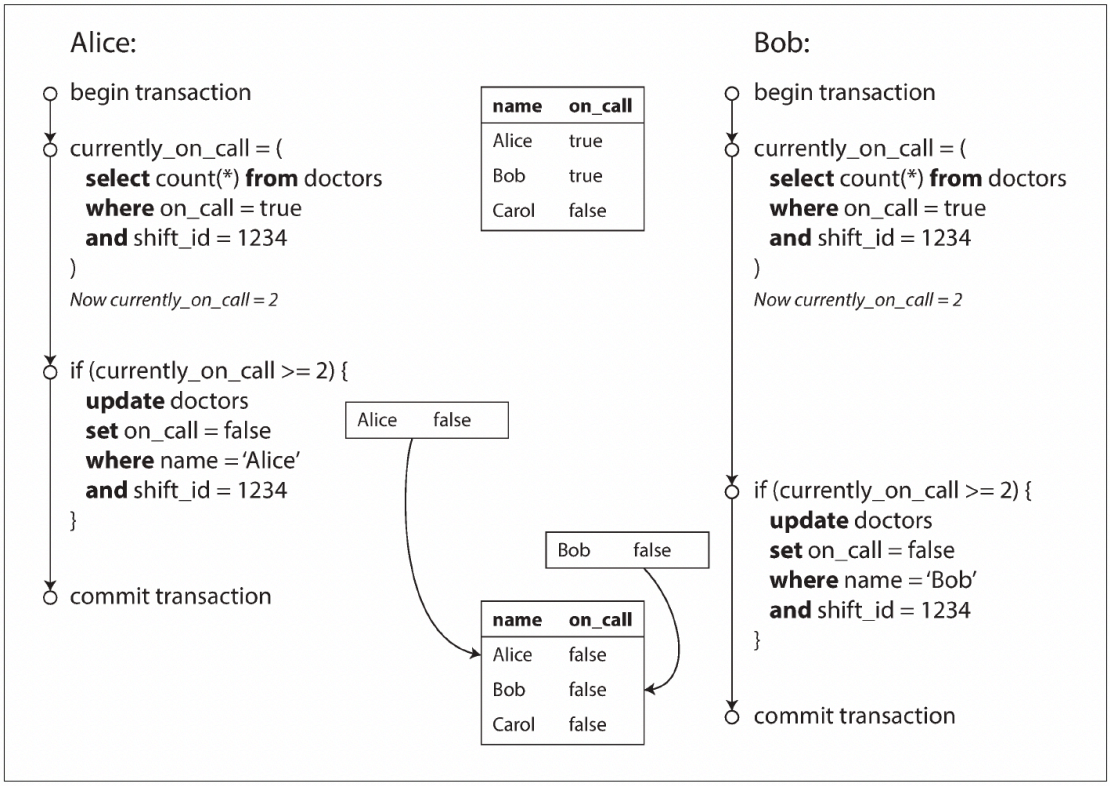


Figure 7-8. Example of write skew causing an application bug.

Characterizing write skew

Write skew: NOT a dirty write, NOT a lost update, because the **two transactions** are updating **two different objects**. It is less obvious that a conflict occurred here, but it's definitely a **race condition** and the transactions ran **concurrently**.
Write skew is a generalization of the lost update problem. Write skew can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects).

Ways to prevent write skew are more restricted:

- Atomic single-object operations don't help as multiple objects are involved.

- The automatic detection of lost updates in some implementations of snapshot isolation doesn't help either. Automatically preventing write skew **requires true serializable isolation.**
- Some databases allow you to **configure constraints**, which are then enforced by the database (e.g., uniqueness, foreign key constraints, or restrictions on a particular value). But here needs a constraint that involves multiple objects. Most databases do not have built-in support for such constraints, but may implement them with **triggers or materialized views** depending on the database.
- **The second-best option** in this case is to **explicitly lock** the rows that the transaction depends on. Use something like the following:

```
BEGIN TRANSACTION;
SELECT * FROM doctors
    WHERE on_call = true
    AND shift_id = 1234 FOR UPDATE;
UPDATE doctors
    SET on_call = false
    WHERE name = 'Alice'
    AND shift_id = 1234;
COMMIT;
```

More examples of write skew

Meeting room booking system

A meeting room booking system tries to avoid double-booking (**not safe** under snapshot isolation)

```
BEGIN TRANSACTION;
-- Check for any existing bookings that overlap with the period of noon-1pm
SELECT COUNT(*) FROM bookings
    WHERE room_id = 123 AND
        end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';
-- If the previous query returned zero:
INSERT INTO bookings
    (room_id, start_time, end_time, user_id)
    VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);
COMMIT;
```

Snapshot isolation does not prevent another user from concurrently inserting a conflicting meeting. Need serializable isolation to guarantee no conflict scheduling.

Multiplayer game

In the explicit locking example we used a lock to prevent lost updates (that is, making sure that two players can't move the same figure at the same time). However, the lock doesn't prevent players from moving **two different figures** to the **same position** on the board or potentially making some other move that violates the rules of the game. Depending on the kind of rule you

are enforcing, you might be able to use a **unique constraint**, but otherwise you're vulnerable to write skew.

Claiming a username

It is not safe under snapshot isolation. Fortunately, a **unique constraint** is a simple solution here (the second transaction that tries to register the username will be aborted due to violating the constraint).

Preventing double-spending

A service that allows users to spend money or points needs to check that a user doesn't spend more than they have. You might implement this by inserting a tentative spending item into a user's account, listing all the items in the account, and checking that the sum is positive. With write skew, it could happen that **two spending items** are inserted concurrently that together **cause the balance to go negative**, but that neither transaction notices the other.

Phantoms causing write skew

All the examples follow a similar **pattern**:

1. A SELECT query checks whether some requirement is satisfied.
2. Depending on the result of the query, the application code decides how to continue.
3. If the application decides to go ahead, it writes to the database and commits the transaction.

The steps may occur in a **different order**. For example, first make the write, then the SELECT query, and finally decide whether to abort or commit based on the result of the query.

In the case of the doctor on call example, the row being modified in step 3 was one of the rows returned in step 1, so we could make the transaction safe and avoid write skew by locking the rows in step 1 (**SELECT FOR UPDATE**). However, the other 4 examples are different: **check for the absence of rows** matching some search condition, and the **write adds a row** matching the same condition. If the query in step 1 doesn't return any rows, SELECT FOR UPDATE **can't attach locks to anything.**

A phantom: **a write in one transaction changes the result of a search query in another transaction**. Snapshot isolation avoids phantoms in read-only queries, but in read-write transactions like the above examples, phantoms can lead to tricky cases of write skew.

Materializing conflicts

Artificially introduce a lock object into the database if the problem of phantoms is that there is no object to attach the locks.

Materializing conflicts: take a phantom and turn it into a lock conflict on a concrete set of rows that exist in the database.

It's hard and error-prone, and it's ugly to let a concurrency control mechanism leak into the application data model.

A serializable isolation level is much better.

Serializability

Reasons:

- Race conditions cannot be prevented by the read committed and snapshot isolation levels.
- Isolation levels are hard to understand and databases have various implementations.
- From 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 the **strongest isolation level**.
When transactions run in parallel, the database **guarantees** the results are the same as executing one at a time without concurrency, which prevents all possible race conditions.
This part will focus on **single-node databases**, Chapter 9 will involve multiple nodes in a distributed system.

Actual Serial Execution

The simplest way of avoiding concurrency problems is to remove the concurrency entirely: to **execute only one transaction at a time, in serial order, on a single thread**.
Reasons:

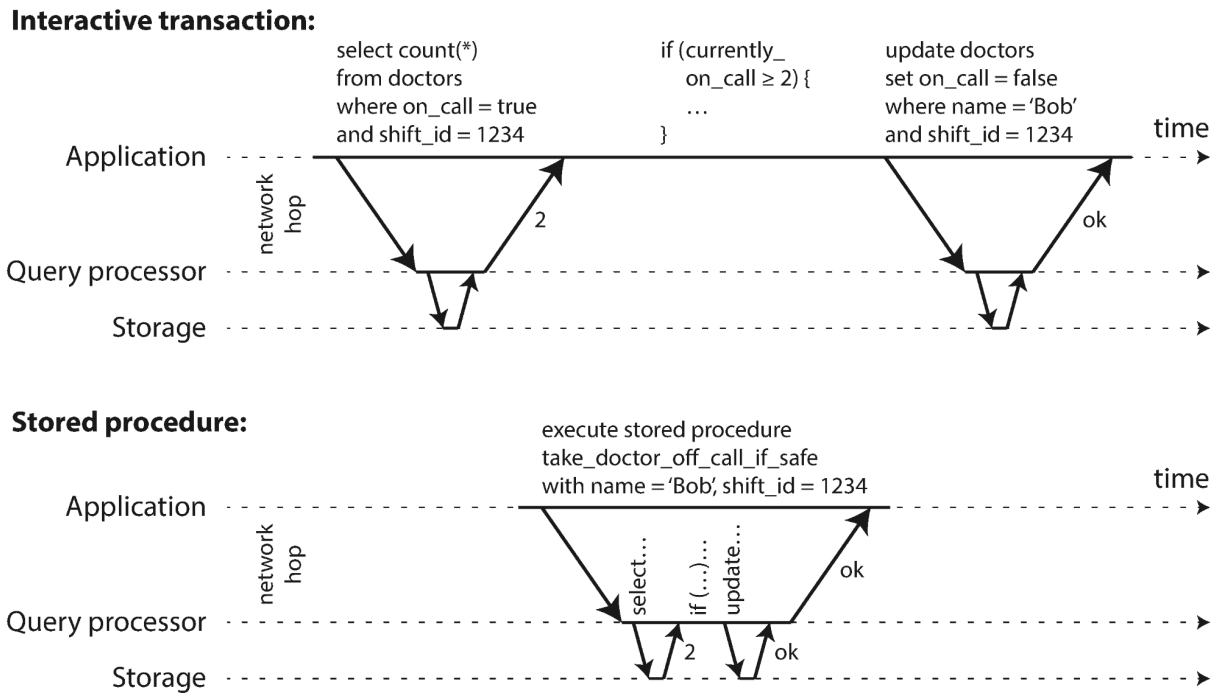
- RAM became cheap enough to keep the entire active dataset in memory
- Since OLTP transactions are short and only make a small number of reads and writes. By contrast, long-running analytic queries are typically read-only, so use snapshot isolation. They can be separated.

Encapsulating transactions in stored procedures

Problems with interactive client/server style:

- Example: a multi-stage process like booking an airline ticket. If the entire process is one transaction so that it could be committed atomically.
- **Humans are slow** to make up their minds and respond. The database may have too many concurrent transactions, most of them idle.
- Even without human factors, if transactions execute in an interactive client/server style, one statement at a time. An application makes a query, reads the result, makes another query depending on the result of the first query and so on. **The queries and results are sent back and forth between the application code and the database server. Performance is compromised** in this case.

Systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions. Instead, the application must submit the entire transaction code to the database ahead of time, as a **stored procedure**.
The difference between an interactive transaction and a stored procedure(stored procedure is faster):



Pros and cons of stored procedures

Historically, Stored procedures have bad reputations:

- Each database vendor has its own language for stored procedures.
- Code running in a database is difficult to manage: compared to an application server, it's harder to debug, more awkward to keep in version control and deploy, trickier to test, and difficult to integrate with a metrics collection system for monitoring.
- A database is often much more performance-sensitive than an application server, since a single database instance can be shared by many application servers.

But these issues can be overcome.

Partitioning

- Read-only transactions may execute elsewhere, using snapshot isolation. But for applications with high write throughput, the single-threaded transaction processor is a bottleneck.
- In order to scale to multiple CPU cores, and multiple nodes, you can potentially partition your data.
- If you find a way of partitioning the dataset so that each transaction only reads and writes data within a single partition, then each partition can have its own transaction processing thread running independently from the others.

Summary of serial execution

Serial execution of transactions is a way of achieving **serializable isolation** within certain constraints:

- Every transaction must be small and fast, because it takes only one slow transaction to stall all transaction processing.
- Limited to use cases where the active dataset can fit in memory. Rarely accessed data can be moved to disk.
- Write throughput must be low on a single CPU core, or else transactions need to be partitioned without requiring cross-partition coordination.
- Cross-partition transactions are possible but limited to certain use cases.

Two-Phase Locking (2PL)

The only widely used **algorithm for serializability** in **databases**: two-phase locking (2PL).

Locks are often used to prevent **dirty writes**: if two transactions concurrently write to the same object, the lock ensures that the second writer waits until the first one has aborted or committed its transaction before it may continue.
Two-phase locking is similar, but **lock** requirements are **stronger**. Concurrent **read** transactions are allowed if no writes on the object. When writes happen, **exclusive access** is required:

- If transaction A **reads** an object and transaction B **writes** to that object, B **waits** until A commits or aborts before continuing.
- If transaction A **writes** an object and transaction B **reads** that object, B **waits** until A commits or aborts before it can continue. (Reading an old version of the object is not acceptable under 2PL.)

Difference:
2PL: **writers block other writers and readers** , **readers block other writers**.
Snapshot isolation: **readers never block writers, and writers never block readers.**

2PL provides serializability, it protects against all the race conditions, including lost updates and write skew.

Implementation of two-phase locking

The blocking of readers and writers is implemented by **having a lock on each object in the database**. The lock can either be in **shared mode** or in **exclusive mode**.
The lock is used as follows:

- If a transaction **reads** an object, it must first acquire the lock in **shared mode**. Several transactions are allowed to hold the lock in shared mode simultaneously, but if another transaction **already** has an **exclusive** lock on the object, these transactions must wait.
- If a transaction **writes** to an object, it must first acquire the lock in **exclusive mode**. No other transaction may hold the lock at the same time (either in shared or in exclusive mode), so if there is **any existing** lock on the object, the transaction must wait.

- If a transaction **first reads and then writes** an object, it may **upgrade** its shared lock to an exclusive lock. The upgrade works the same as getting an exclusive lock directly.
- After a transaction has acquired the lock, it must continue to **hold the lock until the end** of the transaction (commit or abort). This is where the name “two-phase” comes from: the first phase (while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released.

Deadlock: Since so many locks are in use, it can happen that transaction A is stuck waiting for transaction B to release its lock and vice versa. The database **automatically detects deadlocks** between transactions and **aborts one of them** so that the others can make progress. The aborted transaction needs to be **retried** by the application.

Performance of two-phase locking

- **Downside** of two-phase locking is **performance**. Transaction **throughput and response times** of queries are significantly worse under two-phase locking than under **weak isolation**.
- **Reasons**: reduced concurrency, acquiring and releasing locks.
- Databases with 2PL can have **unstable latencies**, and can be very slow at high percentiles. One slow transaction, or one transaction that accesses a lot of data and acquires many locks, can cause the rest of the system to halt.
- Translation abortion and retry due to deadlocks should be **infrequent**.

Predicate locks

Problem of phantoms: one transaction changes the results of another transaction’s search query. A database with serializable isolation must prevent phantoms.

Meeting room example:
How do we implement this? Conceptually, we need a predicate lock. It works similarly to the shared/exclusive lock described earlier, but rather than belonging to a particular object (e.g., one row in a table), it belongs to all objects that match some search condition, such as:

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

A predicate lock restricts access as follows:

- If transaction A wants to read objects matching some condition, like in that SELECT query, it must acquire a shared-mode predicate lock on the conditions of the query. If another transaction B currently has an exclusive lock on any object matching those conditions, A must wait until B releases its lock before it is allowed to make its query.
- If transaction A wants to insert, update, or delete any object, it must first check whether either the old or the new value matches any existing predicate lock. If there is a matching

predicate lock held by transaction B, then A must wait until B has committed or aborted before it can continue.

The key idea here is that a predicate lock applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms). If two-phase locking includes predicate locks, the database prevents all forms of write skew and other race conditions, and so its isolation becomes serializable.

Index-range locks

Predicate locks do not perform well: if there are many locks by active transactions, checking for matching locks is time-consuming. Thus most databases with 2PL actually implement **index-range locking** or **next-key locking**, which is a simplified approximation of predicate locking.

It's safe to simplify a predicate by making it match a greater set of objects. For example, if you have a predicate lock for bookings of room 123 between noon and 1 p.m., you can approximate it by locking bookings for room 123 at any time, or you can approximate it by locking all rooms (not just room 123) between noon and 1 p.m. This is safe, because any write that matches the original predicate will definitely also match the approximations.

In the room bookings database:

- If **index on room_id**, the database uses the index to find existing bookings for room 123. Now the database can attach a **shared lock** to this index entry, indicating that a transaction has searched for bookings of room 123.
- Alternatively, if the database uses a **time-based index** to find existing bookings, it can attach a **shared lock** to a range of values in that index, indicating that a transaction has searched for bookings that overlap with the time period.

Either way, an approximation of the **search condition** is **attached** to one of the indexes. If another transaction wants to insert, update, or delete a booking for the same room and/or an overlapping time period, it has to **update the same part of the index**. It will encounter the **shared lock**, and **wait** until the lock is released.

Index-range locks prevent phantoms and write skew. Index-range locks are **not as precise as predicate locks** (they may lock a bigger range of objects than is strictly necessary to maintain serializability).

If no **suitable index for attaching range lock**, the database can fallback to a shared lock on the entire table.

Serializable Snapshot Isolation (SSI)

Pessimistic versus optimistic concurrency control

Decisions based on an outdated premise

Detecting stale MVCC reads

Detecting writes that affect prior reads

Performance of serializable snapshot isolation