Chapter 8. Transactions

> *Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.*
>
> —James Corbett et al., *Spanner: Google's Globally-Distributed Database* (2012)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. The GitHub repo for this book is *https://github.com/ept/ddia2-feedback*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out on GitHub.

In the harsh reality of data systems, many things can go wrong:

- The database software or hardware may fail at any time (including in the middle of a write operation).
- The application may crash at any time (including halfway through a series of operations).
- Interruptions in the network can unexpectedly cut off the application from the database, or one database node from another.
- Several clients may write to the database at the same time, overwriting each other's changes.

- A client may read data that doesn't make sense because it has only partially been updated.
- Race conditions between clients can cause surprising bugs.

In order to be reliable, a system has to deal with these faults and ensure that they don't cause catastrophic failure of the entire system. However, implementing fault-tolerance mechanisms is a lot of work. It requires a lot of careful thinking about all the things that can go wrong, and a lot of testing to ensure that the solution actually works.

For decades, *transactions* have been the mechanism of choice for simplifying these issues. 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. With transactions, error handling becomes much simpler for an application, because it doesn't need to worry about partial failure—i.e., the case where some operations succeed and some fail (for whatever reason).

If you have spent years working with transactions, they may seem obvious, but we shouldn't take them for granted. Transactions are not a law of nature; they were created with a purpose, namely 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 (we call these *safety guarantees*).

Not every application needs transactions, and sometimes there are advantages to weakening transactional guarantees or abandoning them entirely (for example, to achieve higher performance or higher availability). Some safety properties can be achieved without transactions. On the other hand, transactions can prevent a lot of grief: for example, the technical cause behind the Post Office Horizon scandal (see "How Important Is Reliability?") was probably a lack of ACID transactions in the underlying accounting system [1].

How do you figure out whether you need transactions? In order to answer that question, we first need to understand exactly what safety guarantees transactions can provide, and what costs are associated with them. Although transactions seem straightforward at first glance, there are actually many subtle but important details that come into play.

In this chapter, we will examine many examples of things that can go wrong, and explore the algorithms that databases use to guard against those issues. We will go especially deep in the area of concurrency control, discussing various kinds of race conditions that can occur and how databases implement isolation levels such as *read committed*, *snapshot isolation*, and *serializability*.

Concurrency control is relevant for both single-node and distributed databases. Later in this chapter, in "Distributed Transactions", we will examine the *two-phase commit* protocol and the challenge of achieving atomicity in a distributed transaction.

What Exactly Is a Transaction?

Almost all relational databases today, and some nonrelational databases, support transactions. Most of them follow the style that was introduced in 1975 by IBM System R, the first SQL database [2, 3, 4]. Although some implementation details have changed, the general idea has remained virtually the same for 50 years: the transaction support in MySQL, PostgreSQL, Oracle, SQL Server, etc., is uncannily similar to that of System R.

In the late 2000s, nonrelational (NoSQL) databases started gaining popularity. They aimed to improve upon the relational status quo by offering a choice of new data models (see Chapter 3), and by including replication (Chapter 6) and sharding (Chapter 7) by default. Transactions were the main casualty of this movement: many of this generation of databases abandoned transactions entirely, or redefined the word to describe a much weaker set of guarantees than had previously been understood.

The hype around NoSQL distributed databases led to a popular belief that transactions were fundamentally unscalable, and that any large-scale system would have to abandon transactions in order to maintain good performance and high availability. More recently, that belief has turned out to be wrong. So-called "NewSQL" databases such as CockroachDB [5], TiDB [6], Spanner [7], FoundationDB [8], and Yugabyte have shown that transactional systems can scale to large data volumes and high throughput. These systems combine sharding with consensus protocols (Chapter 10) to provide strong ACID guarantees at scale.

However, that doesn't mean that every system must be transactional either: like every other technical design choice, transactions have advantages and limitations. In order to understand those trade-offs, let's go into the details of the guarantees that transactions can provide—both in normal operation and in various extreme (but realistic) circumstances.

The Meaning of ACID

The safety guarantees provided by transactions are often described by the well-known acronym *ACID*, which stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*. It was coined in 1983 by Theo Härder and Andreas Reuter [9] in an effort to establish precise terminology for fault-tolerance mechanisms in databases.

However, in practice, one database's implementation of ACID does not equal another's implementation. For example, as we shall see, there is a lot of ambiguity around the meaning of *isolation* [10]. 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.

(Systems that do not meet the ACID criteria are sometimes called *BASE*, which stands for *Basically Available*, *Soft state*, and *Eventual consistency* [11]. This is even more vague than the definition of ACID. It seems that the only sensible definition of BASE is "not ACID"; i.e., it can mean almost anything you want.)

Let's dig into the definitions of atomicity, consistency, isolation, and durability, as this will let us refine our idea of transactions.

Atomicity

In general, *atomic* refers to something that cannot be broken down into smaller parts. The word means similar but subtly different things in different branches of computing. For example, in multi-threaded programming, if one thread executes an atomic operation, that means there is no way that another thread could see the half-finished result of the operation. The system can only be in the state it was before the operation or after the operation, not something in between.

By contrast, in the context of ACID, atomicity is *not* about concurrency. It does not describe what happens if several processes try to access the same data at the same time, because that is covered under the letter *I*, for *isolation* (see "Isolation").

Rather, ACID atomicity describes what happens if a client wants to make several writes, but a fault occurs after some of the writes have been processed —for example, 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.

Without atomicity, if an error occurs partway through making multiple changes, it's difficult to know which changes have taken effect and which haven't. The application could try again, but that risks making the same change twice, leading to duplicate or incorrect data. 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.

The ability to abort a transaction on error and have all writes from that transaction discarded is the defining feature of ACID atomicity. Perhaps *abortability* would have been a better term than *atomicity*, but we will stick with *atomicity* since that's the usual word.

Consistency

The word *consistency* is terribly overloaded:

- In Chapter 6 we discussed *replica consistency* and the issue of *eventual consistency* that arises in asynchronously replicated systems (see "Problems with Replication Lag").
- A *consistent snapshot* of a database, e.g. for a backup, is a snapshot of the entire database as it existed at one moment in time. More precisely, it is consistent with the happens-before relation (see "The "happens-before" relation and concurrency"): that is, if the snapshot contains a value that was written at a particular time, then it also reflects all the writes that happened before that value was written.
- *Consistent hashing* is an approach to sharding that some systems use for rebalancing (see "Consistent hashing").

- In the CAP theorem (see Chapter 10), the word *consistency* is used to mean *linearizability* (see "Linearizability").
- In the context of ACID, *consistency* refers to an application-specific notion of the database being in a "good state."

It's unfortunate that the same word is used with at least five different meanings.

The idea of ACID consistency is that you have certain statements about your data (*invariants*) that must always be true—for example, in an accounting system, credits and debits across all accounts must always be balanced. If a transaction starts with a database that is valid according to these invariants, and any writes during the transaction preserve the validity, then you can be sure that the invariants are always satisfied. (An invariant may be temporarily violated during transaction execution, but it should be satisfied again at transaction commit.)

If you want the database to enforce your invariants, you need to declare them as *constraints* as part of the schema. For example, foreign key constraints, uniqueness constraints, or check constraints (which restrict the values that can appear in an individual row) are often used to model specific types of invariants. More complex consistency requirements can sometimes be modeled using triggers or materialized views [12].

However, complex invariants can be difficult or impossible to model using the constraints that databases usually provide. In that case, it's the application's responsibility to define its transactions correctly so that they preserve consistency. If you write bad data that violates your invariants, but you haven't declared those invariants, the database can't stop you. As such, the C in ACID often depends on how the application uses the database, and it's not a property of the database alone.

Isolation

Most databases are accessed by several clients at the same time. That is no problem if they are reading and writing different parts of the database, but if they are accessing the same database records, you can run into concurrency problems (race conditions).

Figure 8-1 is a simple example of this kind of problem. Say you have two clients simultaneously incrementing a counter that is stored in a database. Each client needs to read the current value, add 1, and write the new value back (assuming there is no increment operation built into the database). In Figure 8-1 the counter should have increased from 42 to 44, because two increments happened, but it actually only went to 43 because of the race condition.

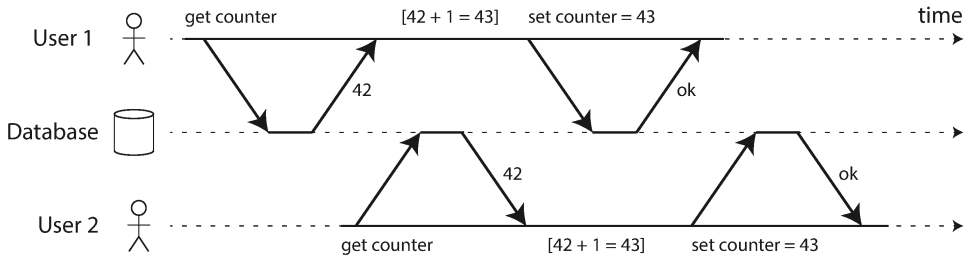

Figure 8-1. A race condition between two clients concurrently incrementing a counter.

Isolation in the sense of ACID means that concurrently executing transactions are isolated from each other: they cannot step on each other's toes. The classic database textbooks formalize isolation as *serializability*, which means that 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* (one after another), even though in reality they may have run concurrently [13].

However, serializability has a performance cost. In practice, many databases use forms of isolation that are weaker than serializability: that is, they allow concurrent transactions to interfere with each other in limited ways. Some popular databases, such as Oracle, don't even implement it (Oracle has an isolation level called "serializable," but it actually implements *snapshot isolation*, which is a weaker guarantee than serializability [10, 14]). This means that some kinds of race conditions can still occur. We will explore snapshot isolation and other forms of isolation in "Weak Isolation Levels".

Durability

The purpose of a database system is to provide a safe place where data can be stored without fear of losing it. *Durability* is the promise that once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.

In a single-node database, durability typically means that the data has been written to nonvolatile storage such as a hard drive or SSD. Regular file writes are usually buffered in memory before being sent to the disk sometime later, which means they would be lost if there is a sudden power failure; many databases therefore use the `fsync()` system call to ensure the data really has been written to disk. Databases usually also have a write-ahead log or similar (see "Making B-trees reliable"), which allows them to recover in the event that a crash occurs part way through a write. Many databases such as MySQL, MongoDB, and PostgreSQL store their data with a checksum, which allows them to detect corrupted or incomplete log entries, and thus helps them restore the database to a consistent snapshot after a crash.

In a replicated database, durability may mean that the data has been successfully copied to some number of nodes. In order to provide a durability guarantee, a database must wait until these writes or replications are complete before reporting a transaction as successfully committed. However, as discussed in "Reliability and Fault Tolerance", perfect durability does not exist: if all your hard disks and all your backups are destroyed at the same time, there's obviously nothing your database can do to save you.

Historically, durability meant writing to an archive tape. Then it was understood as writing to a disk or SSD. More recently, it has been adapted to mean replication. Which implementation is better?

The truth is, nothing is perfect:

- If you write to disk and the machine dies, even though your data isn't lost, it is inaccessible until you either fix the machine or transfer the disk to another machine. Replicated systems can remain available.
- A correlated fault—a power outage or a bug that crashes every node on a particular input—can knock out all replicas at once (see "Reliability and Fault Tolerance"), losing any data that is only in memory. Writing to disk is therefore still relevant for replicated databases.
- In an asynchronously replicated system, recent writes may be lost when the leader becomes unavailable (see "Handling Node Outages").
- When the power is suddenly cut, SSDs in particular have been shown to sometimes violate the guarantees they are supposed to provide: even `fsync` isn't guaranteed to work correctly [15]. Disk firmware can have bugs, just like any other kind of software [16, 17], e.g. causing drives to fail after exactly 32,768 hours of operation [18]. And `fsync` is hard to use; even PostgreSQL used it incorrectly for over 20 years [19, 20, 21].
- Subtle interactions between the storage engine and the filesystem implementation can lead to bugs that are hard to track down, and may cause files on disk to be corrupted after a crash [22, 23]. Filesystem errors on one replica can sometimes spread to other replicas as well [24].
- Data on disk can gradually become corrupted without this being detected [25, 26]. If data has been corrupted for some time, replicas and recent backups may also be corrupted. In this case, you will need to try to restore the data from a historical backup.
- One study of SSDs found that between 30% and 80% of drives develop at least one bad block during the first four years of operation, and only some of these can be corrected by the firmware [27]. Magnetic hard drives have a lower rate of bad sectors, but a higher rate of complete failure than SSDs.
- When a worn-out SSD (that has gone through many write/erase cycles) is disconnected from power, it can start losing data within a timescale of weeks to months, depending on the temperature [28]. This is less of a problem for drives with lower wear levels [29].

In practice, there is no one technique that can provide absolute guarantees. There are only various risk-reduction techniques, including writing to disk, replicating to remote machines, and backups—and they can and should be used together. As always, it's wise to take any theoretical "guarantees" with a healthy grain of salt.

Single-Object and Multi-Object Operations

To recap, in ACID, atomicity and isolation describe what the database should do if a client makes several writes within the same transaction:

Atomicity

> If an error occurs halfway through a sequence of writes, the transaction should be aborted, and the writes made up to that point should be discarded. In other words, the database saves you from having to worry about partial failure, by giving an all-or-nothing guarantee.

Isolation

> Concurrently running transactions shouldn't interfere with each other. For example, if one transaction makes several writes, then another transaction should see either all or none of those writes, but not some subset.

These definitions assume that you want to modify several objects (rows, documents, records) at once. Such *multi-object transactions* are often needed if several pieces of data need to be kept in sync. Figure 8-2 shows an example from an email application. To display the number of unread messages for a user, you could query something like:

```
SELECT COUNT(*) FROM emails WHERE recipient_id = 2 AND
```

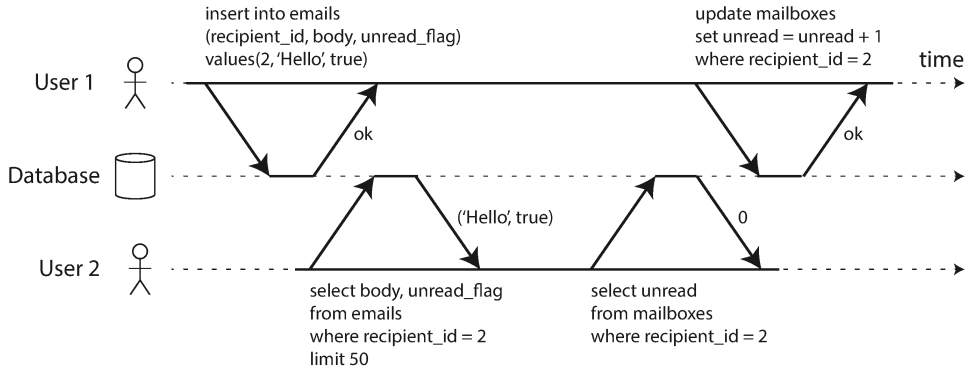

Figure 8-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a "dirty read").

However, you might find this query to be too slow if there are many emails, and decide to store the number of unread messages in a separate field (a kind of denormalization, which we discuss in "Normalization, Denormalization, and Joins"). Now, whenever a new message comes in, you have to increment the unread counter as well, and whenever a message is marked as read, you also have to decrement the unread counter.

In Figure 8-2, user 2 experiences an anomaly: the mailbox listing shows an unread message, but the counter shows zero unread messages because the counter increment has not yet happened. (If an incorrect counter in an email application seems too insignificant, think of a customer account balance instead of an unread counter, and a payment transaction instead of an email.) Isolation would have prevented this issue by ensuring that user 2 sees either both the inserted email and the updated counter, or neither, but not an inconsistent halfway point.

Figure 8-3 illustrates the need for atomicity: if an error occurs somewhere over the course of the transaction, the contents of the mailbox and the unread counter might become out of sync. In an atomic transaction, if the update to the counter fails, the transaction is aborted and the inserted email is rolled back.

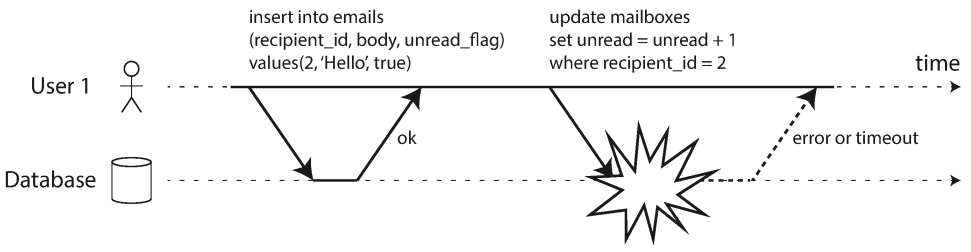

Figure 8-3. Atomicity ensures that if an error occurs any prior writes from that transaction are undone, to avoid an inconsistent state.

Multi-object transactions require some way of determining which read and write operations belong to the same transaction. In relational databases, that is typically done based on the client's TCP connection to the database server: on any particular connection, everything between a `BEGIN TRANSACTION` and a `COMMIT` statement is considered to be part of the same transaction. If the TCP connection is interrupted, the transaction must be aborted.

On the other hand, many nonrelational databases don't have such a way of grouping operations together. Even if there is a multi-object API (for example, a key-value store may have a *multi-put* operation that updates several keys in one operation), that doesn't necessarily mean it has transaction semantics: the command may succeed for some keys and fail for others, leaving the database in a partially updated state.

Single-object writes

Atomicity and isolation also apply when a single object is being changed. For example, imagine you are writing a 20 KB JSON document to a database:

- If the network connection is interrupted after the first 10 KB have been sent, does the database store that unparseable 10 KB fragment of JSON?
- If the power fails while the database is in the middle of overwriting the previous value on disk, do you end up with the old and new values spliced together?
- If another client reads that document while the write is in progress, will it see a partially updated value?

Those issues would be incredibly confusing, so storage engines almost universally aim to provide atomicity and isolation on the level of a single object (such as a key-value pair) on one node. Atomicity can be implemented using a log for crash recovery (see "Making B-trees reliable"), and isolation can be implemented using a lock on each object (allowing only one thread to access an object at any one time).

Some databases also provide more complex atomic operations, such as an increment operation, which removes the need for a read-modify-write cycle like that in Figure 8-1. Similarly popular is a *conditional write* operation, which allows a write to happen only if the value has not been concurrently changed by someone else (see "Conditional writes (compare-and-set)"),

similarly to a compare-and-set or compare-and-swap (CAS) operation in shared-memory concurrency.

NOTE

Strictly speaking, the term *atomic increment* uses the word *atomic* in the sense of multi-threaded programming. In the context of ACID, it should actually be called an *isolated* or *serializable* increment, but that's not the usual term.

These single-object operations are useful, as they can prevent lost updates when several clients try to write to the same object concurrently (see "Preventing Lost Updates"). However, they are not transactions in the usual sense of the word. For example, the "lightweight transactions" feature of Cassandra and ScyllaDB, and Aerospike's "strong consistency" mode offer linearizable (see "Linearizability") reads and conditional writes on a single object, but no guarantees across multiple objects.

The need for multi-object transactions

Do we need multi-object transactions at all? Would it be possible to implement any application with only a key-value data model and single-object operations?

There are some use cases in which single-object inserts, updates, and deletes are sufficient. However, in many other cases writes to several different objects need to be coordinated:

- In a relational data model, a row in one table often 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. Multi-object transactions allow you to ensure that these references remain valid: when inserting several records that refer to one another, the foreign keys have to be correct and up to date, or the data becomes nonsensical.
- In a document data model, the fields that need to be updated together are often within the same document, which is treated as a single object—no multi-object transactions are needed when updating a single document. However, document databases lacking join functionality also encourage denormalization (see "When to Use Which Model"). When denormalized information needs to be updated, like in the example of Figure 8-2, 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 (almost everything except pure key-value stores), 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: for example, without transaction isolation, it's possible for a record to appear in one index but not another, because the update to the second index hasn't happened yet (see "Sharding and Secondary Indexes").

Such applications can still be implemented without transactions. However, error handling becomes much more complicated without atomicity, and the lack of isolation can cause concurrency problems. We will discuss those in "Weak Isolation Levels", and explore alternative approaches in Chapter 13.

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.

Not all systems follow that philosophy, though. In particular, datastores with leaderless replication (see "Leaderless Replication") work much more on a "best effort" basis, which could be summarized as "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.

Errors will inevitably happen, but many software developers prefer to think only about the happy path rather than the intricacies of error handling. For example, popular object-relational mapping (ORM) frameworks such as Rails's ActiveRecord and Django don't retry aborted transactions—the error usually results in an exception bubbling up the stack, so any user input is thrown away and the user gets an error message. This is a shame, because the whole point of aborts is to enable safe retries.

Although retrying an aborted transaction is a simple and effective error handling mechanism, it isn't perfect:

- If the transaction actually succeeded, but the network was interrupted while the server tried to acknowledge the successful commit to the client (so it timed out from the client's point of view), 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 or high contention between concurrent transactions, 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 (see "When an Overloaded System Won't Recover").
- 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 (we will discuss this in "Two-Phase Commit (2PC)").
- If the client process crashes while retrying, any data it was trying to write to the database is lost.

Weak Isolation Levels

If two transactions don't access the same data, or if both are read-only, they can safely be run in parallel, because neither depends on the other. Concurrency issues (race conditions) only come into play when one transaction reads data that is concurrently modified by another transaction, or when the two transactions try to modify the same data.

Concurrency bugs are hard to find by testing, because such bugs are only triggered when you get unlucky with the timing. Such timing issues might occur very rarely, and are usually difficult to reproduce. Concurrency is also very difficult to reason about, especially in a large application where you don't necessarily know which other pieces of code are accessing the database. Application development is difficult enough if you just have one user at a time; having many concurrent users makes it much harder still, because any piece of data could unexpectedly change at any time.

For that reason, databases have long tried to hide concurrency issues from application developers by providing *transaction isolation*. In theory, isolation should make your life easier by letting you pretend that no concurrency is happening: *serializable* isolation means that the database guarantees that transactions have the same effect as if they ran *serially* (i.e., one at a time, without any concurrency).

In practice, isolation is unfortunately not that simple. Serializable isolation has a performance cost, and many databases don't want to pay that price [10]. It's therefore common for systems to use weaker levels of isolation, which protect against *some* concurrency issues, but not all. Those levels of isolation are much harder to understand, and they can lead to subtle bugs, but they are nevertheless used in practice [30].

Concurrency bugs caused by weak transaction isolation and race conditions are not just a theoretical problem. They have caused substantial loss of money, including bankrupting a Bitcoin exchange [31, 32, 33, 34], led to investigation by financial auditors [35], and caused customer data to be corrupted [36]. A popular comment on revelations of such problems is "Use an ACID database if you're handling financial data!"—but that misses the point. Even many popular relational database systems (which are usually considered "ACID") use weak isolation, so they wouldn't necessarily have prevented these bugs from occurring.

NOTE

Incidentally, much of the banking system relies on text files that are exchanged via secure FTP [37]. In this context, having an audit trail and some human-level fraud prevention measures is actually more important than ACID properties.

Those examples also highlight an important point: even if concurrency issues are rare in normal operation, you have to consider the possibility that an attacker deliberately sends a burst of highly concurrent requests to your API in an attempt to deliberately exploit concurrency bugs [32]. Therefore, in order to build applications that are reliable and secure, you have to ensure that such bugs are systematically prevented.

In this section we will look at several weak (nonserializable) isolation levels that are used in practice, and discuss in detail what kinds of race conditions

can and cannot occur, so that you can decide what level is appropriate to your application. Once we've done that, we will discuss serializability in detail (see "Serializability"). Our discussion of isolation levels will be informal, using examples. If you want rigorous definitions and analyses of their properties, you can find them in the academic literature [38, 39, 40, 41].

Read Committed

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

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*).

Let's discuss these two guarantees in more detail.

No dirty reads

Imagine a transaction has written some data to the database, but the transaction has not yet committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a *dirty read* [3].

Transactions running at the read committed isolation level must prevent dirty reads. This means that any writes by a transaction only become visible to others when that transaction commits (and then all of its writes become visible at once). This is illustrated in Figure 8-4, where user 1 has set $x = 3$, but user 2's *get x* still returns the old value, 2, while user 1 has not yet committed.

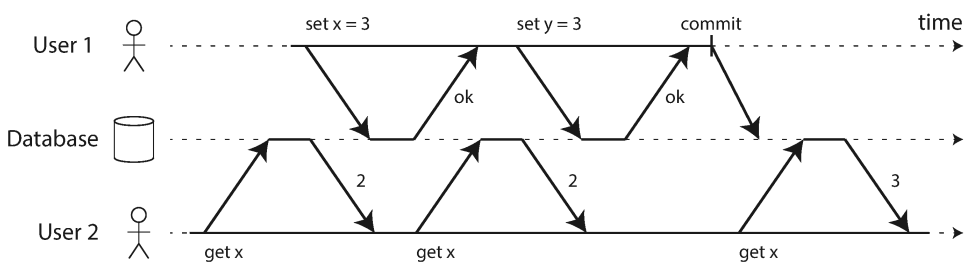

Figure 8-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

There are a few reasons why it's useful to prevent dirty reads:

- If a transaction needs to update several rows, a dirty read means that another transaction may see some of the updates but not others. For example, in Figure 8-2, the user sees the new unread email but not the updated counter. This is a dirty read of the email. Seeing the database in a partially updated state is confusing to users and may cause other transactions to take incorrect decisions.
- If a transaction aborts, any writes it has made need to be rolled back (like in Figure 8-3). If the database allows dirty reads, that means a transaction may see data that is later rolled back—i.e., which is never actually committed to the database. Any transaction that read uncommitted data would also need to be aborted, leading to a problem called *cascading aborts*.

No dirty writes

What happens if two transactions concurrently try to update the same row in a database? We don't know in which order the writes will happen, but we normally assume that the later write overwrites the earlier write.

However, what happens if the earlier write is part of a transaction that has not yet committed, so the later write overwrites an uncommitted value? This is called a *dirty write* [38]. 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.

By preventing dirty writes, this isolation level avoids some kinds of concurrency problems:

- If transactions update multiple rows, dirty writes can lead to a bad outcome. For example, consider Figure 8-5, which illustrates a used car sales website on which two people, Aaliyah and Bryce, are simultaneously trying to buy the same car. Buying a car requires two database writes: the listing on the website needs to be updated to reflect the buyer, and the sales invoice needs to be sent to the buyer. In the case of Figure 8-5, the sale is awarded to Bryce (because he performs the winning update to the `listings` table), but the invoice is sent to Aaliyah (because she performs the winning update to the `invoices` table). Read committed prevents such mishaps.
- However, read committed does *not* prevent the race condition between two counter increments in Figure 8-1. In this case, the second write happens

after the first transaction has committed, so it's not a dirty write. It's still incorrect, but for a different reason—in "Preventing Lost Updates" we will discuss how to make such counter increments safe.

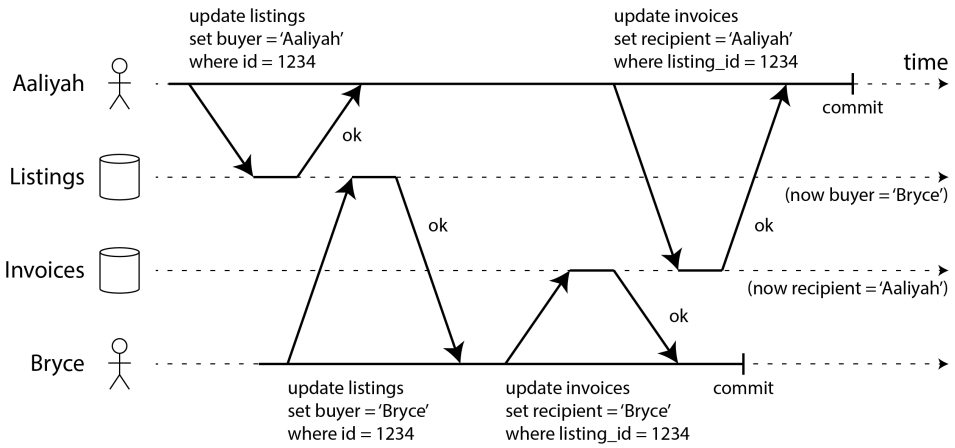

Figure 8-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Implementing read committed

Read committed is a very popular isolation level. It is the default setting in Oracle Database, PostgreSQL, SQL Server, and many other databases [10].

Most commonly, databases prevent dirty writes by using row-level locks: when a transaction wants to modify a particular row (or document or some other object), it must first acquire a lock on that row. It must then hold that lock until the transaction is committed or aborted. Only one transaction can hold the lock for any given row; if another transaction wants to write to the same row, 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).

How do we prevent dirty reads? One option would be to use the same lock, and to require any transaction that wants to read a row to briefly acquire the lock and then release it again immediately after reading. This would ensure that a read couldn't happen while a row has a dirty, uncommitted value (because during that time the lock would be held by the transaction that has made the write).

However, the approach of requiring read locks does not work well in practice, because one long-running write transaction can force many other transactions to wait until the long-running transaction has completed, even if the other transactions only read and do not write anything to the database. 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.

Nevertheless, locks are used to prevent dirty reads in some databases, such as IBM Db2 and Microsoft SQL Server in the `read_committed_snapshot=off` setting [30].

A more commonly used approach to preventing dirty reads is the one illustrated in Figure 8-4: for every row 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 row are simply given the old value. Only when the new value is committed do transactions switch over to reading the new value (see "Multi-version concurrency control (MVCC)" for more detail).

Some databases support an even weaker isolation level called *read uncommitted*. It prevents dirty writes, but does not prevent dirty reads—in other words, it immediately returns the latest written value, even if the writing transaction hasn't committed yet. This can provide better performance, since the database does not need to store two versions of the row. Moreover, it can reduce the probability (but not prevent) lost updates, which we will talk about in "Preventing Lost Updates".

Snapshot Isolation and Repeatable Read

If you look superficially at read committed isolation, you could be forgiven for thinking that it does everything that a transaction needs to do: it allows aborts (required for atomicity), it prevents reading the incomplete results of transactions, and it prevents concurrent writes from getting intermingled. Indeed, those are useful features, and much stronger guarantees than you can get from a system that has no transactions.

However, there are still plenty of ways in which you can have concurrency bugs when using this isolation level. For example, Figure 8-6 illustrates a problem that can occur with read committed.

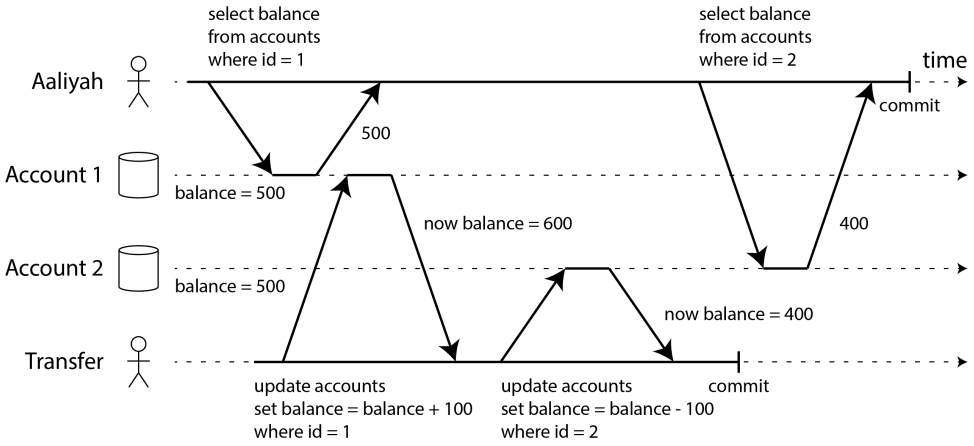

Figure 8-6. Read skew: Aaliyah observes the database in an inconsistent state.

Say Aaliyah has $1,000 of savings at a bank, split across two accounts with $500 each. Now a transaction transfers $100 from one of her accounts to the other. If she is unlucky enough to look at her list of account balances in the same moment as that transaction is being processed, she may see one account balance at a time before the incoming payment has arrived (with a balance of $500), and the other account after the outgoing transfer has been made (the new balance being $400). To Aaliyah it now appears as though she only has a total of $900 in her accounts—it seems that $100 has vanished into thin air.

This anomaly is called *read skew*, and it is an example of a *nonrepeatable read*: if Aaliyah were to read the balance of account 1 again at the end of the transaction, she would see a different value ($600) than she saw in her previous query. Read skew is considered acceptable under read committed isolation: the account balances that Aaliyah saw were indeed committed at the time when she read them.

NOTE

The term *skew* is unfortunately overloaded: we previously used it in the sense of an *unbalanced workload with hot spots* (see "Skewed Workloads and Relieving Hot Spots"), whereas here it means *timing anomaly*.

In Aaliyah's case, this is not a lasting problem, because she will most likely see consistent account balances if she reloads the online banking website a few seconds later. However, some situations cannot tolerate such temporary inconsistency:

Backups

Taking a backup requires making a copy of the entire database, which may take hours on a large database. During the time that the backup process is running, writes will continue to be made to the database. Thus, you could end up with some parts of the backup containing an older version of the data, and other parts containing a newer version. If you need to restore from such a backup, the inconsistencies (such as disappearing money) become permanent.

Analytic queries and integrity checks

Sometimes, you may want to run a query that scans over large parts of the database. Such queries are common in analytics (see "Operational Versus Analytical Systems"), 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 [38] is the most common solution to this problem. The idea is that each transaction reads from a *consistent snapshot* of the database—that is, the transaction sees all the data that was committed in the database at the start of the transaction. Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.

Snapshot isolation is a boon for long-running, read-only queries such as backups and analytics. It is very hard to reason about the meaning of a query if the data on which it operates is changing at the same time as the query is executing. When a transaction can see a consistent snapshot of the database, frozen at a particular point in time, it is much easier to understand.

Snapshot isolation is a popular feature: variants of it are supported by PostgreSQL, MySQL with the InnoDB storage engine, Oracle, SQL Server, and others, although the detailed behavior varies from one system to the next [30, 42, 43]. Some databases, such as Oracle, TiDB, and Aurora DSQL, even choose snapshot isolation as their highest isolation level. Cloud data warehouses such as BigQuery frequently use snapshot isolation as well, as it provides a point-in-time view of the database for analytical queries.

Multi-version concurrency control (MVCC)

Like read committed isolation, implementations of snapshot isolation typically use write locks to prevent dirty writes (see "Implementing read committed"), which means that a transaction that makes a write can block the progress of another transaction that writes to the same row. However, reads do not require any locks. From a performance point of view, a key principle of snapshot isolation is *readers never block writers, and writers never block readers*. This allows a database to handle long-running read queries on a consistent snapshot at the same time as processing writes normally, without any lock contention between the two.

To implement snapshot isolation, databases use a generalization of the mechanism we saw for preventing dirty reads in Figure 8-4. Instead of two versions of each row (the committed version and the overwritten-but-not-yet-committed version), the database must potentially keep several different committed versions of a row, because various in-progress transactions may need to see the state of the database at different points in time. Because it maintains several versions of a row side by side, this technique is known as *multi-version concurrency control* (MVCC).

Figure 8-7 illustrates how MVCC-based snapshot isolation is implemented in PostgreSQL [42, 44, 45] (other implementations are similar). When a transaction is started, it is given a unique, always-increasing transaction ID (`txid`). Whenever a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer. (To be precise, transaction IDs in PostgreSQL are 32-bit integers, so they overflow after approximately 4 billion transactions. The vacuum process performs cleanup to ensure that overflow does not affect the data.)

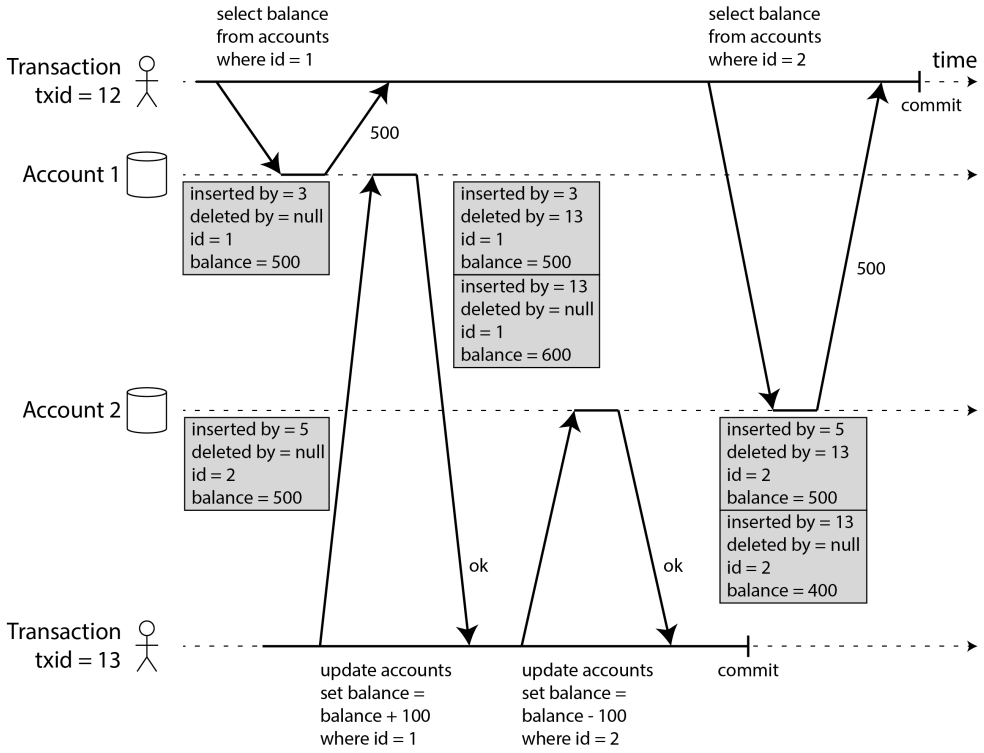

Figure 8-7. Implementing snapshot isolation using multi-version concurrency control.

Each row in a table has a `inserted_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 removed 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 or overwritten data, a garbage collection process in the database removes any rows marked for deletion and frees their space.

An update is internally translated into a delete and a insert [46]. For example, in Figure 8-7, transaction 13 deducts $100 from account 2, changing the balance from $500 to $400. The `accounts` table now actually contains two rows for account 2: a row with a balance of $500 which was marked as deleted by transaction 13, and a row with a balance of $400 which was inserted by transaction 13.

All of the versions of a row are stored within the same database heap (see "Storing values within the index"), regardless of whether the transactions that wrote them have committed or not. The versions of the same row form a linked list, going either from newest version to oldest version or the other way round, so that queries can internally iterate over all versions of a row [47, 48].

Visibility rules for observing a consistent snapshot

When a transaction reads from the database, transaction IDs are used to decide which row versions 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. This works roughly as follows [45]:

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. This ensures that we see a consistent snapshot that is not affected by another transaction committing.
2. Any writes made by transactions with a later transaction ID (i.e., which started after the current transaction started, and which are therefore not included in the list of in-progress transactions) are ignored, regardless of whether those transactions have committed.
3. Any writes made by aborted transactions are ignored, regardless of when that abort happened. This has the advantage that when a transaction aborts, we don't need to immediately remove the rows it wrote from storage, since the visibility rule filters them out. The garbage collection process can remove them later.
4. All other writes are visible to the application's queries.

These rules apply to both insertion and deletion of rows. In Figure 8-7, when transaction 12 reads from account 2, it sees a balance of $500 because the deletion of the $500 balance was made by transaction 13 (according to rule 2, transaction 12 cannot see a deletion made by transaction 13), and the insertion of the $400 balance is not yet visible (by the same rule).

Put another way, a row is visible if both of the following conditions are true:

- At the time when the reader's transaction started, the transaction that inserted the row had already committed.
- The row 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 inserting 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

How do indexes work in a multi-version database? The most common approach is that each index entry points at one of the versions of a row that matches the entry (either the oldest or the newest version). Each row version may contain a reference to the next-oldest or next-newest version. A query that uses the index must then iterate over the rows to find one that is visible, and where the value matches what the query is looking for. When garbage collection removes old row versions that are no longer visible to any transaction, the corresponding index entries can also be removed.

Many implementation details affect the performance of multi-version concurrency control [47, 48]. For example, PostgreSQL has optimizations for avoiding index updates if different versions of the same row can fit on the same page [42]. Some other databases avoid storing full copies of modified rows, and only store differences between versions to save space.

Another approach is used in CouchDB, Datomic, and LMDB. Although they also use B-trees (see "B-Trees"), they use an *immutable* (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 can be shared with the new tree [49].

With immutable 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 rows based on transaction IDs because subsequent writes cannot modify an existing B-tree; they can only create new tree roots. This approach also requires a background process for compaction and garbage collection.

Snapshot isolation, repeatable read, and naming confusion

MVCC is a commonly used implementation technique for databases, and often it is used to implement snapshot isolation. However, different databases sometimes use different terms to refer to the same thing: for example,

snapshot isolation is called “repeatable read” in PostgreSQL, and “serializable” in Oracle [30]. Sometimes different systems use the same term to mean different things: for example, while in PostgreSQL “repeatable read” means snapshot isolation, in MySQL it means an implementation of MVCC with weaker consistency than snapshot isolation [43].

The reason for this naming confusion is that the SQL standard doesn’t have the concept of snapshot isolation, because the standard is based on System R’s 1975 definition of isolation levels [3] and snapshot isolation hadn’t yet been invented then. Instead, it defines repeatable read, which looks superficially similar to snapshot isolation. PostgreSQL calls its snapshot isolation level “repeatable read” because it meets the requirements of the standard, and so they can claim standards compliance.

Unfortunately, the SQL standard’s definition of isolation levels is flawed—it is ambiguous, imprecise, and not as implementation-independent as a standard should be [38]. Even though several databases implement repeatable read, there are big differences in the guarantees they actually provide, despite being ostensibly standardized [30]. There has been a formal definition of repeatable read in the research literature [39, 40], but most implementations don’t satisfy that formal definition. And to top it off, IBM Db2 uses “repeatable read” to refer to serializability [10].

As a result, nobody really knows what repeatable read means.

Preventing Lost Updates

The read committed and snapshot isolation levels we’ve discussed so far have been primarily about the guarantees of what a read-only transaction can see in the presence of concurrent writes. We have mostly ignored the issue of two transactions writing concurrently—we have only discussed dirty writes (see “No dirty writes”), one particular type of write-write conflict that can occur.

There are several other interesting kinds of conflicts that can occur between concurrently writing transactions. The best known of these is the *lost update* problem, illustrated in Figure 8-1 with the example of two concurrent counter increments.

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. (We sometimes say that the later write *clobbers* the earlier write.) This pattern occurs in various different scenarios:

- Incrementing a counter or updating an account balance (requires reading the current value, calculating the new value, and writing back the updated value)
- 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

Because this is such a common problem, a variety of solutions have been developed [50].

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, which can be handled using algorithms discussed in "CRDTs and Operational Transformation"—but in situations where atomic operations can be used, they are usually the best choice.

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. Another option is to simply force all atomic operations to be executed on a single thread.

Unfortunately, object-relational mapping (ORM) frameworks make it easy to accidentally write code that performs unsafe read-modify-write cycles instead of using atomic operations provided by the database [51, 52, 53]. This can be a source of subtle bugs that are difficult to find by testing.

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 update or lock the same object, it is forced to wait until the first read-modify-write cycle has completed.

For example, consider a multiplayer game in which several players can 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, as illustrated in Example 8-1.

Example 8-1. 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 positic
-- of the piece that was returned by the previous SELEC
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

The `FOR UPDATE` clause indicates that the database should take a lock

❶ The `FOR UPDATE` clause indicates that the database should take a lock on all rows returned by this query.

This works, but to get it right, you need to carefully think about your application logic. It's easy to forget to add a necessary lock somewhere in the code, and thus introduce a race condition.

Moreover, if you lock multiple objects there is a risk of deadlock, where two or more transactions are waiting for each other to release their locks. Many databases automatically detect deadlocks, and abort one of the involved transactions so that the system can make progress. You can handle this situation at the application level by retrying the aborted transaction.

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. Indeed, 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. However, MySQL/InnoDB's repeatable read does not detect lost updates [30, 43]. Some authors [38, 40] argue that a database must prevent lost updates in order to qualify as providing snapshot isolation, so MySQL does not provide snapshot isolation under this definition.

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. However, you also have to retry aborted transactions at the application level.

Conditional writes (compare-and-set)

In databases that don't provide transactions, you sometimes find a *conditional write* operation that can prevent lost updates by allowing an update to happen only if the value has not changed since you last read it (previously mentioned in "Single-object writes"). 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. It is the database equivalent of an atomic *compare-and-set* or *compare-and-swap* (CAS) instruction that is supported by many CPUs.

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 databa
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. Instead of comparing the full content, you could also use a version number column that you increment on every update, and apply the update only if the current version number hasn't changed. This approach is sometimes called *optimistic locking* [54].

Note that if another transaction has concurrently modified `content`, the new content may not be visible under the MVCC visibility rules (see "Visibility rules for observing a consistent snapshot"). Many implementations of MVCC have an exception to the visibility rules for this scenario, where values written by other transactions are visible to the evaluation of the `WHERE` clause of `UPDATE` and `DELETE` queries, even though those writes are not otherwise visible in the snapshot.

Conflict resolution and replication

In replicated databases (see Chapter 6), 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 conditional write 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 conditional writes do not

apply in this context. (We will revisit this issue in more detail in "Linearizability".)

Instead, as discussed in "Dealing with Conflicting 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.

Merging conflicting values can prevent lost updates if the updates 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. That is the idea behind CRDTs, which we encountered in "CRDTs and Operational Transformation". However, some operations such as conditional writes cannot be made commutative.

On the other hand, the *last write wins* (LWW) conflict resolution method is prone to lost updates, as discussed in "Last write wins (discarding concurrent writes)". Unfortunately, LWW is the default in many replicated databases.

Write Skew and Phantoms

In the previous sections we saw *dirty writes* and *lost updates*, two kinds of race conditions that can occur when different transactions concurrently try to write to the same objects. In order to avoid data corruption, those race conditions need to be prevented—either automatically by the database, or by manual safeguards such as using locks or atomic write operations.

However, that is not the end of the list of potential race conditions that can occur between concurrent writes. In this section we will see some subtler examples of conflicts.

To begin, imagine this example: you are writing an application for doctors to manage their on-call shifts at a hospital. The hospital usually tries to have several doctors on call at any one time, but it absolutely must have at least one doctor on call. Doctors can give up their shifts (e.g., if they are sick themselves), provided that at least one colleague remains on call in that shift [55, 56].

Now imagine that Aaliyah and Bryce are the two on-call doctors for a particular shift. Both are feeling unwell, so they both decide to request leave. Unfortunately, they happen to click the button to go off call at approximately the same time. What happens next is illustrated in Figure 8-8.

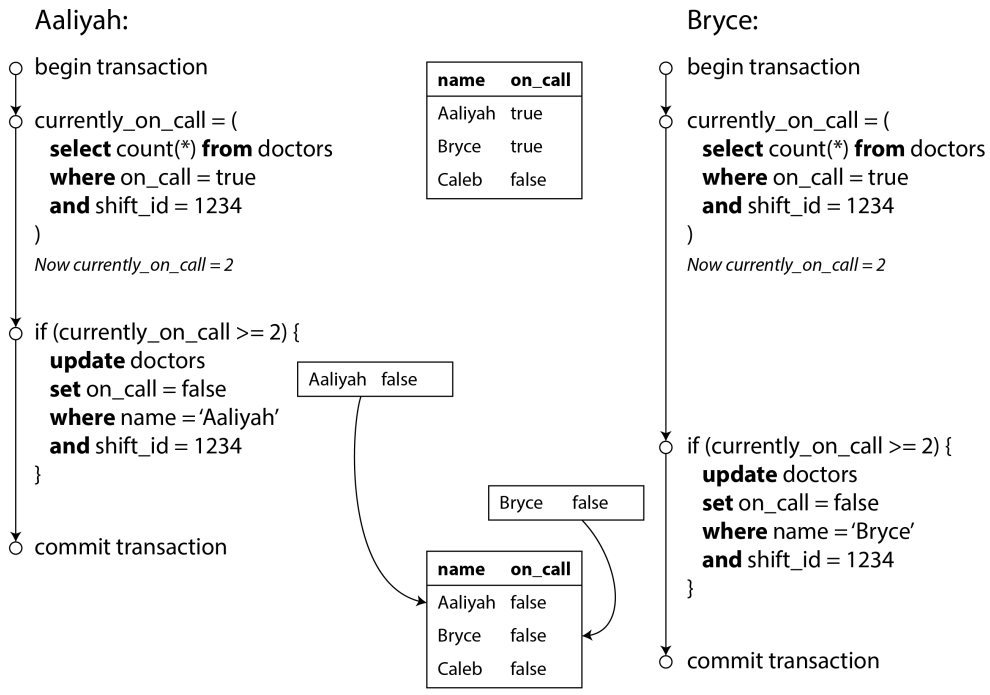

Figure 8-8. Example of write skew causing an application bug.

In each transaction, your application first checks that two or more doctors are currently on call; if yes, it assumes it's safe for one doctor to go off call. Since the database is using snapshot isolation, both checks return `2`, so both transactions proceed to the next stage. Aaliyah updates her own record to take herself off call, and Bryce updates his own record likewise. Both transactions commit, and now no doctor is on call. Your requirement of having at least one doctor on call has been violated.

Characterizing write skew

This anomaly is called *write skew* [38]. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects (Aaliyah's and Bryce's on-call records, respectively). It is less obvious that a conflict occurred here, but it's definitely a race condition: if the two transactions had run one after another, the second doctor would have been prevented from going off call. The anomalous behavior was only possible because the transactions ran concurrently.

You can think of write skew as 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). In the special case where different transactions update the same object, you get a dirty write or lost update anomaly (depending on the timing).

We saw that there are various different ways of preventing lost updates. With write skew, our options are more restricted:

- Atomic single-object operations don't help, as multiple objects are involved.
- The automatic detection of lost updates that you find in some implementations of snapshot isolation unfortunately doesn't help either: write skew is not automatically detected in PostgreSQL's repeatable read, MySQL/InnoDB's repeatable read, Oracle's serializable, or SQL Server's snapshot isolation level [30]. Automatically preventing write skew requires true serializable isolation (see "Serializability").
- 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). However, in order to specify that at least one doctor must be on call, you would need a constraint that involves multiple objects. Most databases do not have built-in support for such constraints, but you may be able to implement them with triggers or materialized views, as discussed in "Consistency" [12].
- If you can't use a serializable isolation level, the second-best option in this case is probably to explicitly lock the rows that the transaction depends on. In the doctors example, you could write something like the following:

```
BEGIN TRANSACTION;

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


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

COMMIT;


❶
```

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

More examples of write skew

Write skew may seem like an esoteric issue at first, but once you're aware of it, you may notice more situations in which it can occur. Here are some more examples:

Meeting room booking system

Say you want to enforce that there cannot be two bookings for the same meeting room at the same time [57]. When someone wants to make a booking, you first check for any conflicting bookings (i.e., bookings for the same room with an overlapping time range), and if none are found, you create the meeting (see Example 8-2).

Example 8-2. A meeting room booking system tries to avoid double-booking (not safe under snapshot isolation)

```
BEGIN TRANSACTION;

-- Check for any existing bookings that overlap w
SELECT COUNT(*) FROM bookings
  WHERE room_id = 123 AND
    end_time > '2025-01-01 12:00' AND start_time

-- If the previous query returned zero:
INSERT INTO bookings
  (room_id, start_time, end_time, user_id)
  VALUES (123, '2025-01-01 12:00', '2025-01-01 13

COMMIT;
```

Unfortunately, snapshot isolation does not prevent another user from concurrently inserting a conflicting meeting. In order to guarantee you won't get scheduling conflicts, you once again need serializable isolation.

Multiplayer game

In Example 8-1, 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

On a website where each user has a unique username, two users may try to create accounts with the same username at the same time. You may use a transaction to check whether a name is taken and, if not, create an account with that name. However, like in the previous examples, that 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 of these examples follow a similar pattern:

1. A `SELECT` query checks whether some requirement is satisfied by searching for rows that match some search condition (there are at least two doctors on call, there are no existing bookings for that room at that time, the position on the board doesn't already have another figure on it, the username isn't already taken, there is still money in the account).
2. Depending on the result of the first query, the application code decides how to continue (perhaps to go ahead with the operation, or perhaps to report an error to the user and abort).

3. If the application decides to go ahead, it makes a write (`INSERT` , `UPDATE` , or `DELETE`) to the database and commits the transaction. The effect of this write changes the precondition of the decision of step 2. In other words, if you were to repeat the `SELECT` query from step 1 after committing the write, you would get a different result, because the write changed the set of rows matching the search condition (there is now one fewer doctor on call, the meeting room is now booked for that time, the position on the board is now taken by the figure that was moved, the username is now taken, there is now less money in the account).

The steps may occur in a different order. For example, you could 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 four examples are different: they 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 [58].

This effect, where a write in one transaction changes the result of a search query in another transaction, is called a *phantom* [4]. Snapshot isolation avoids phantoms in read-only queries, but in read-write transactions like the examples we discussed, phantoms can lead to particularly tricky cases of write skew. The SQL generated by ORMs is also prone to write skew [52, 53].

Materializing conflicts

If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?

For example, in the meeting room booking case you could imagine creating a table of time slots and rooms. Each row in this table corresponds to a particular room for a particular time period (say, 15 minutes). You create rows for all possible combinations of rooms and time periods ahead of time, e.g. for the next six months.

Now a transaction that wants to create a booking can lock (`SELECT FOR UPDATE`) the rows in the table that correspond to the desired room and time period. After it has acquired the locks, it can check for overlapping bookings and insert a new booking as before. Note that the additional table isn't used to store information about the booking—it's purely a collection of locks which is used to prevent bookings on the same room and time range from being modified concurrently.

This approach is called *materializing conflicts*, because it takes a phantom and turns it into a lock conflict on a concrete set of rows that exist in the database [14]. Unfortunately, it can be hard and error-prone to figure out how to materialize conflicts, and it's ugly to let a concurrency control mechanism leak into the application data model. For those reasons, materializing conflicts should be considered a last resort if no alternative is possible. A serializable isolation level is much preferable in most cases.

Serializability

In this chapter we have seen several examples of transactions that are prone to race conditions. Some race conditions are prevented by the read committed and snapshot isolation levels, but others are not. We encountered some particularly tricky examples with write skew and phantoms. It's a sad situation:

- Isolation levels are hard to understand, and inconsistently implemented in different databases (e.g., the meaning of "repeatable read" varies significantly).
- If you look at your application code, it's difficult to tell whether it is safe to run at a particular isolation level—especially in a large application, where you might not be aware of all the things that may be happening concurrently.
- There are no good tools to help us detect race conditions. In principle, static analysis may help [35], but research techniques have not yet found their way into practical use. Testing for concurrency issues is hard, because they are usually nondeterministic—problems only occur if you get unlucky with the timing.

This is not a new problem—it has been like this since the 1970s, when weak isolation levels were first introduced [3]. All along, the answer from

researchers has been simple: use *serializable* isolation!

Serializable isolation is 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. Thus, the database guarantees that if the transactions behave correctly when run individually, they continue to be correct when run concurrently—in other words, the database prevents *all* possible race conditions.

But if serializable isolation is so much better than the mess of weak isolation levels, then why isn't everyone using it? To answer this question, we need to look at the options for implementing serializability, and how they perform. Most databases that provide serializability today use one of three techniques, which we will explore in the rest of this chapter:

- Literally executing transactions in a serial order (see "Actual Serial Execution")
- Two-phase locking (see "Two-Phase Locking (2PL)"), which for several decades was the only viable option
- Optimistic concurrency control techniques such as serializable snapshot isolation (see "Serializable Snapshot Isolation (SSI)")

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. By doing so, we completely sidestep the problem of detecting and preventing conflicts between transactions: the resulting isolation is by definition serializable.

Even though this seems like an obvious idea, it was only in the 2000s that database designers decided that a single-threaded loop for executing transactions was feasible [59]. If multi-threaded concurrency was considered essential for getting good performance during the previous 30 years, what changed to make single-threaded execution possible?

Two developments caused this rethink:

- RAM became cheap enough that for many use cases it is now feasible to keep the entire active dataset in memory (see "Keeping everything in

memory”). When all data that a transaction needs to access is in memory, transactions can execute much faster than if they have to wait for data to be loaded from disk.

- Database designers realized that OLTP transactions are usually short and only make a small number of reads and writes (see “Operational Versus Analytical Systems”). By contrast, long-running analytic queries are typically read-only, so they can be run on a consistent snapshot (using snapshot isolation) outside of the serial execution loop.

The approach of executing transactions serially is implemented in VoltDB/H-Store, Redis, and Datomic, for example [60, 61, 62]. A system designed for single-threaded execution can sometimes perform better than a system that supports concurrency, because it can avoid the coordination overhead of locking. However, its throughput is limited to that of a single CPU core. In order to make the most of that single thread, transactions need to be structured differently from their traditional form.

Encapsulating transactions in stored procedures

In the early days of databases, the intention was that a database transaction could encompass an entire flow of user activity. For example, booking an airline ticket is a multi-stage process (searching for routes, fares, and available seats; deciding on an itinerary; booking seats on each of the flights of the itinerary; entering passenger details; making payment). Database designers thought that it would be neat if that entire process was one transaction so that it could be committed atomically.

Unfortunately, humans are very slow to make up their minds and respond. If a database transaction needs to wait for input from a user, the database needs to support a potentially huge number of concurrent transactions, most of them idle. Most databases cannot do that efficiently, and so almost all OLTP applications keep transactions short by avoiding interactively waiting for a user within a transaction. On the web, this means that a transaction is committed within the same HTTP request—a transaction does not span multiple requests. A new HTTP request starts a new transaction.

Even though the human has been taken out of the critical path, transactions have continued to be executed in an interactive client/server style, one statement at a time. An application makes a query, reads the result, perhaps 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 (running on one machine) and the database server (on another machine).

In this interactive style of transaction, a lot of time is spent in network communication between the application and the database. If you were to disallow concurrency in the database and only process one transaction at a time, the throughput would be dreadful because the database would spend most of its time waiting for the application to issue the next query for the current transaction. In this kind of database, it's necessary to process multiple transactions concurrently in order to get reasonable performance.

For this reason, systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions. Instead, the application must either limit itself to transactions containing a single statement, or submit the entire transaction code to the database ahead of time, as a *stored procedure* [63].

The differences between interactive transactions and stored procedures is illustrated in Figure 8-9. Provided that all data required by a transaction is in memory, the stored procedure can execute very quickly, without waiting for any network or disk I/O.

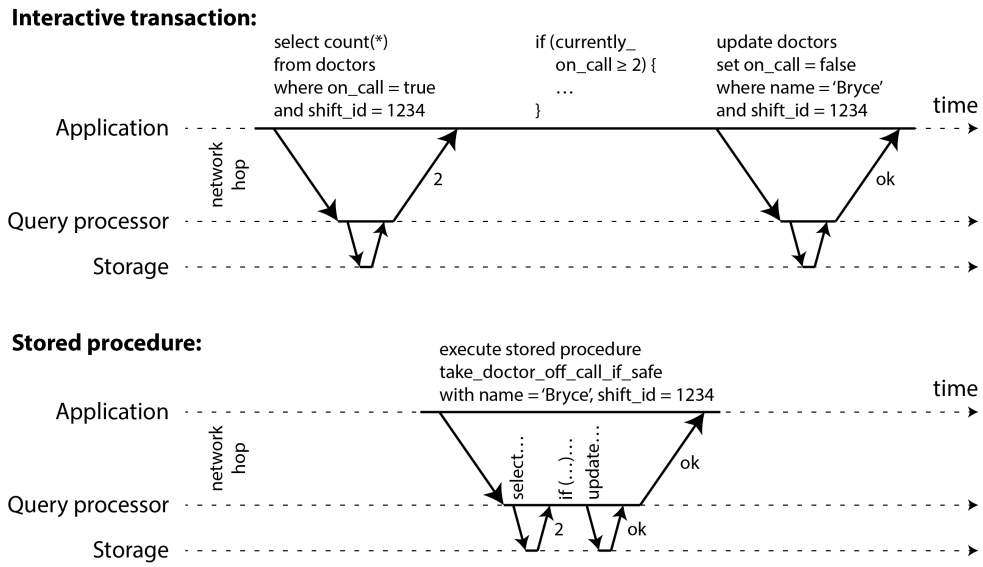

Figure 8-9. The difference between an interactive transaction and a stored procedure (using the example transaction of Figure 8-8).

Pros and cons of stored procedures

Stored procedures have existed for some time in relational databases, and they have been part of the SQL standard (SQL/PSM) since 1999. They have gained a somewhat bad reputation, for various reasons:

- Traditionally, each database vendor had its own language for stored procedures (Oracle has PL/SQL, SQL Server has T-SQL, PostgreSQL has PL/pgSQL, etc.). These languages haven't kept up with developments in general-purpose programming languages, so they look quite ugly and archaic from today's point of view, and they lack the ecosystem of libraries that you find with most programming languages.
- 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, because a single database instance is often shared by many application servers. A badly written stored procedure (e.g., using a lot of memory or CPU time) in a database can cause much more trouble than equivalent badly written code in an application server.
- In a multitenant system that allows tenants to write their own stored procedures, it's a security risk to execute untrusted code in the same process as the database kernel [64].

However, those issues can be overcome. Modern implementations of stored procedures have abandoned PL/SQL and use existing general-purpose programming languages instead: VoltDB uses Java or Groovy, Datomic uses Java or Clojure, Redis uses Lua, and MongoDB uses Javascript.

Stored procedures are also useful in cases where application logic can't easily be embedded elsewhere. Applications that use GraphQL, for example, might directly expose their database through a GraphQL proxy. If the proxy doesn't support complex validation logic, you can embed such logic directly in the database using a stored procedure. If the database doesn't support stored procedures, you would have to deploy a validation service between the proxy and the database to do validation.

With stored procedures and in-memory data, executing all transactions on a single thread becomes feasible. When stored procedures don't need to wait for I/O and avoid the overhead of other concurrency control mechanisms, they can achieve quite good throughput on a single thread.

VoltDB also uses stored procedures for replication: instead of copying a transaction's writes from one node to another, it executes the same stored procedure on each replica. VoltDB therefore requires that stored procedures

are *deterministic* (when run on different nodes, they must produce the same result). If a transaction needs to use the current date and time, for example, it must do so through special deterministic APIs (see "Durable Execution and Workflows" for more details on deterministic operations). This approach is called *state machine replication*, and we will return to it in Chapter 10.

Sharding

Executing all transactions serially makes concurrency control much simpler, but limits the transaction throughput of the database to the speed of a single CPU core on a single machine. Read-only transactions may execute elsewhere, using snapshot isolation, but for applications with high write throughput, the single-threaded transaction processor can become a serious bottleneck.

In order to scale to multiple CPU cores, and multiple nodes, you can shard your data (see Chapter 7), which is supported in VoltDB. If you can find a way of sharding your dataset so that each transaction only needs to read and write data within a single shard, then each shard can have its own transaction processing thread running independently from the others. In this case, you can give each CPU core its own shard, which allows your transaction throughput to scale linearly with the number of CPU cores [61].

However, for any transaction that needs to access multiple shards, the database must coordinate the transaction across all the shards that it touches. The stored procedure needs to be performed in lock-step across all shards to ensure serializability across the whole system.

Since cross-shard transactions have additional coordination overhead, they are vastly slower than single-shard transactions. VoltDB reports a throughput of about 1,000 cross-shard writes per second, which is orders of magnitude below its single-shard throughput and cannot be increased by adding more machines [63]. More recent research has explored ways of making multi-shard transactions more scalable [65].

Whether transactions can be single-shard depends very much on the structure of the data used by the application. Simple key-value data can often be sharded very easily, but data with multiple secondary indexes is likely to require a lot of cross-shard coordination (see "Sharding and Secondary Indexes").

Summary of serial execution

Serial execution of transactions has become a viable 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.
- It is most appropriate in situations where the active dataset can fit in memory. Rarely accessed data could potentially be moved to disk, but if it needed to be accessed in a single-threaded transaction, the system would get very slow.
- Write throughput must be low enough to be handled on a single CPU core, or else transactions need to be sharded without requiring cross-shard coordination.
- Cross-shard transactions are possible, but their throughput is hard to scale.

Two-Phase Locking (2PL)

For around 30 years, there was only one widely used algorithm for serializability in databases: *two-phase locking* (2PL), sometimes called *strong strict two-phase locking* (SS2PL) to distinguish it from other variants of 2PL.

2PL IS NOT 2PC

Two-phase *locking* (2PL) and two-phase *commit* (2PC) are two very different things. 2PL provides serializable isolation, whereas 2PC provides atomic commit in a distributed database (see "Two-Phase Commit (2PC)"). To avoid confusion, it's best to think of them as entirely separate concepts and to ignore the unfortunate similarity in the names.

We saw previously that locks are often used to prevent dirty writes (see "No dirty writes"): if two transactions concurrently try to write to the same object, the lock ensures that the second writer must wait until the first one has finished its transaction (aborted or committed) before it may continue.

Two-phase locking is similar, but makes the lock requirements much stronger. Several transactions are allowed to concurrently read the same object as long as nobody is writing to it. But as soon as anyone wants to write (modify or delete) an object, exclusive access is required:

- 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. (This ensures that B can't change the object unexpectedly behind A's back.)
- 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. (Reading an old version of the object, like in Figure 8-4, is not acceptable under 2PL.)

In 2PL, writers don't just block other writers; they also block readers and vice versa. Snapshot isolation has the mantra *readers never block writers, and writers never block readers* (see "Multi-version concurrency control (MVCC)"), which captures this key difference between snapshot isolation and two-phase locking. On the other hand, because 2PL provides serializability, it protects against all the race conditions discussed earlier, including lost updates and write skew.

Implementation of two-phase locking

2PL is used by the serializable isolation level in MySQL (InnoDB) and SQL Server, and the repeatable read isolation level in Db2 [30].

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* (also known as a *multi-reader single-writer* lock). The lock is used as follows:

- If a transaction wants to read 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 wants to write 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 (*growing* phase, while the

transaction is executing) is when the locks are acquired, and the second phase (*shrinking* phase, at the end of the transaction) is when all the locks are released. The two phases must not overlap: once a lock is released, no new locks may be acquired in a transaction.

Since so many locks are in use, it can happen quite easily that transaction A is stuck waiting for transaction B to release its lock, and vice versa. This situation is called *deadlock*. 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

The big downside of two-phase locking, and the reason why it hasn't been the default for most systems since the 1970s, is performance: transaction throughput and response times of queries are significantly worse under two-phase locking than under weak isolation.

This is partly due to the overhead of acquiring and releasing all those locks, but more importantly due to reduced concurrency. By design, if two concurrent transactions try to do anything that may in any way result in a race condition, one has to wait for the other to complete.

For example, if you have a transaction that needs to read an entire table (e.g. a backup, analytics query, or integrity check, as discussed in "Snapshot Isolation and Repeatable Read"), that transaction has to take a shared lock on the entire table. Therefore, the reading transaction first has to wait until all in-progress transactions writing to that table have completed; then, while the whole table is being read (which may take a long time on a large table), all other transactions that want to write to that table are blocked until the big read-only transaction commits. In effect, the database becomes unavailable for writes for an extended time.

For this reason, databases running 2PL can have quite unstable latencies, and they can be very slow at high percentiles (see "Describing Performance") if there is contention in the workload. 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. Transaction timeouts and slow query monitoring are used to detect and limit a misbehaving query.

Although deadlocks can happen with the lock-based read committed isolation level, they occur much more frequently under 2PL serializable isolation (depending on the access patterns of your transaction). This can be an additional performance problem: when a transaction is aborted due to deadlock and is retried, it needs to do its work all over again. If deadlocks are frequent, this can mean significant wasted effort.

Predicate locks

In the preceding description of locks, we glossed over a subtle but important detail. In "Phantoms causing write skew" we discussed the problem of *phantoms*—that is, one transaction changing the results of another transaction's search query. A database with serializable isolation must prevent phantoms.

In the meeting room booking example this means that if one transaction has searched for existing bookings for a room within a certain time window (see Example 8-2), another transaction is not allowed to concurrently insert or update another booking for the same room and time range. (It's okay to concurrently insert bookings for other rooms, or for the same room at a different time that doesn't affect the proposed booking.)

How do we implement this? Conceptually, we need a *predicate lock* [4]. 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   > '2025-01-01 12:00' AND
    start_time < '2025-01-01 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

Unfortunately, predicate locks do not perform well: if there are many locks by active transactions, checking for matching locks becomes time-consuming. For that reason, most databases with 2PL actually implement *index-range locking* (also known as *next-key locking*), which is a simplified approximation of predicate locking [56, 66].

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 you would probably have an index on the `room_id` column, and/or indexes on `start_time` and `end_time` (otherwise the preceding query would be very slow on a large database):

- Say your index is on `room_id`, and the database uses this index to find existing bookings for room 123. Now the database can simply 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 of noon to 1 p.m. on January 1, 2025.

Either way, an approximation of the search condition is attached to one of the indexes. Now, if another transaction wants to insert, update, or delete a booking for the same room and/or an overlapping time period, it will have to update the same part of the index. In the process of doing so, it will encounter the shared lock, and it will be forced to wait until the lock is released.

This provides effective protection against phantoms and write skew. Index-range locks are not as precise as predicate locks would be (they may lock a bigger range of objects than is strictly necessary to maintain serializability), but since they have much lower overheads, they are a good compromise.

If there is no suitable index where a range lock can be attached, the database can fall back to a shared lock on the entire table. This will not be good for performance, since it will stop all other transactions writing to the table, but it's a safe fallback position.

Serializable Snapshot Isolation (SSI)

This chapter has painted a bleak picture of concurrency control in databases. On the one hand, we have implementations of serializability that don't perform well (two-phase locking) or don't scale well (serial execution). On the other hand, we have weak isolation levels that have good performance, but are prone to various race conditions (lost updates, write skew, phantoms, etc.). Are serializable isolation and good performance fundamentally at odds with each other?

It seems not: an algorithm called *serializable snapshot isolation* (SSI) provides full serializability with only a small performance penalty compared to snapshot isolation. SSI is comparatively new: it was first described in 2008 [55, 67].

Today SSI and similar algorithms are used in single-node databases (the serializable isolation level in PostgreSQL [56], SQL Server's In-Memory OLTP/Hekaton [68], and HyPer [69]), distributed databases (CockroachDB [5] and FoundationDB [8]), and embedded storage engines such as BadgerDB.

Pessimistic versus optimistic concurrency control

Two-phase locking is a so-called *pessimistic* concurrency control mechanism: it is based on the principle that if anything might possibly go wrong (as

indicated by a lock held by another transaction), it's better to wait until the situation is safe again before doing anything. It is like *mutual exclusion*, which is used to protect data structures in multi-threaded programming.

Serial execution is, in a sense, pessimistic to the extreme: it is essentially equivalent to each transaction having an exclusive lock on the entire database (or one shard of the database) for the duration of the transaction. We compensate for the pessimism by making each transaction very fast to execute, so it only needs to hold the "lock" for a short time.

By contrast, serializable snapshot isolation is an *optimistic* concurrency control technique. Optimistic in this context means that instead of blocking if something potentially dangerous happens, transactions continue anyway, in the hope that everything will turn out all right. When a transaction wants to commit, the database checks whether anything bad happened (i.e., whether isolation was violated); if so, the transaction is aborted and has to be retried. Only transactions that executed serializably are allowed to commit.

Optimistic concurrency control is an old idea [70], and its advantages and disadvantages have been debated for a long time [71]. It performs badly if there is high contention (many transactions trying to access the same objects), as this leads to a high proportion of transactions needing to abort. If the system is already close to its maximum throughput, the additional transaction load from retried transactions can make performance worse.

However, if there is enough spare capacity, and if contention between transactions is not too high, optimistic concurrency control techniques tend to perform better than pessimistic ones. Contention can be reduced with commutative atomic operations: for example, if several transactions concurrently want to increment a counter, it doesn't matter in which order the increments are applied (as long as the counter isn't read in the same transaction), so the concurrent increments can all be applied without conflicting.

As the name suggests, SSI is based on snapshot isolation—that is, all reads within a transaction are made from a consistent snapshot of the database (see "Snapshot Isolation and Repeatable Read"). On top of snapshot isolation, SSI adds an algorithm for detecting serialization conflicts among reads and writes, and determining which transactions to abort.

Decisions based on an outdated premise

When we previously discussed write skew in snapshot isolation (see "Write Skew and Phantoms"), we observed a recurring pattern: a transaction reads some data from the database, examines the result of the query, and decides to take some action (write to the database) based on the result that it saw. However, under snapshot isolation, the result from the original query may no longer be up-to-date by the time the transaction commits, because the data may have been modified in the meantime.

Put another way, the transaction is taking an action based on a *premise* (a fact that was true at the beginning of the transaction, e.g., "There are currently two doctors on call"). Later, when the transaction wants to commit, the original data may have changed—the premise may no longer be true.

When the application makes a query (e.g., "How many doctors are currently on call?"), the database doesn't know how the application logic uses the result of that query. To be safe, the database needs to assume that any change in the query result (the premise) means that writes in that transaction may be invalid. In other words, there may be a causal dependency between the queries and the writes in the transaction. In order to provide serializable isolation, the database must detect situations in which a transaction may have acted on an outdated premise and abort the transaction in that case.

How does the database know if a query result might have changed? There are two cases to consider:

- Detecting reads of a stale MVCC object version (uncommitted write occurred before the read)
- Detecting writes that affect prior reads (the write occurs after the read)

Detecting stale MVCC reads

Recall that snapshot isolation is usually implemented by multi-version concurrency control (MVCC; see "Multi-version concurrency control (MVCC)"). When a transaction reads from a consistent snapshot in an MVCC database, it ignores writes that were made by any other transactions that hadn't yet committed at the time when the snapshot was taken.

In Figure 8-10, transaction 43 sees Aaliyah as having `on_call = true`, because transaction 42 (which modified Aaliyah's on-call status) is uncommitted. However, by the time transaction 43 wants to commit, transaction 42 has already committed. This means that the write that was ignored when reading from the consistent snapshot has now taken effect, and transaction 43's premise is no longer true. Things get even more complicated when a writer inserts data that didn't exist before (see "Phantoms causing write skew"). We'll discuss detecting phantom writes for SSI in "Detecting writes that affect prior reads".

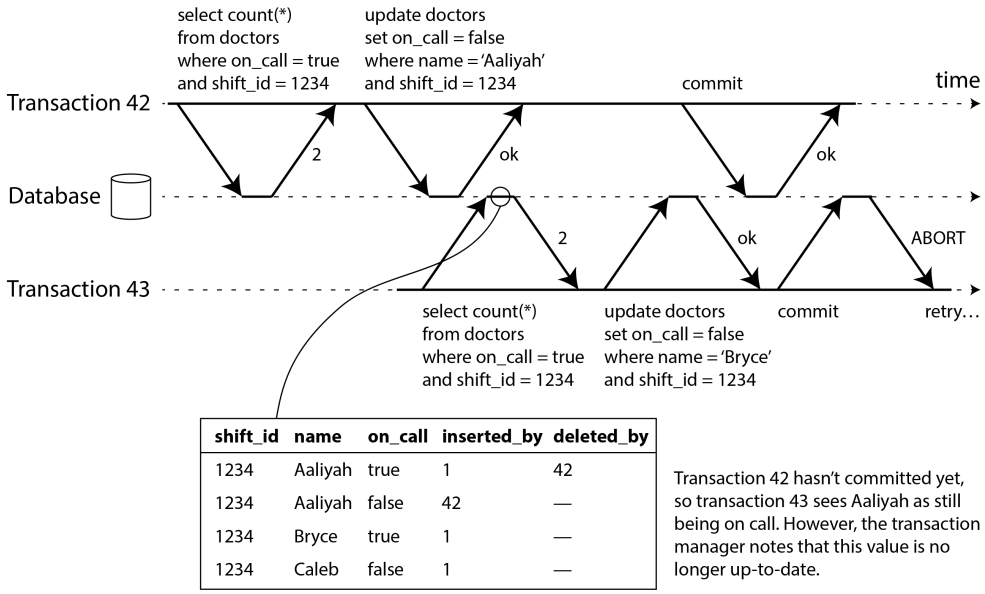

Figure 8-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

In order to prevent this anomaly, the database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules. When the transaction wants to commit, the database checks whether any of the ignored writes have now been committed. If so, the transaction must be aborted.

Why wait until committing? Why not abort transaction 43 immediately when the stale read is detected? Well, if transaction 43 was a read-only transaction, it wouldn't need to be aborted, because there is no risk of write skew. At the time when transaction 43 makes its read, the database doesn't yet know whether that transaction is going to later perform a write. Moreover, transaction 42 may yet abort or may still be uncommitted at the time when transaction 43 is committed, and so the read may turn out not to have been stale after all. By avoiding unnecessary aborts, SSI preserves snapshot isolation's support for long-running reads from a consistent snapshot.

Detecting writes that affect prior reads

The second case to consider is when another transaction modifies data after it has been read. This case is illustrated in Figure 8-11.

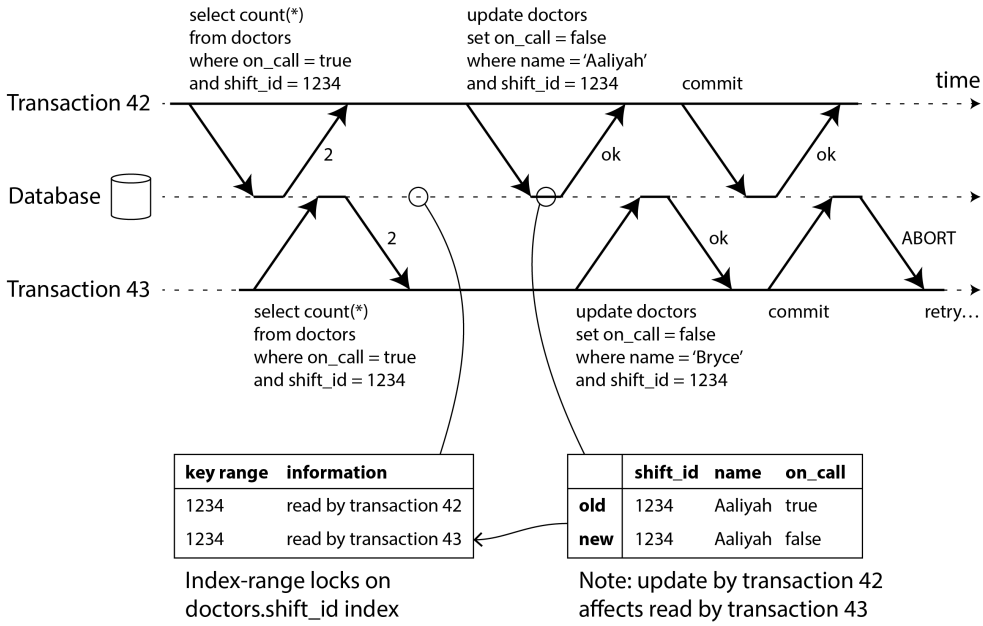

Figure 8-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

In the context of two-phase locking we discussed index-range locks (see "Index-range locks"), which allow the database to lock access to all rows matching some search query, such as `WHERE shift_id = 1234`. We can use a similar technique here, except that SSI locks don't block other transactions.

In Figure 8-11, transactions 42 and 43 both search for on-call doctors during shift `1234`. If there is an index on `shift_id`, the database can use the index entry 1234 to record the fact that transactions 42 and 43 read this data. (If there is no index, this information can be tracked at the table level.) This information only needs to be kept for a while: after a transaction has finished (committed or aborted), and all concurrent transactions have finished, the database can forget what data it read.

When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data. This process is similar to acquiring a write lock on the affected key range, but rather than blocking until the readers have committed, the lock acts as a tripwire: it simply notifies the transactions that the data they read may no longer be up to date.

In Figure 8-11, transaction 43 notifies transaction 42 that its prior read is outdated, and vice versa. Transaction 42 is first to commit, and it is successful: although transaction 43's write affected 42, 43 hasn't yet committed, so the write has not yet taken effect. However, when transaction 43 wants to commit, the conflicting write from 42 has already been committed, so 43 must abort.

Performance of serializable snapshot isolation

As always, many engineering details affect how well an algorithm works in practice. For example, one trade-off is the granularity at which transactions' reads and writes are tracked. If the database keeps track of each transaction's activity in great detail, it can be precise about which transactions need to abort, but the bookkeeping overhead can become significant. Less detailed tracking is faster, but may lead to more transactions being aborted than strictly necessary.

In some cases, it's okay for a transaction to read information that was overwritten by another transaction: depending on what else happened, it's sometimes possible to prove that the result of the execution is nevertheless serializable. PostgreSQL uses this theory to reduce the number of unnecessary aborts [14, 56].

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. Like under snapshot isolation, writers don't block readers, and vice versa. This design principle makes query latency much more predictable and less variable. In particular, read-only queries can run on a consistent snapshot without requiring any locks, which is very appealing for read-heavy workloads.

Compared to serial execution, serializable snapshot isolation is not limited to the throughput of a single CPU core: for example, FoundationDB distributes the detection of serialization conflicts across multiple machines, allowing it to scale to very high throughput. Even though data may be sharded across multiple machines, transactions can read and write data in multiple shards while ensuring serializable isolation.

Compared to non-serializable snapshot isolation, the need to check for serializability violations introduces some performance overheads. How significant these overheads are is a matter of debate: some believe that

serializability checking is not worth it [72], while others believe that the performance of serializability is now so good that there is no need to use the weaker snapshot isolation any more [69].

The rate of aborts significantly affects the overall performance of SSI. For example, a transaction that reads and writes data over a long period of time is likely to run into conflicts and abort, so SSI requires that read-write transactions be fairly short (long-running read-only transactions are okay). However, SSI is less sensitive to slow transactions than two-phase locking or serial execution.

Distributed Transactions

In a single-node transaction, you have a single machine that is in charge of executing the transaction logic, such as the concurrency control algorithms for transaction isolation. If your database uses single-leader replication, the transaction execution happens only on the leader, and the followers simply apply the log of writes that were committed by transactions on the leader.

However, what if multiple nodes are involved in a transaction? For example, perhaps you have a transaction that needs to touch multiple shards of a sharded database, or a global secondary index (in which the index entry may be on a different node from the primary data; see "Sharding and Secondary Indexes"). This is called a *distributed transaction*.

For concurrency control in distributed transactions, the algorithms are broadly similar to single-node concurrency control. We discussed serial execution on sharded databases previously; 2PL works in a distributed setting; and for SSI there are distributed serializability checkers [8]. We won't go into any more detail on these.

Achieving atomicity in a distributed transaction is a whole new challenge, though, and that's what the rest of this chapter will focus on.

For single-node transactions, atomicity is commonly implemented by the storage engine. When the client asks the database node to commit the transaction, the database makes the transaction's writes durable (typically in a write-ahead log; see "Making B-trees reliable") and then appends a commit record to the log on disk. If the database crashes in the middle of this process,

the transaction is recovered from the log when the node restarts: if the commit record was successfully written to disk before the crash, the transaction is considered committed; if not, any writes from that transaction are rolled back.

Thus, on a single node, transaction commitment crucially depends on the *order* in which data is durably written to disk: first the data, then the commit record [22]. The key deciding moment for whether the transaction commits or aborts is the moment at which the disk finishes writing the commit record: before that moment, it is still possible to abort (due to a crash), but after that moment, the transaction is committed (even if the database crashes). Thus, it is a single device (the controller of one particular disk drive, attached to one particular node) that makes the commit atomic.

In a distributed transaction, determining whether a transaction has committed or not is not so straightforward. For example, when a transaction wants to commit, it is not sufficient to simply send a commit request to all of the nodes and independently commit the transaction on each one. It could easily happen that the commit succeeds on some nodes and fails on other nodes, as shown in Figure 8-12:

- Some nodes may detect a constraint violation or conflict, making an abort necessary, while other nodes are successfully able to commit.
- Some of the commit requests might be lost in the network, eventually aborting due to a timeout, while other commit requests get through.
- Some nodes may crash before the commit record is fully written and roll back on recovery, while others successfully commit.

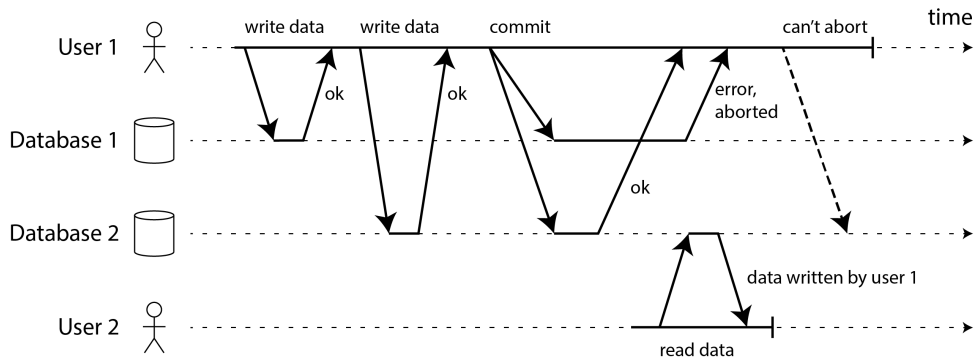

Figure 8-12. When a transaction involves multiple database nodes, it may commit on some and fail on others.

If some nodes commit the transaction but others abort it, the nodes become inconsistent with each other. And once a transaction has been committed on one node, it cannot be retracted again if it later turns out that it was aborted on

another node. This is because once data has been committed, it becomes visible to other transactions under *read committed* or stronger isolation. For example, in Figure 8-12, by the time user 1 notices that its commit failed on database 1, user 2 has already read the data from the same transaction on database 2. If user 1's transaction was later aborted, user 2's transaction would have to be reverted as well, since it was based on data that was retroactively declared not to have existed.

A better approach is to ensure that the nodes involved in a transaction either all commit or all abort, and to prevent a mixture of the two. Ensuring this is known as the *atomic commitment* problem.

Two-Phase Commit (2PC)

Two-phase commit is an algorithm for achieving atomic transaction commit across multiple nodes. It is a classic algorithm in distributed databases [13, 73, 74]. 2PC is used internally in some databases and also made available to applications in the form of *XA transactions* [75] (which are supported by the Java Transaction API, for example) or via WS-AtomicTransaction for SOAP web services [76, 77].

The basic flow of 2PC is illustrated in Figure 8-13. Instead of a single commit request, as with a single-node transaction, the commit/abort process in 2PC is split into two phases (hence the name).

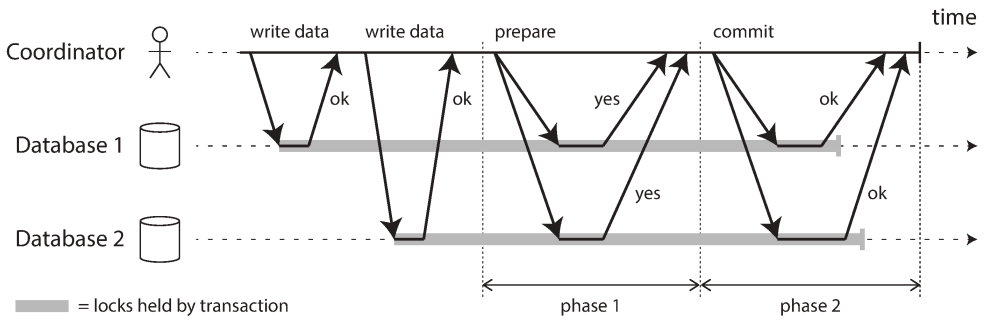

Figure 8-13. A successful execution of two-phase commit (2PC).

2PC uses a new component that does not normally appear in single-node transactions: a *coordinator* (also known as *transaction manager*). The coordinator is often implemented as a library within the same application process that is requesting the transaction (e.g., embedded in a Java EE container), but it can also be a separate process or service. Examples of such coordinators include Narayana, JOTM, BTM, or MSDTC.

When 2PC is used, a distributed transaction begins with the application reading and writing data on multiple database nodes, as normal. We call these database nodes *participants* in the transaction. When the application is ready to commit, the coordinator begins phase 1: it sends a *prepare* request to each of the nodes, asking them whether they are able to commit. The coordinator then tracks the responses from the participants:

- If all participants reply "yes," indicating they are ready to commit, then the coordinator sends out a *commit* request in phase 2, and the commit actually takes place.
- If any of the participants replies "no," the coordinator sends an *abort* request to all nodes in phase 2.

This process is somewhat like the traditional marriage ceremony in Western cultures: the minister asks the bride and groom individually whether each wants to marry the other, and typically receives the answer "I do" from both. After receiving both acknowledgments, the minister pronounces the couple husband and wife: the transaction is committed, and the happy fact is broadcast to all attendees. If either bride or groom does not say "yes," the ceremony is aborted [78].

A system of promises

From this short description it might not be clear why two-phase commit ensures atomicity, while one-phase commit across several nodes does not. Surely the prepare and commit requests can just as easily be lost in the two-phase case. What makes 2PC different?

To understand why it works, we have to break down the process in a bit more detail:

1. When the application wants to begin a distributed transaction, it requests a transaction ID from the coordinator. This transaction ID is globally unique.
2. The application begins a single-node transaction on each of the participants, and attaches the globally unique transaction ID to the single-node transaction. All reads and writes are done in one of these single-node transactions. If anything goes wrong at this stage (for example, a node crashes or a request times out), the coordinator or any of the participants can abort.

3. When the application is ready to commit, the coordinator sends a prepare request to all participants, tagged with the global transaction ID. If any of these requests fails or times out, the coordinator sends an abort request for that transaction ID to all participants.
4. When a participant receives the prepare request, it makes sure that it can definitely commit the transaction under all circumstances. This includes writing all transaction data to disk (a crash, a power failure, or running out of disk space is not an acceptable excuse for refusing to commit later), and checking for any conflicts or constraint violations. By replying "yes" to the coordinator, the node promises to commit the transaction without error if requested. In other words, the participant surrenders the right to abort the transaction, but without actually committing it.
5. When the coordinator has received responses to all prepare requests, it makes a definitive decision on whether to commit or abort the transaction (committing only if all participants voted "yes"). The coordinator must write that decision to its transaction log on disk so that it knows which way it decided in case it subsequently crashes. This is called the *commit point*.
6. Once the coordinator's decision has been written to disk, the commit or abort request is sent to all participants. If this request fails or times out, the coordinator must retry forever until it succeeds. There is no more going back: if the decision was to commit, that decision must be enforced, no matter how many retries it takes. If a participant has crashed in the meantime, the transaction will be committed when it recovers—since the participant voted "yes," it cannot refuse to commit when it recovers.

Thus, the protocol contains two crucial "points of no return": when a participant votes "yes," it promises that it will definitely be able to commit later (although the coordinator may still choose to abort); and once the coordinator decides, that decision is irrevocable. Those promises ensure the atomicity of 2PC. (Single-node atomic commit lumps these two events into one: writing the commit record to the transaction log.)

Returning to the marriage analogy, before saying "I do," you and your bride/groom have the freedom to abort the transaction by saying "No way!" (or something to that effect). However, after saying "I do," you cannot retract that statement. If you faint after saying "I do" and you don't hear the minister speak the words "You are now husband and wife," that doesn't change the fact that the transaction was committed. When you recover consciousness later, you can find out whether you are married or not by querying the minister for

the status of your global transaction ID, or you can wait for the minister's next retry of the commit request (since the retries will have continued throughout your period of unconsciousness).

Coordinator failure

We have discussed what happens if one of the participants or the network fails during 2PC: if any of the prepare requests fails or times out, the coordinator aborts the transaction; if any of the commit or abort requests fails, the coordinator retries them indefinitely. However, it is less clear what happens if the coordinator crashes.

If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction. But once the participant has received a prepare request and voted "yes," it can no longer abort unilaterally—it must wait to hear back from the coordinator whether the transaction was committed or aborted. If the coordinator crashes or the network fails at this point, the participant can do nothing but wait. A participant's transaction in this state is called *in doubt* or *uncertain*.

The situation is illustrated in Figure 8-14. In this particular example, the coordinator actually decided to commit, and database 2 received the commit request. However, the coordinator crashed before it could send the commit request to database 1, and so database 1 does not know whether to commit or abort. Even a timeout does not help here: if database 1 unilaterally aborts after a timeout, it will end up inconsistent with database 2, which has committed. Similarly, it is not safe to unilaterally commit, because another participant may have aborted.

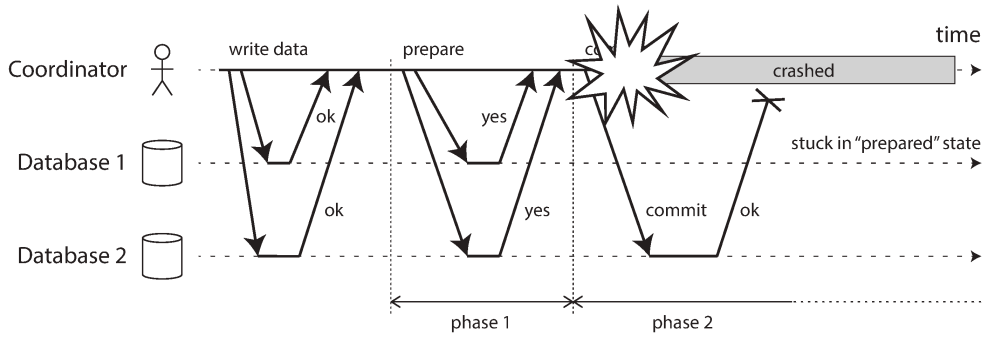

Figure 8-14. The coordinator crashes after participants vote "yes." Database 1 does not know whether to commit or abort.

Without hearing from the coordinator, the participant has no way of knowing whether to commit or abort. In principle, the participants could communicate

among themselves to find out how each participant voted and come to some agreement, but that is not part of the 2PC protocol.

The only way 2PC can complete is by waiting for the coordinator to recover. This is why the coordinator must write its commit or abort decision to a transaction log on disk before sending commit or abort requests to participants: when the coordinator recovers, it determines the status of all in-doubt transactions by reading its transaction log. Any transactions that don't have a commit record in the coordinator's log are aborted. Thus, the commit point of 2PC comes down to a regular single-node atomic commit on the coordinator.

Futhermore, if the coordinator's disk fails and its log is lost, there is no way for the system to automatically recover—the only option is for an administrator to manually commit or abort the in-doubt transactions. If only the most recent part of the transaction log is lost, the recovering coordinator may believe that already committed transactions have not yet been committed and try to abort them, violating atomicity.

Three-phase commit

Two-phase commit is called a *blocking* atomic commit protocol due to the fact that 2PC can become stuck waiting for the coordinator to recover. It is possible to make an atomic commit protocol *nonblocking*, so that it does not get stuck if a node fails. However, making this work in practice is not so straightforward.

As an alternative to 2PC, an algorithm called *three-phase commit* (3PC) has been proposed [13, 79]. However, 3PC assumes a network with bounded delay and nodes with bounded response times; in most practical systems with unbounded network delay and process pauses (see Chapter 9), it cannot guarantee atomicity.

A better solution in practice is to replace the single-node coordinator with a fault-tolerant consensus protocol. We will see how to do this in Chapter 10.

Distributed Transactions Across Different Systems

Distributed transactions and two-phase commit have a mixed reputation. On the one hand, they are seen as providing an important safety guarantee that

would be hard to achieve otherwise; on the other hand, they are criticized for causing operational problems, killing performance, and promising more than they can deliver [80, 81, 82, 83]. Many cloud services choose not to implement distributed transactions due to the operational problems they engender [84].

Some implementations of distributed transactions carry a heavy performance penalty. Much of the performance cost inherent in two-phase commit is due to the additional disk forcing (`fsync`) that is required for crash recovery, and the additional network round-trips.

However, rather than dismissing distributed transactions outright, we should examine them in some more detail, because there are important lessons to be learned from them. To begin, we should be precise about what we mean by "distributed transactions." Two quite different types of distributed transactions are often conflated:

Database-internal distributed transactions

Some distributed databases (i.e., databases that use replication and sharding in their standard configuration) support internal transactions among the nodes of that database. For example, YugabyteDB, TiDB, FoundationDB, Spanner, VoltDB, Cassandra, and MySQL Cluster's NDB storage engine have such internal transaction support. In this case, all the nodes participating in the transaction are running the same database software.

Heterogeneous distributed transactions

In a *heterogeneous* transaction, the participants are two or more different technologies: for example, two databases from different vendors, or even non-database systems such as message brokers. A distributed transaction across these systems must ensure atomic commit, even though the systems may be entirely different under the hood.

Database-internal transactions do not have to be compatible with any other system, so they can use any protocol and apply optimizations specific to that particular technology. For that reason, database-internal distributed transactions can often work quite well. On the other hand, transactions spanning heterogeneous technologies are a lot more challenging.

Exactly-once message processing

Heterogeneous distributed transactions allow diverse systems to be integrated in powerful ways. For example, a message from a message queue can be acknowledged as processed if and only if the database transaction for processing the message was successfully committed. This is implemented by atomically committing the message acknowledgment and the database writes in a single transaction. With distributed transaction support, this is possible, even if the message broker and the database are two unrelated technologies running on different machines.

If either the message delivery or the database transaction fails, both are aborted, and so the message broker may safely redeliver the message later. Thus, by atomically committing the message and the side effects of its processing, we can ensure that the message is *effectively* processed exactly once, even if it required a few retries before it succeeded. The abort discards any side effects of the partially completed transaction. This is known as *exactly-once semantics*.

Such a distributed transaction is only possible if all systems affected by the transaction are able to use the same atomic commit protocol, however. For example, say a side effect of processing a message is to send an email, and the email server does not support two-phase commit: it could happen that the email is sent two or more times if message processing fails and is retried. But if all side effects of processing a message are rolled back on transaction abort, then the processing step can safely be retried as if nothing had happened.

We will return to the topic of exactly-once semantics later in this chapter. Let's look first at the atomic commit protocol that allows such heterogeneous distributed transactions.

XA transactions

X/Open XA (short for *eXtended Architecture*) is a standard for implementing two-phase commit across heterogeneous technologies [75]. It was introduced in 1991 and has been widely implemented: XA is supported by many traditional relational databases (including PostgreSQL, MySQL, Db2, SQL Server, and Oracle) and message brokers (including ActiveMQ, HornetQ, MSMQ, and IBM MQ).

XA is not a network protocol—it is merely a C API for interfacing with a transaction coordinator. Bindings for this API exist in other languages; for example, in the world of Java EE applications, XA transactions are implemented using the Java Transaction API (JTA), which in turn is supported by many drivers for databases using Java Database Connectivity (JDBC) and drivers for message brokers using the Java Message Service (JMS) APIs.

XA assumes that your application uses a network driver or client library to communicate with the participant databases or messaging services. If the driver supports XA, that means it calls the XA API to find out whether an operation should be part of a distributed transaction—and if so, it sends the necessary information to the database server. The driver also exposes callbacks through which the coordinator can ask the participant to prepare, commit, or abort.

The transaction coordinator implements the XA API. The standard does not specify how it should be implemented, but in practice the coordinator is often simply a library that is loaded into the same process as the application issuing the transaction (not a separate service). It keeps track of the participants in a transaction, collects partipants' responses after asking them to prepare (via a callback into the driver), and uses a log on the local disk to keep track of the commit/abort decision for each transaction.

If the application process crashes, or the machine on which the application is running dies, the coordinator goes with it. Any participants with prepared but uncommitted transactions are then stuck in doubt. Since the coordinator's log is on the application server's local disk, that server must be restarted, and the coordinator library must read the log to recover the commit/abort outcome of each transaction. Only then can the coordinator use the database driver's XA callbacks to ask participants to commit or abort, as appropriate. The database server cannot contact the coordinator directly, since all communication must go via its client library.

Holding locks while in doubt

Why do we care so much about a transaction being stuck in doubt? Can't the rest of the system just get on with its work, and ignore the in-doubt transaction that will be cleaned up eventually?

The problem is with *locking*. As discussed in "Read Committed", database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty writes. In addition, if you want serializable isolation, a database using two-phase locking would also have to take a shared lock on any rows *read* by the transaction.

The database cannot release those locks until the transaction commits or aborts (illustrated as a shaded area in Figure 8-13). Therefore, when using two-phase commit, a transaction must hold onto the locks throughout the time it is in doubt. If the coordinator has crashed and takes 20 minutes to start up again, those locks will be held for 20 minutes. If the coordinator's log is entirely lost for some reason, those locks will be held forever—or at least until the situation is manually resolved by an administrator.

While those locks are held, no other transaction can modify those rows. Depending on the isolation level, other transactions may even be blocked from reading those rows. Thus, other transactions cannot simply continue with their business—if they want to access that same data, they will be blocked. This can cause large parts of your application to become unavailable until the in-doubt transaction is resolved.

Recovering from coordinator failure

In theory, if the coordinator crashes and is restarted, it should cleanly recover its state from the log and resolve any in-doubt transactions. However, in practice, *orphaned* in-doubt transactions do occur [85, 86]—that is, transactions for which the coordinator cannot decide the outcome for whatever reason (e.g., because the transaction log has been lost or corrupted due to a software bug). These transactions cannot be resolved automatically, so they sit forever in the database, holding locks and blocking other transactions.

Even rebooting your database servers will not fix this problem, since a correct implementation of 2PC must preserve the locks of an in-doubt transaction even across restarts (otherwise it would risk violating the atomicity guarantee). It's a sticky situation.

The only way out is for an administrator to manually decide whether to commit or roll back the transactions. The administrator must examine the participants of each in-doubt transaction, determine whether any participant has committed or aborted already, and then apply the same outcome to the

other participants. Resolving the problem potentially requires a lot of manual effort, and most likely needs to be done under high stress and time pressure during a serious production outage (otherwise, why would the coordinator be in such a bad state?).

Many XA implementations have an emergency escape hatch called *heuristic decisions*: allowing a participant to unilaterally decide to abort or commit an in-doubt transaction without a definitive decision from the coordinator [75]. To be clear, *heuristic* here is a euphemism for *probably breaking atomicity*, since the heuristic decision violates the system of promises in two-phase commit. Thus, heuristic decisions are intended only for getting out of catastrophic situations, and not for regular use.

Problems with XA transactions

A single-node coordinator is a single point of failure for the entire system, and making it part of the application server is also problematic because the coordinator's logs on its local disk become a crucial part of the durable system state—as important as the databases themselves.

In principle, the coordinator of an XA transaction could be highly available and replicated, just like we would expect of any other important database. Unfortunately, this still doesn't solve a fundamental problem with XA, which is that it provides no way for the coordinator and the participants of a transaction to communicate with each other directly. They can only communicate via the application code that invoked the transaction, and the database drivers through which it calls the participants.

Even if the coordinator were replicated, the application code would therefore be a single point of failure. Solving this problem would require totally redesigning how application code is run to make it replicated or restartable, which could perhaps look similar to durable execution (see "Durable Execution and Workflows"). However, there don't seem to be any tools that actually take this approach in practice.

Another problem is that since XA needs to be compatible with a wide range of data systems, it is necessarily a lowest common denominator. For example, it cannot detect deadlocks across different systems (since that would require a standardized protocol for systems to exchange information on the locks that each transaction is waiting for), and it does not work with SSI (see

"Serializable Snapshot Isolation (SSI)"), since that would require a protocol for identifying conflicts across different systems.

These problems are somewhat inherent in performing transactions across heterogeneous technologies. However, keeping several heterogeneous data systems consistent with each other is still a real and important problem, so we need to find a different solution to it. This can be done, as we will see in the next section and in Chapter 12.

Database-internal Distributed Transactions

As explained previously, there is a big difference between distributed transactions that span multiple heterogeneous storage technologies, and those that are internal to a system—i.e., where all the participating nodes are part of the same database running the same software. Such internal distributed transactions are a defining feature of "NewSQL" databases such as CockroachDB [5], TiDB [6], Spanner [7], FoundationDB [8], and YugabyteDB, for example. Some message brokers such as Kafka also support internal distributed transactions [87].

Many of these systems use 2-phase commit to ensure atomicity of transactions that write to multiple shards, and yet they don't suffer the same problems as XA transactions. The reason is that because their distributed transactions don't need to interface with any other technologies, they avoid the lowest-common-denominator trap—the designers of these systems are free to use better protocols that are more reliable and faster.

The biggest problems with XA can be fixed by:

- Replicating the coordinator, with automatic failover to another coordinator node if the primary one crashes;
- Allowing the coordinator and data shards to communicate directly without going via application code;
- Replicating the participating shards, so that the risk of having to abort a transaction because of a fault in one of the shards is reduced; and
- Coupling the atomic commitment protocol with a distributed concurrency control protocol that supports deadlock detection and consistent reads across shards.

Consensus algorithms are commonly used to replicate the coordinator and the database shards. We will see in Chapter 10 how atomic commitment for distributed transactions can be implemented using a consensus algorithm. These algorithms tolerate faults by automatically failing over from one node to another without any human intervention, and while continuing to guarantee strong consistency properties.

The isolation levels offered for distributed transactions depend on the system, but snapshot isolation and serializable snapshot isolation are both possible across shards. The details of how this works can be found in the papers referenced at the end of this chapter.

Exactly-once message processing revisited

We saw in "Exactly-once message processing" that an important use case for distributed transactions is to ensure that some operation takes effect exactly once, even if a crash occurs while it is being processed and the processing needs to be retried. If you can atomically commit a transaction across a message broker and a database, you can acknowledge the message to the broker if and only if it was successfully processed and the database writes resulting from the process were committed.

However, you don't actually need such distributed transactions to achieve exactly-once semantics. An alternative approach is as follows, which only requires transactions within the database:

1. Assume every message has a unique ID, and in the database you have a table of message IDs that have been processed. When you start processing a message from the broker, you begin a new transaction on the database, and check the message ID. If the same message ID is already present in the database, you know that it has already been processed, so you can acknowledge the message to the broker and drop it.
2. If the message ID is not already in the database, you add it to the table. You then process the message, which may result in additional writes to the database within the same transaction. When you finish processing the message, you commit the transaction on the database.
3. Once the database transaction is successfully committed, you can acknowledge the message to the broker.
4. Once the message has successfully been acknowledged to the broker, you know that it won't try processing the same message again, so you can

delete the message ID from the database (in a separate transaction).

If the message processor crashes before committing the database transaction, the transaction is aborted and the message broker will retry processing. If it crashes after committing but before acknowledging the message to the broker, it will also retry processing, but the retry will see the message ID in the database and drop it. If it crashes after acknowledging the message but before deleting the message ID from the database, you will have an old message ID lying around, which doesn't do any harm besides taking a little bit of storage space. If a retry happens before the database transaction is aborted (which could happen if communication between the message processor and the database is interrupted), a uniqueness constraint on the table of message IDs should prevent the same message ID from being inserted by two concurrent transactions.

Thus, achieving exactly-once processing only requires transactions within the database—atomicity across database and message broker is not necessary for this use case. Recording the message ID in the database makes the message processing *idempotent*, so that message processing can be safely retried without duplicating its side-effects. A similar approach is used in stream processing frameworks such as Kafka Streams to achieve exactly-once semantics, as we shall see in Chapter 12.

However, internal distributed transactions within the database are still useful for the scalability of patterns such as these: for example, they would allow the message IDs to be stored on one shard and the main data updated by the message processing to be stored on other shards, and to ensure atomicity of the transaction commit across those shards.

Summary

Transactions are an abstraction layer that allows an application to pretend that certain concurrency problems and certain kinds of hardware and software faults don't exist. A large class of errors is reduced down to a simple *transaction abort*, and the application just needs to try again.

In this chapter we saw many examples of problems that transactions help prevent. Not all applications are susceptible to all those problems: an application with very simple access patterns, such as reading and writing only

a single record, can probably manage without transactions. However, for more complex access patterns, transactions can hugely reduce the number of potential error cases you need to think about.

Without transactions, various error scenarios (processes crashing, network interruptions, power outages, disk full, unexpected concurrency, etc.) mean that data can become inconsistent in various ways. For example, denormalized data can easily go out of sync with the source data. Without transactions, it becomes very difficult to reason about the effects that complex interacting accesses can have on the database.

In this chapter, we went particularly deep into the topic of concurrency control. We discussed several widely used isolation levels, in particular *read committed*, *snapshot isolation* (sometimes called *repeatable read*), and *serializable*. We characterized those isolation levels by discussing various examples of race conditions, summarized in Table 8-1.

Table 8-1. Summary of anomalies that can occur at various isolation levels

Isolation level	Dirty reads	Read skew	Phantom reads	Lost updates	W
Read uncommitted	✗ Possible	✗ Possible	✗ Possible	✗ Possible	✗
Read committed	✓ Prevented	✗ Possible	✗ Possible	✗ Possible	✗
Snapshot isolation	✓ Prevented	✓ Prevented	✓ Prevented	? Depends	✗
Serializable	✓ Prevented	✓ Prevented	✓ Prevented	✓ Prevented	✓

Dirty reads

One client reads another client's writes before they have been committed. The read committed isolation level and stronger levels prevent dirty reads.

Dirty writes

One client overwrites data that another client has written, but not yet committed. Almost all transaction implementations prevent dirty writes.

Read skew

A client sees different parts of the database at different points in time. Some cases of read skew are also known as *nonrepeatable reads*. This issue is most commonly prevented with snapshot isolation, which allows a transaction to read from a consistent snapshot corresponding to one particular point in time. It is usually implemented with *multi-version concurrency control* (MVCC).

Lost updates

Two clients concurrently perform a read-modify-write cycle. One overwrites the other's write without incorporating its changes, so data is lost. Some implementations of snapshot isolation prevent this anomaly automatically, while others require a manual lock (`SELECT FOR UPDATE`).

Write skew

A transaction reads something, makes a decision based on the value it saw, and writes the decision to the database. However, by the time the write is made, the premise of the decision is no longer true. Only serializable isolation prevents this anomaly.

Phantom reads

A transaction reads objects that match some search condition. Another client makes a write that affects the results of that search. Snapshot isolation prevents straightforward phantom reads, but phantoms in the context of write skew require special treatment, such as index-range locks.

Weak isolation levels protect against some of those anomalies but leave you, the application developer, to handle others manually (e.g., using explicit locking). Only serializable isolation protects against all of these issues. We discussed three different approaches to implementing serializable transactions:

Literally executing transactions in a serial order

If you can make each transaction very fast to execute (typically by using stored procedures), and the transaction throughput is low enough to process on a single CPU core or can be sharded, this is a simple and effective option.

Two-phase locking

For decades this has been the standard way of implementing serializability, but many applications avoid using it because of its poor performance.

Serializable snapshot isolation (SSI)

A comparatively new algorithm that avoids most of the downsides of the previous approaches. It uses an optimistic approach, allowing transactions to proceed without blocking. When a transaction wants to commit, it is checked, and it is aborted if the execution was not serializable.

Finally, we examined how to achieve atomicity when a transaction is distributed across multiple nodes, using two-phase commit. If those nodes are all running the same database software, distributed transactions can work quite well, but across different storage technologies (using XA transactions), 2PC is problematic: it is very sensitive to faults in the coordinator and the application code driving the transaction, and it interacts poorly with concurrency control mechanisms. Fortunately, idempotence can ensure exactly-once semantics without requiring atomic commit across different storage technologies, and we will see more on this in later chapters.

The examples in this chapter used a relational data model. However, as discussed in "The need for multi-object transactions", transactions are a valuable database feature, no matter which data model is used.

FOOTNOTES

REFERENCES

[1] Steven J. Murdoch. What went wrong with Horizon: learning from the Post Office Trial. *benthamsgaze.org*, July 2021. Archived at perma.cc/CNM4-553F

[2] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A History and Evaluation of System R. *Communications of the ACM*, volume 24, issue 10, pages 632–646, October 1981. doi:10.1145/358769.358784

[3] Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. in *Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, edited by G. M. Nijssen, pages 364–394, Elsevier/North Holland Publishing, 1976. Also in *Readings in Database Systems*, 4th edition, edited by Joseph M. Hellerstein and Michael Stonebraker, MIT Press, 2005. ISBN: 978-0-262-69314-1

[4] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, volume 19, issue 11, pages 624–633, November 1976. doi:10.1145/360363.360369

[5] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. At *ACM SIGMOD International Conference on Management of Data* (SIGMOD), pages 1493–1509, June 2020. doi:10.1145/3318464.3386134

[6] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, volume 13, issue 12, pages 3072–3084. doi:10.14778/3415478.3415535

[7] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's Globally-Distributed Database. At *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.

[8] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A Distributed Unbundled Transactional Key Value Store. At *ACM International Conference on Management of Data* (SIGMOD), June 2021. doi:10.1145/3448016.3457559

[9] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, volume 15, issue 4, pages 287–317, December 1983.

doi:10.1145/289.291

[10] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, not CAP: Towards Highly Available Transactions. At *14th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2013.

[11] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. At *16th ACM Symposium on Operating Systems Principles* (SOSP), October 1997. doi:10.1145/268998.266662

[12] Tony Andrews. Enforcing Complex Constraints in Oracle. *tonyandrews.blogspot.co.uk*, October 2004. Archived at archive.org

[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 978-0-201-10715-9, available online at *microsoft.com*.

[14] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems*, volume 30, issue 2, pages 492–528, June 2005. doi:10.1145/1071610.1071615

[15] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDs Under Power Fault. At *11th USENIX Conference on File and Storage Technologies* (FAST), February 2013.

[16] Laurie Denness. SSDs: A Gift and a Curse. *laur.ie*, June 2015. Archived at perma.cc/6GLP-BX3T

[17] Adam Surak. When Solid State Drives Are Not That Solid. *blog.algolia.com*, June 2015. Archived at perma.cc/CBR9-QZEE

[18] Hewlett Packard Enterprise. Bulletin: (Revision) HPE SAS Solid State Drives - Critical Firmware Upgrade Required for Certain HPE SAS Solid State Drive Models to Prevent Drive Failure at 32,768 Hours of Operation. *support.hpe.com*, November 2019. Archived at perma.cc/CZR4-AQBS

[19] Craig Ringer et al. PostgreSQL's handling of fsync() errors is unsafe and risks data loss at least on XFS. Email thread on pgsql-hackers mailing list, *postgresql.org*, March 2018. Archived at perma.cc/5RKU-57FL

[20] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can Applications Recover from fsync Failures? At *USENIX Annual Technical Conference* (ATC), July 2020.

[21] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency: Rethinking the Fundamental Abstractions of the File System. *ACM Queue*, volume 13, issue 7, pages 20–28, July 2015. doi:10.1145/2800695.2801719

[22] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. At *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.

[23] Chris Siebenmann. Unix's File Durability Problem. *utcc.utoronto.ca*, April 2016. Archived at perma.cc/VSS8-5MC4

[24] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. At *15th USENIX Conference on File and Storage Technologies* (FAST), February 2017.

[25] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. At *6th USENIX Conference on File and Storage Technologies* (FAST), February 2008.

[26] Richard van der Hoff. How we discovered, and recovered from, Postgres corruption on the matrix.org homeserver. *matrix.org*, July 2025. Archived at perma.cc/CDF5-NRBK

[27] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. At *14th USENIX Conference on File and Storage Technologies* (FAST), February 2016.

[28] Don Allison. SSD Storage – Ignorance of Technology Is No Excuse. *blog.korelogic.com*, March 2015. Archived at perma.cc/9QN4-9SNJ

[29] Gordon Mah Ung. Debunked: Your SSD won't lose data if left unplugged after all. *pcworld.com*, May 2015. Archived at perma.cc/S46H-JUDU

[30] Martin Kleppmann. Hermitage: Testing the 'I' in ACID. *martin.kleppmann.com*, November 2014. Archived at perma.cc/KP2Y-AQGK

[31] Vlad Mihalcea. The race condition that led to Flexcoin bankruptcy. *vladmihalcea.com*, February 2025. Archived at perma.cc/RRK5-TFAU

[32] Todd Warszawski and Peter Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. At *ACM International Conference on Management of Data* (SIGMOD), May 2017. doi:10.1145/3035918.3064037

[33] Tristan D'Agosta. BTC Stolen from Poloniex. *bitcointalk.org*, March 2014. Archived at perma.cc/YHA6-4C5D

[34] bitcointhief2. How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More! *reddit.com*, February 2014. Archived at archive.org

[35] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the Detection of Snapshot Isolation Anomalies. At *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

[36] Michael Melanson. Transactions: The Limits of Isolation. *michaelmelanson.net*, November 2014. Archived at perma.cc/RG5R-KMYZ

[37] Edward Kim. How ACH works: A developer perspective — Part 1. *engineering.gusto.com*, April 2014. Archived at perma.cc/7B2H-PU94

[38] Hal Berenson, Philip A. Bernstein, Jim N. Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. At *ACM International Conference on Management of Data* (SIGMOD), May 1995. doi:10.1145/568271.223785

[39] Atul Adya. Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. PhD Thesis, Massachusetts Institute of Technology, March 1999. Archived at perma.cc/E97M-HW5Q

[40] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. At *40th International Conference on Very Large Data Bases* (VLDB), September 2014.

[41] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is Believing: A Client-Centric Specification of Database Isolation. At *ACM Symposium on Principles of Distributed Computing* (PODC), pages 73–82, July 2017. doi:10.1145/3087801.3087802

[42] Bruce Momjian. MVCC Unmasked. *momjian.us*, July 2014. Archived at perma.cc/KQ47-9GYB

[43] Peter Alvaro and Kyle Kingsbury. MySQL 8.0.34. *jepsen.io*, December 2023. Archived at perma.cc/HGE2-Z878

[44] Egor Rogov. PostgreSQL 14 Internals. *postgrespro.com*, April 2023. Archived at perma.cc/FRK2-D7WB

[45] Hironobu Suzuki. The Internals of PostgreSQL. *interdb.jp*, 2017.

[46] Rohan Reddy Alleti. Internals of MVCC in Postgres: Hidden costs of Updates vs Inserts. *medium.com*, March 2025. Archived at perma.cc/3ACX-DFXT

[47] Andy Pavlo and Bohan Zhang. The Part of PostgreSQL We Hate the Most. *cs.cmu.edu*, April 2023. Archived at perma.cc/XSP6-3JBN

[48] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, volume 10, issue 7, pages 781–792, March 2017. doi:10.14778/3067421.3067427

[49] Nikita Prokopov. Unofficial Guide to Datomic Internals. *tonsky.me*, May 2014.

[50] Daniil Svetlov. A Practical Guide to Taming Postgres Isolation Anomalies. *dansvetlov.me*, March 2025. Archived at perma.cc/L7LE-TDLS

[51] Nate Wiger. An Atomic Rant. *nateware.com*, February 2010. Archived at perma.cc/5ZYB-PE44

[52] James Coglan. Reading and writing, part 3: web applications. *blog.jcoglan.com*, October 2020. Archived at perma.cc/A7EK-PJVS

[53] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. At *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2737784

[54] Jaana Dogan. Things I Wished More Developers Knew About Databases. *rakyll.medium.com*, April 2020. Archived at perma.cc/6EFK-P2TD

[55] Michael J. Cahill, Uwe Röhm, and Alan Fekete. Serializable Isolation for Snapshot Databases. At *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376690

[56] Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation in PostgreSQL. At *38th International Conference on Very Large Databases* (VLDB), August 2012.

[57] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly

Connected Replicated Storage System. At *15th ACM Symposium on Operating Systems Principles* (SOSP), December 1995. doi:10.1145/224056.224070

[58] Hans-Jürgen Schönig. Constraints over multiple rows in PostgreSQL. *cybertec-postgresql.com*, June 2021. Archived at perma.cc/2TGH-XUPZ

[59] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). At *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

[60] John Hugg. H-Store/VoltDB Architecture vs. CEP Systems and Newer Streaming Architectures. At *Data @Scale Boston*, November 2014.

[61] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, volume 1, issue 2, pages 1496–1499, August 2008.

[62] Rich Hickey. The Architecture of Datomic. *infoq.com*, November 2012. Archived at perma.cc/5YWU-8XJK

[63] John Hugg. Debunking Myths About the VoltDB In-Memory Database. *dzone.com*, May 2014. Archived at perma.cc/2Z9N-HPKF

[64] Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck. At *15th Annual Conference on Innovative Data Systems Research* (CIDR), January 2025.

[65] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. Lotus: scalable multi-partition transactions on single-threaded partitioned databases. *Proceedings of the VLDB Endowment* (PVLDB), volume 15, issue 11, pages 2939–2952, July 2022. doi:10.14778/3551793.3551843

[66] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, volume 1, issue 2, pages 141–259, November 2007. doi:10.1561/1900000002

[67] Michael J. Cahill. Serializable Isolation for Snapshot Databases. PhD Thesis, University of Sydney, July 2009. Archived at perma.cc/727J-NTMP

[68] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-

Optimized OLTP Engine. At *ACM SIGMOD International Conference on Management of Data* (SIGMOD), pages 1243–1254, June 2013. doi:10.1145/2463676.2463710

[69] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. At *ACM SIGMOD International Conference on Management of Data* (SIGMOD), pages 677–689, May 2015. doi:10.1145/2723372.2749436

[70] D. Z. Badal. Correctness of Concurrency Control and Implications in Distributed Databases. At *3rd International IEEE Computer Software and Applications Conference* (COMPSAC), November 1979. doi:10.1109/CMPSAC.1979.762563

[71] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems* (TODS), volume 12, issue 4, pages 609–654, December 1987. doi:10.1145/32204.32220

[72] Marc Brooker. Snapshot Isolation vs Serializability. *brooker.co.za*, December 2024. Archived at perma.cc/5TRC-CR5G

[73] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, and B. W. Wade. Notes on Distributed Databases. IBM Research, Research Report RJ2571(33471), July 1979. Archived at perma.cc/EPZ3-MHDD

[74] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, volume 11, issue 4, pages 378–396, December 1986. doi:10.1145/7239.7266

[75] X/Open Company Ltd. Distributed Transaction Processing: The XA Specification. Technical Standard XO/CAE/91/300, December 1991. ISBN: 978-1-872-63024-3, archived at perma.cc/Z96H-29JB

[76] Ivan Silva Neto and Francisco Reverbel. Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction. At *7th IEEE/ACIS International Conference on Computer and Information Science* (ICIS), May 2008. doi:10.1109/ICIS.2008.75

[77] James E. Johnson, David E. Langworthy, Leslie Lamport, and Friedrich H. Vogt. Formal Specification of a Web Services Protocol. At *1st International Workshop on Web Services and Formal Methods* (WS-FM), February 2004. doi:10.1016/j.entcs.2004.02.022

[78] Jim Gray. The Transaction Concept: Virtues and Limitations. At *7th International Conference on Very Large Data Bases* (VLDB), September 1981.

[79] Dale Skeen. Nonblocking Commit Protocols. At *ACM International Conference on Management of Data* (SIGMOD), April 1981. doi:10.1145/582318.582339

[80] Gregor Hohpe. Your Coffee Shop Doesn't Use Two-Phase Commit. *IEEE Software*, volume 22, issue 2, pages 64–66, March 2005. doi:10.1109/MS.2005.52

[81] Pat Helland. Life Beyond Distributed Transactions: An Apostate's Opinion. At *3rd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2007.

[82] Jonathan Oliver. My Beef with MSDTC and Two-Phase Commits. *blog.jonathanoliver.com*, April 2011. Archived at perma.cc/K8HF-Z4EN

[83] Oren Eini (Ahende Rahien). The Fallacy of Distributed Transactions. *ayende.com*, July 2014. Archived at perma.cc/VB87-2JEF

[84] Clemens Vasters. Transactions in Windows Azure (with Service Bus) – An Email Discussion. *learn.microsoft.com*, July 2012. Archived at perma.cc/4EZ9-5SKW

[85] Ajmer Dhariwal. Orphaned MSDTC Transactions (-2 spids). *eraofdata.com*, December 2008. Archived at perma.cc/YG6F-U34C

[86] Paul Randal. Real World Story of DBCC PAGE Saving the Day. *sqlskills.com*, June 2013. Archived at perma.cc/2MJN-A5QH

[87] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. At *ACM International Conference on Management of Data* (SIGMOD), June 2021. doi:10.1145/3448016.3457556