# PART II
# Distributed Data

> *For a successful technology, reality must take precedence over public relations, for nature cannot be fooled.*
>
> —Richard Feynman, *Rogers Commission Report* (1986)

In Part I of this book, we discussed aspects of data systems that apply when data is stored on a single machine. Now, in Part II, we move up a level and ask: what happens if multiple machines are involved in storage and retrieval of data?

There are various reasons why you might want to distribute a database across multiple machines:

*Scalability*
: If your data volume, read load, or write load grows bigger than a single machine can handle, you can potentially spread the load across multiple machines.

*Fault tolerance/high availability*
: If your application needs to continue working even if one machine (or several machines, or the network, or an entire datacenter) goes down, you can use multiple machines to give you redundancy. When one fails, another one can take over.

*Latency*
: If you have users around the world, you might want to have servers at various locations worldwide so that each user can be served from a datacenter that is geographically close to them. That avoids the users having to wait for network packets to travel halfway around the world.

# Scaling to Higher Load

If all you need is to scale to higher load, the simplest approach is to buy a more powerful machine (sometimes called *vertical scaling* or *scaling up*). Many CPUs, many RAM chips, and many disks can be joined together under one operating system, and a fast interconnect allows any CPU to access any part of the memory or disk. In this kind of *shared-memory architecture*, all the components can be treated as a single machine [1].[i]

The problem with a shared-memory approach is that the cost grows faster than linearly: a machine with twice as many CPUs, twice as much RAM, and twice as much disk capacity as another typically costs significantly more than twice as much. And due to bottlenecks, a machine twice the size cannot necessarily handle twice the load.

A shared-memory architecture may offer limited fault tolerance—high-end machines have hot-swappable components (you can replace disks, memory modules, and even CPUs without shutting down the machines)—but it is definitely limited to a single geographic location.

Another approach is the *shared-disk architecture*, which uses several machines with independent CPUs and RAM, but stores data on an array of disks that is shared between the machines, which are connected via a fast network.[ii] This architecture is used for some data warehousing workloads, but contention and the overhead of locking limit the scalability of the shared-disk approach [2].

## Shared-Nothing Architectures

By contrast, *shared-nothing architectures* [3] (sometimes called *horizontal scaling* or *scaling out*) have gained a lot of popularity. In this approach, each machine or virtual machine running the database software is called a *node*. Each node uses its CPUs, RAM, and disks independently. Any coordination between nodes is done at the software level, using a conventional network.

No special hardware is required by a shared-nothing system, so you can use whatever machines have the best price/performance ratio. You can potentially distribute data across multiple geographic regions, and thus reduce latency for users and potentially be able to survive the loss of an entire datacenter. With cloud deployments of virtual

i. In a large machine, although any CPU can access any part of memory, some banks of memory are closer to one CPU than to others (this is called *nonuniform memory access*, or NUMA [1]). To make efficient use of this architecture, the processing needs to be broken down so that each CPU mostly accesses memory that is nearby—which means that partitioning is still required, even when ostensibly running on one machine.

ii. *Network Attached Storage* (NAS) or *Storage Area Network* (SAN).

machines, you don't need to be operating at Google scale: even for small companies, a multi-region distributed architecture is now feasible.

In this part of the book, we focus on shared-nothing architectures—not because they are necessarily the best choice for every use case, but rather because they require the most caution from you, the application developer. If your data is distributed across multiple nodes, you need to be aware of the constraints and trade-offs that occur in such a distributed system—the database cannot magically hide these from you.

While a distributed shared-nothing architecture has many advantages, it usually also incurs additional complexity for applications and sometimes limits the expressiveness of the data models you can use. In some cases, a simple single-threaded program can perform significantly better than a cluster with over 100 CPU cores [4]. On the other hand, shared-nothing systems can be very powerful. The next few chapters go into details on the issues that arise when data is distributed.

## Replication Versus Partitioning

There are two common ways data is distributed across multiple nodes:

*Replication*
: Keeping a copy of the same data on several different nodes, potentially in different locations. Replication provides redundancy: if some nodes are unavailable, the data can still be served from the remaining nodes. Replication can also help improve performance. We discuss replication in Chapter 5.

*Partitioning*
: Splitting a big database into smaller subsets called *partitions* so that different partitions can be assigned to different nodes (also known as *sharding*). We discuss partitioning in Chapter 6.

These are separate mechanisms, but they often go hand in hand, as illustrated in Figure II-1.

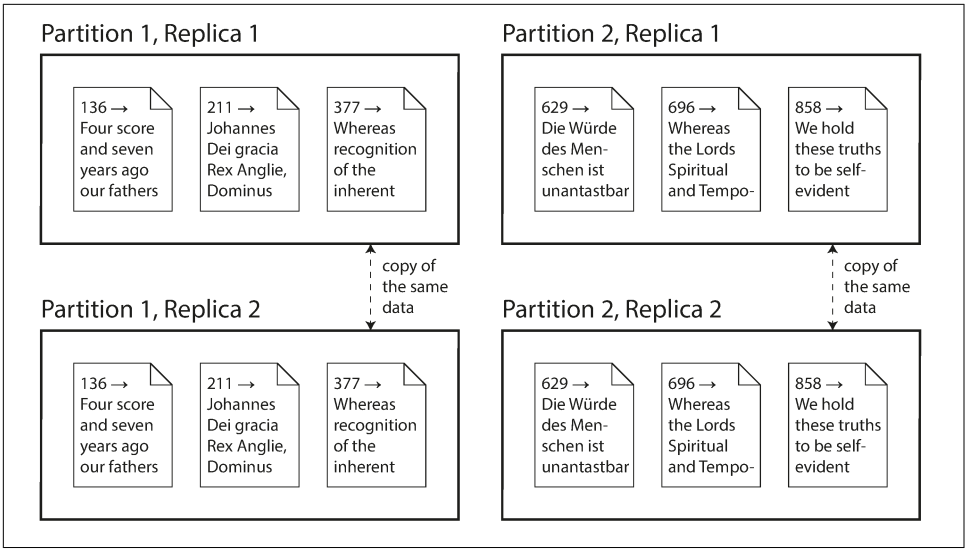



*Figure II-1. A database split into two partitions, with two replicas per partition.*

With an understanding of those concepts, we can discuss the difficult trade-offs that you need to make in a distributed system. We'll discuss *transactions* in Chapter 7, as that will help you understand all the many things that can go wrong in a data system, and what you can do about them. We'll conclude this part of the book by discussing the fundamental limitations of distributed systems in Chapters 8 and 9.

Later, in Part III of this book, we will discuss how you can take several (potentially distributed) datastores and integrate them into a larger system, satisfying the needs of a complex application. But first, let's talk about distributed data.

## References

[1] Ulrich Drepper: "What Every Programmer Should Know About Memory," *akkadia.org*, November 21, 2007.

[2] Ben Stopford: "Shared Nothing vs. Shared Disk Architectures: An Independent View," *benstopford.com*, November 24, 2009.

[3] Michael Stonebraker: "The Case for Shared Nothing," *IEEE Database Engineering Bulletin*, volume 9, number 1, pages 4–9, March 1986.

[4] Frank McSherry, Michael Isard, and Derek G. Murray: "Scalability! But at What COST?," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

# CHAPTER 5
# Replication

> *The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.*
>
> —Douglas Adams, *Mostly Harmless* (1992)

*Replication* means keeping a copy of the same data on multiple machines that are connected via a network. As discussed in the introduction to Part II, there are several reasons why you might want to replicate data:

- To keep data geographically close to your users (and thus reduce access latency)
- To allow the system to continue working even if some of its parts have failed (and thus increase availability)
- To scale out the number of machines that can serve read queries (and thus increase read throughput)

In this chapter we will assume that your dataset is so small that each machine can hold a copy of the entire dataset. In Chapter 6 we will relax that assumption and discuss *partitioning* (*sharding*) of datasets that are too big for a single machine. In later chapters we will discuss various kinds of faults that can occur in a replicated data system, and how to deal with them.

If the data that you're replicating does not change over time, then replication is easy: you just need to copy the data to every node once, and you're done. All of the difficulty in replication lies in handling *changes* to replicated data, and that's what this chapter is about. We will discuss three popular algorithms for replicating changes between nodes: *single-leader*, *multi-leader*, and *leaderless* replication. Almost all distributed databases use one of these three approaches. They all have various pros and cons, which we will examine in detail.

There are many trade-offs to consider with replication: for example, whether to use synchronous or asynchronous replication, and how to handle failed replicas. Those are often configuration options in databases, and although the details vary by database, the general principles are similar across many different implementations. We will discuss the consequences of such choices in this chapter.

Replication of databases is an old topic—the principles haven't changed much since they were studied in the 1970s [1], because the fundamental constraints of networks have remained the same. However, outside of research, many developers continued to assume for a long time that a database consisted of just one node. Mainstream use of distributed databases is more recent. Since many application developers are new to this area, there has been a lot of misunderstanding around issues such as *eventual consistency*. In "Problems with Replication Lag" on page 161 we will get more precise about eventual consistency and discuss things like the *read-your-writes* and *monotonic reads* guarantees.

## Leaders and Followers

Each node that stores a copy of the database is called a *replica*. With multiple replicas, a question inevitably arises: how do we ensure that all the data ends up on all the replicas?

Every write to the database needs to be processed by every replica; otherwise, the replicas would no longer contain the same data. The most common solution for this is called *leader-based replication* (also known as *active/passive* or *master–slave replication*) and is illustrated in Figure 5-1. It works as follows:

1. One of the replicas is designated the *leader* (also known as *master* or *primary*). When clients want to write to the database, they must send their requests to the leader, which first writes the new data to its local storage.
2. The other replicas are known as *followers* (*read replicas*, *slaves*, *secondaries*, or *hot standbys*).[i] Whenever the leader writes new data to its local storage, it also sends the data change to all of its followers as part of a *replication log* or *change stream*. Each follower takes the log from the leader and updates its local copy of the database accordingly, by applying all writes in the same order as they were processed on the leader.



i. Different people have different definitions for *hot*, *warm*, and *cold* standby servers. In PostgreSQL, for example, *hot standby* is used to refer to a replica that accepts reads from clients, whereas a *warm standby* processes changes from the leader but doesn't process any queries from clients. For purposes of this book, the difference isn't important.

3. When a client wants to read from the database, it can query either the leader or any of the followers. However, writes are only accepted on the leader (the followers are read-only from the client's point of view).

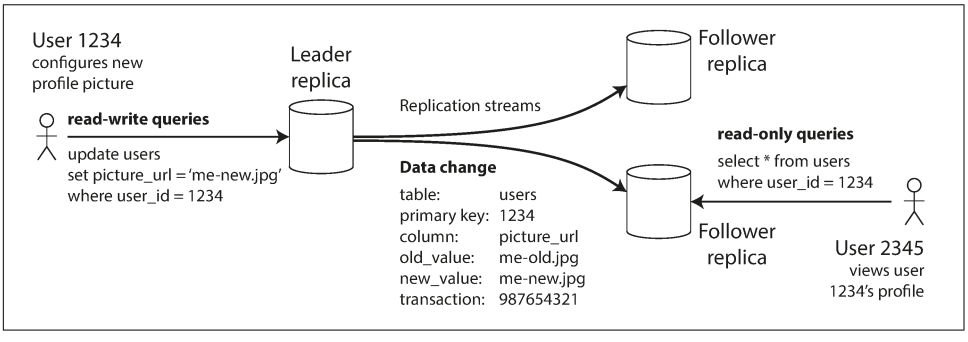



*Figure 5-1. Leader-based (master–slave) replication.*

This mode of replication is a built-in feature of many relational databases, such as PostgreSQL (since version 9.0), MySQL, Oracle Data Guard [2], and SQL Server's AlwaysOn Availability Groups [3]. It is also used in some nonrelational databases, including MongoDB, RethinkDB, and Espresso [4]. Finally, leader-based replication is not restricted to only databases: distributed message brokers such as Kafka [5] and RabbitMQ highly available queues [6] also use it. Some network filesystems and replicated block devices such as DRBD are similar.

## Synchronous Versus Asynchronous Replication

An important detail of a replicated system is whether the replication happens *synchronously* or *asynchronously*. (In relational databases, this is often a configurable option; other systems are often hardcoded to be either one or the other.)

Think about what happens in Figure 5-1, where the user of a website updates their profile image. At some point in time, the client sends the update request to the leader; shortly afterward, it is received by the leader. At some point, the leader forwards the data change to the followers. Eventually, the leader notifies the client that the update was successful.

Figure 5-2 shows the communication between various components of the system: the user's client, the leader, and two followers. Time flows from left to right. A request or response message is shown as a thick arrow.

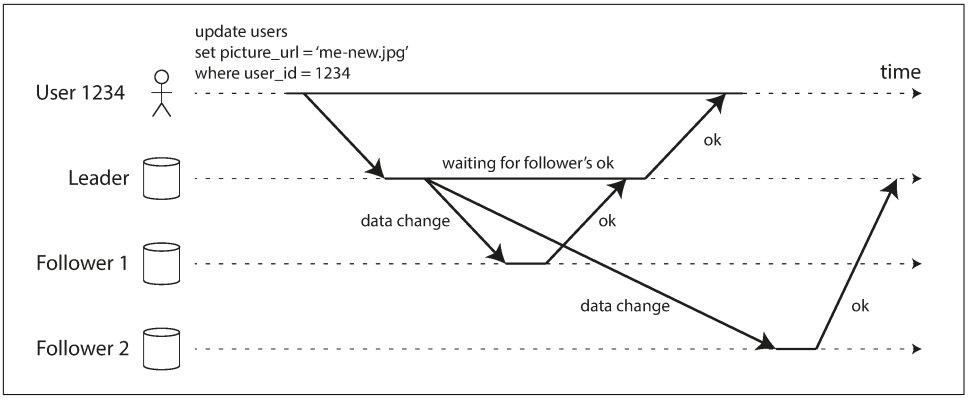



*Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.*

In the example of Figure 5-2, the replication to follower 1 is *synchronous*: the leader waits until follower 1 has confirmed that it received the write before reporting success to the user, and before making the write visible to other clients. The replication to follower 2 is *asynchronous*: the leader sends the message, but doesn't wait for a response from the follower.

The diagram shows that there is a substantial delay before follower 2 processes the message. Normally, replication is quite fast: most database systems apply changes to followers in less than a second. However, there is no guarantee of how long it might take. There are circumstances when followers might fall behind the leader by several minutes or more; for example, if a follower is recovering from a failure, if the system is operating near maximum capacity, or if there are network problems between the nodes.

The advantage of synchronous replication is that the follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader. If the leader suddenly fails, we can be sure that the data is still available on the follower. The disadvantage is that if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed. The leader must block all writes and wait until the synchronous replica is available again.

For that reason, it is impracticable for all followers to be synchronous: any one node outage would cause the whole system to grind to a halt. In practice, if you enable synchronous replication on a database, it usually means that *one* of the followers is synchronous, and the others are asynchronous. If the synchronous follower becomes unavailable or slow, one of the asynchronous followers is made synchronous. This guarantees that you have an up-to-date copy of the data on at least two nodes: the

leader and one synchronous follower. This configuration is sometimes also called *semi-synchronous* [7].

Often, leader-based replication is configured to be completely asynchronous. In this case, if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost. This means that a write is not guaranteed to be durable, even if it has been confirmed to the client. However, a fully asynchronous configuration has the advantage that the leader can continue processing writes, even if all of its followers have fallen behind.

Weakening durability may sound like a bad trade-off, but asynchronous replication is nevertheless widely used, especially if there are many followers or if they are geographically distributed. We will return to this issue in "Problems with Replication Lag" on page 161.

### Research on Replication

It can be a serious problem for asynchronously replicated systems to lose data if the leader fails, so researchers have continued investigating replication methods that do not lose data but still provide good performance and availability. For example, *chain replication* [8, 9] is a variant of synchronous replication that has been successfully implemented in a few systems such as Microsoft Azure Storage [10, 11].

There is a strong connection between consistency of replication and *consensus* (getting several nodes to agree on a value), and we will explore this area of theory in more detail in Chapter 9. In this chapter we will concentrate on the simpler forms of replication that are most commonly used in databases in practice.

## Setting Up New Followers

From time to time, you need to set up new followers—perhaps to increase the number of replicas, or to replace failed nodes. How do you ensure that the new follower has an accurate copy of the leader's data?

Simply copying data files from one node to another is typically not sufficient: clients are constantly writing to the database, and the data is always in flux, so a standard file copy would see different parts of the database at different points in time. The result might not make any sense.

You could make the files on disk consistent by locking the database (making it unavailable for writes), but that would go against our goal of high availability. Fortunately, setting up a follower can usually be done without downtime. Conceptually, the process looks like this:

1. Take a consistent snapshot of the leader's database at some point in time—if possible, without taking a lock on the entire database. Most databases have this feature, as it is also required for backups. In some cases, third-party tools are needed, such as *innobackupex* for MySQL [12].
2. Copy the snapshot to the new follower node.
3. The follower connects to the leader and requests all the data changes that have happened since the snapshot was taken. This requires that the snapshot is associated with an exact position in the leader's replication log. That position has various names: for example, PostgreSQL calls it the *log sequence number*, and MySQL calls it the *binlog coordinates*.
4. When the follower has processed the backlog of data changes since the snapshot, we say it has *caught up*. It can now continue to process data changes from the leader as they happen.

The practical steps of setting up a follower vary significantly by database. In some systems the process is fully automated, whereas in others it can be a somewhat arcane multi-step workflow that needs to be manually performed by an administrator.

## Handling Node Outages

Any node in the system can go down, perhaps unexpectedly due to a fault, but just as likely due to planned maintenance (for example, rebooting a machine to install a kernel security patch). Being able to reboot individual nodes without downtime is a big advantage for operations and maintenance. Thus, our goal is to keep the system as a whole running despite individual node failures, and to keep the impact of a node outage as small as possible.

How do you achieve high availability with leader-based replication?

### Follower failure: Catch-up recovery

On its local disk, each follower keeps a log of the data changes it has received from the leader. If a follower crashes and is restarted, or if the network between the leader and the follower is temporarily interrupted, the follower can recover quite easily: from its log, it knows the last transaction that was processed before the fault occurred. Thus, the follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected. When it has applied these changes, it has caught up to the leader and can continue receiving a stream of data changes as before.

### Leader failure: Failover

Handling a failure of the leader is trickier: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader. This process is called *failover*.

Failover can happen manually (an administrator is notified that the leader has failed and takes the necessary steps to make a new leader) or automatically. An automatic failover process usually consists of the following steps:

1. *Determining that the leader has failed.* There are many things that could potentially go wrong: crashes, power outages, network issues, and more. There is no foolproof way of detecting what has gone wrong, so most systems simply use a timeout: nodes frequently bounce messages back and forth between each other, and if a node doesn't respond for some period of time—say, 30 seconds—it is assumed to be dead. (If the leader is deliberately taken down for planned maintenance, this doesn't apply.)
2. *Choosing a new leader.* This could be done through an election process (where the leader is chosen by a majority of the remaining replicas), or a new leader could be appointed by a previously elected *controller node*. The best candidate for leadership is usually the replica with the most up-to-date data changes from the old leader (to minimize any data loss). Getting all the nodes to agree on a new leader is a consensus problem, discussed in detail in Chapter 9.
3. *Reconfiguring the system to use the new leader.* Clients now need to send their write requests to the new leader (we discuss this in "Request Routing" on page 214). If the old leader comes back, it might still believe that it is the leader, not realizing that the other replicas have forced it to step down. The system needs to ensure that the old leader becomes a follower and recognizes the new leader.

Failover is fraught with things that can go wrong:

- If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed. If the former leader rejoins the cluster after a new leader has been chosen, what should happen to those writes? The new leader may have received conflicting writes in the meantime. The most common solution is for the old leader's unreplicated writes to simply be discarded, which may violate clients' durability expectations.
- Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents. For example, in one incident at GitHub [13], an out-of-date MySQL follower was promoted to leader. The database used an autoincrementing counter to assign primary keys to new

rows, but because the new leader's counter lagged behind the old leader's, it reused some primary keys that were previously assigned by the old leader. These primary keys were also used in a Redis store, so the reuse of primary keys resulted in inconsistency between MySQL and Redis, which caused some private data to be disclosed to the wrong users.

- In certain fault scenarios (see Chapter 8), it could happen that two nodes both believe that they are the leader. This situation is called *split brain*, and it is dangerous: if both leaders accept writes, and there is no process for resolving conflicts (see "Multi-Leader Replication" on page 168), data is likely to be lost or corrupted. As a safety catch, some systems have a mechanism to shut down one node if two leaders are detected.[ii] However, if this mechanism is not carefully designed, you can end up with both nodes being shut down [14].
- What is the right timeout before the leader is declared dead? A longer timeout means a longer time to recovery in the case where the leader fails. However, if the timeout is too short, there could be unnecessary failovers. For example, a temporary load spike could cause a node's response time to increase above the timeout, or a network glitch could cause delayed packets. If the system is already struggling with high load or network problems, an unnecessary failover is likely to make the situation worse, not better.

There are no easy solutions to these problems. For this reason, some operations teams prefer to perform failovers manually, even if the software supports automatic failover.

These issues—node failures; unreliable networks; and trade-offs around replica consistency, durability, availability, and latency—are in fact fundamental problems in distributed systems. In Chapter 8 and Chapter 9 we will discuss them in greater depth.

## Implementation of Replication Logs

How does leader-based replication work under the hood? Several different replication methods are used in practice, so let's look at each one briefly.

### Statement-based replication

In the simplest case, the leader logs every write request (*statement*) that it executes and sends that statement log to its followers. For a relational database, this means that every `INSERT`, `UPDATE`, or `DELETE` statement is forwarded to followers, and each

ii. This approach is known as *fencing* or, more emphatically, *Shoot The Other Node In The Head* (STONITH). We will discuss fencing in more detail in "The leader and the lock" on page 301.

follower parses and executes that SQL statement as if it had been received from a client.

Although this may sound reasonable, there are various ways in which this approach to replication can break down:

- Any statement that calls a nondeterministic function, such as `NOW()` to get the current date and time or `RAND()` to get a random number, is likely to generate a different value on each replica.
- If statements use an autoincrementing column, or if they depend on the existing data in the database (e.g., `UPDATE … WHERE <some condition>`), they must be executed in exactly the same order on each replica, or else they may have a different effect. This can be limiting when there are multiple concurrently executing transactions.
- Statements that have side effects (e.g., triggers, stored procedures, user-defined functions) may result in different side effects occurring on each replica, unless the side effects are absolutely deterministic.

It is possible to work around those issues—for example, the leader can replace any nondeterministic function calls with a fixed return value when the statement is logged so that the followers all get the same value. However, because there are so many edge cases, other replication methods are now generally preferred.

Statement-based replication was used in MySQL before version 5.1. It is still sometimes used today, as it is quite compact, but by default MySQL now switches to row-based replication (discussed shortly) if there is any nondeterminism in a statement. VoltDB uses statement-based replication, and makes it safe by requiring transactions to be deterministic [15].

#### Write-ahead log (WAL) shipping

In Chapter 3 we discussed how storage engines represent data on disk, and we found that usually every write is appended to a log:

- In the case of a log-structured storage engine (see "SSTables and LSM-Trees" on page 76), this log is the main place for storage. Log segments are compacted and garbage-collected in the background.
- In the case of a B-tree (see "B-Trees" on page 79), which overwrites individual disk blocks, every modification is first written to a write-ahead log so that the index can be restored to a consistent state after a crash.

In either case, the log is an append-only sequence of bytes containing all writes to the database. We can use the exact same log to build a replica on another node: besides writing the log to disk, the leader also sends it across the network to its followers.

When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.

This method of replication is used in PostgreSQL and Oracle, among others [16]. The main disadvantage is that the log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk blocks. This makes replication closely coupled to the storage engine. If the database changes its storage format from one version to another, it is typically not possible to run different versions of the database software on the leader and the followers.

That may seem like a minor implementation detail, but it can have a big operational impact. If the replication protocol allows the follower to use a newer software version than the leader, you can perform a zero-downtime upgrade of the database software by first upgrading the followers and then performing a failover to make one of the upgraded nodes the new leader. If the replication protocol does not allow this version mismatch, as is often the case with WAL shipping, such upgrades require downtime.

#### Logical (row-based) log replication

An alternative is to use different log formats for replication and for the storage engine, which allows the replication log to be decoupled from the storage engine internals. This kind of replication log is called a *logical log*, to distinguish it from the storage engine's (*physical*) data representation.

A logical log for a relational database is usually a sequence of records describing writes to database tables at the granularity of a row:

- For an inserted row, the log contains the new values of all columns.
- For a deleted row, the log contains enough information to uniquely identify the row that was deleted. Typically this would be the primary key, but if there is no primary key on the table, the old values of all columns need to be logged.
- For an updated row, the log contains enough information to uniquely identify the updated row, and the new values of all columns (or at least the new values of all columns that changed).

A transaction that modifies several rows generates several such log records, followed by a record indicating that the transaction was committed. MySQL's binlog (when configured to use row-based replication) uses this approach [17].

Since a logical log is decoupled from the storage engine internals, it can more easily be kept backward compatible, allowing the leader and the follower to run different versions of the database software, or even different storage engines.

A logical log format is also easier for external applications to parse. This aspect is useful if you want to send the contents of a database to an external system, such as a data

warehouse for offline analysis, or for building custom indexes and caches [18]. This technique is called *change data capture*, and we will return to it in Chapter 11.

### Trigger-based replication

The replication approaches described so far are implemented by the database system, without involving any application code. In many cases, that's what you want—but there are some circumstances where more flexibility is needed. For example, if you want to only replicate a subset of the data, or want to replicate from one kind of database to another, or if you need conflict resolution logic (see "Handling Write Conflicts" on page 171), then you may need to move replication up to the application layer.

Some tools, such as Oracle GoldenGate [19], can make data changes available to an application by reading the database log. An alternative is to use features that are available in many relational databases: *triggers* and *stored procedures*.

A trigger lets you register custom application code that is automatically executed when a data change (write transaction) occurs in a database system. The trigger has the opportunity to log this change into a separate table, from which it can be read by an external process. That external process can then apply any necessary application logic and replicate the data change to another system. Databus for Oracle [20] and Bucardo for Postgres [21] work like this, for example.

Trigger-based replication typically has greater overheads than other replication methods, and is more prone to bugs and limitations than the database's built-in replication. However, it can nevertheless be useful due to its flexibility.

## Problems with Replication Lag

Being able to tolerate node failures is just one reason for wanting replication. As mentioned in the introduction to Part II, other reasons are scalability (processing more requests than a single machine can handle) and latency (placing replicas geographically closer to users).

Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica. For workloads that consist of mostly reads and only a small percentage of writes (a common pattern on the web), there is an attractive option: create many followers, and distribute the read requests across those followers. This removes load from the leader and allows read requests to be served by nearby replicas.

In this *read-scaling* architecture, you can increase the capacity for serving read-only requests simply by adding more followers. However, this approach only realistically works with asynchronous replication—if you tried to synchronously replicate to all followers, a single node failure or network outage would make the entire system

unavailable for writing. And the more nodes you have, the likelier it is that one will be down, so a fully synchronous configuration would be very unreliable.

Unfortunately, if an application reads from an *asynchronous* follower, it may see outdated information if the follower has fallen behind. This leads to apparent inconsistencies in the database: if you run the same query on the leader and a follower at the same time, you may get different results, because not all writes have been reflected in the follower. This inconsistency is just a temporary state—if you stop writing to the database and wait a while, the followers will eventually catch up and become consistent with the leader. For that reason, this effect is known as *eventual consistency* [22, 23].[iii]

The term "eventually" is deliberately vague: in general, there is no limit to how far a replica can fall behind. In normal operation, the delay between a write happening on the leader and being reflected on a follower—the *replication lag*—may be only a fraction of a second, and not noticeable in practice. However, if the system is operating near capacity or if there is a problem in the network, the lag can easily increase to several seconds or even minutes.

When the lag is so large, the inconsistencies it introduces are not just a theoretical issue but a real problem for applications. In this section we will highlight three examples of problems that are likely to occur when there is replication lag and outline some approaches to solving them.

## Reading Your Own Writes

Many applications let the user submit some data and then view what they have submitted. This might be a record in a customer database, or a comment on a discussion thread, or something else of that sort. When new data is submitted, it must be sent to the leader, but when the user views the data, it can be read from a follower. This is especially appropriate if data is frequently viewed but only occasionally written.

With asynchronous replication, there is a problem, illustrated in Figure 5-3: if the user views the data shortly after making a write, the new data may not yet have reached the replica. To the user, it looks as though the data they submitted was lost, so they will be understandably unhappy.



iii. The term *eventual consistency* was coined by Douglas Terry et al. [24], popularized by Werner Vogels [22], and became the battle cry of many NoSQL projects. However, not only NoSQL databases are eventually consistent: followers in an asynchronously replicated relational database have the same characteristics.

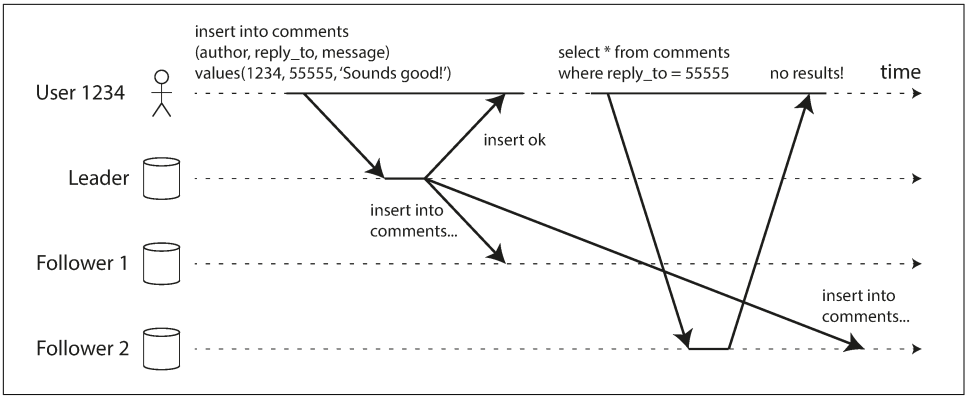



*Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.*

In this situation, we need *read-after-write consistency*, also known as *read-your-writes consistency* [24]. This is a guarantee that if the user reloads the page, they will always see any updates they submitted themselves. It makes no promises about other users: other users' updates may not be visible until some later time. However, it reassures the user that their own input has been saved correctly.

How can we implement read-after-write consistency in a system with leader-based replication? There are various possible techniques. To mention a few:

- When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower. This requires that you have some way of knowing whether something might have been modified, without actually querying it. For example, user profile information on a social network is normally only editable by the owner of the profile, not by anybody else. Thus, a simple rule is: always read the user's own profile from the leader, and any other users' profiles from a follower.
- If most things in the application are potentially editable by the user, that approach won't be effective, as most things would have to be read from the leader (negating the benefit of read scaling). In that case, other criteria may be used to decide whether to read from the leader. For example, you could track the time of the last update and, for one minute after the last update, make all reads from the leader. You could also monitor the replication lag on followers and prevent queries on any follower that is more than one minute behind the leader.
- The client can remember the timestamp of its most recent write—then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. If a replica is not sufficiently up to date, either the read can be handled by another replica or the query can wait until the replica has

caught up. The timestamp could be a *logical timestamp* (something that indicates ordering of writes, such as the log sequence number) or the actual system clock (in which case clock synchronization becomes critical; see "Unreliable Clocks" on page 287).

- If your replicas are distributed across multiple datacenters (for geographical proximity to users or for availability), there is additional complexity. Any request that needs to be served by the leader must be routed to the datacenter that contains the leader.

Another complication arises when the same user is accessing your service from multiple devices, for example a desktop web browser and a mobile app. In this case you may want to provide *cross-device* read-after-write consistency: if the user enters some information on one device and then views it on another device, they should see the information they just entered.

In this case, there are some additional issues to consider:

- Approaches that require remembering the timestamp of the user's last update become more difficult, because the code running on one device doesn't know what updates have happened on the other device. This metadata will need to be centralized.
- If your replicas are distributed across different datacenters, there is no guarantee that connections from different devices will be routed to the same datacenter. (For example, if the user's desktop computer uses the home broadband connection and their mobile device uses the cellular data network, the devices' network routes may be completely different.) If your approach requires reading from the leader, you may first need to route requests from all of a user's devices to the same datacenter.

## Monotonic Reads

Our second example of an anomaly that can occur when reading from asynchronous followers is that it's possible for a user to see things *moving backward in time*.

This can happen if a user makes several reads from different replicas. For example, Figure 5-4 shows user 2345 making the same query twice, first to a follower with little lag, then to a follower with greater lag. (This scenario is quite likely if the user refreshes a web page, and each request is routed to a random server.) The first query returns a comment that was recently added by user 1234, but the second query doesn't return anything because the lagging follower has not yet picked up that write. In effect, the second query observes the system state at an earlier point in time than the first query. This wouldn't be so bad if the first query hadn't returned anything, because user 2345 probably wouldn't know that user 1234 had recently added a com-

ment. However, it's very confusing for user 2345 if they first see user 1234's comment appear, and then see it disappear again.

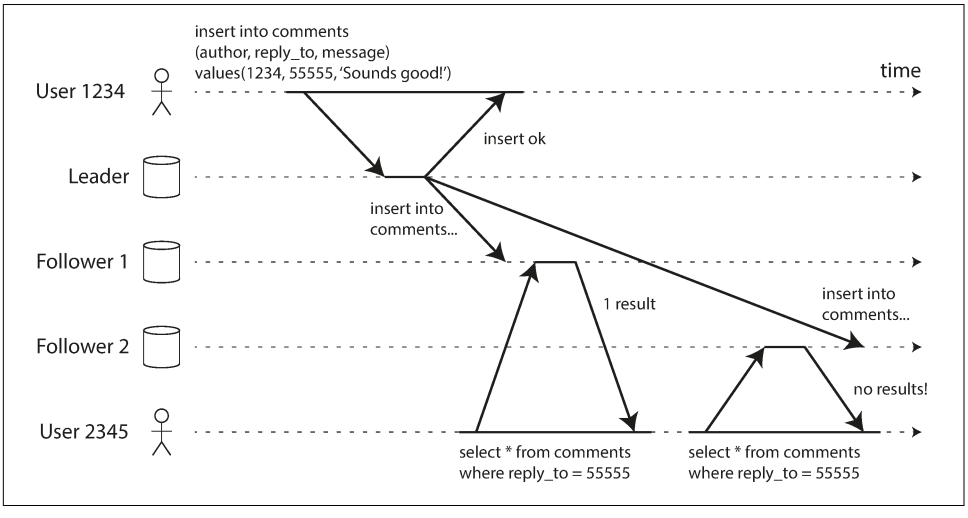



*Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.*

*Monotonic reads* [23] is a guarantee that this kind of anomaly does not happen. It's a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency. When you read data, you may see an old value; monotonic reads only means that if one user makes several reads in sequence, they will not see time go backward—i.e., they will not read older data after having previously read newer data.

One way of achieving monotonic reads is to make sure that each user always makes their reads from the same replica (different users can read from different replicas). For example, the replica can be chosen based on a hash of the user ID, rather than randomly. However, if that replica fails, the user's queries will need to be rerouted to another replica.

## Consistent Prefix Reads

Our third example of replication lag anomalies concerns violation of causality. Imagine the following short dialog between Mr. Poons and Mrs. Cake:

*Mr. Poons*
: How far into the future can you see, Mrs. Cake?

*Mrs. Cake*
: About ten seconds usually, Mr. Poons.

There is a causal dependency between those two sentences: Mrs. Cake heard Mr. Poons's question and answered it.

Now, imagine a third person is listening to this conversation through followers. The things said by Mrs. Cake go through a follower with little lag, but the things said by Mr. Poons have a longer replication lag (see Figure 5-5). This observer would hear the following:

*Mrs. Cake*
: About ten seconds usually, Mr. Poons.

*Mr. Poons*
: How far into the future can you see, Mrs. Cake?

To the observer it looks as though Mrs. Cake is answering the question before Mr. Poons has even asked it. Such psychic powers are impressive, but very confusing [25].

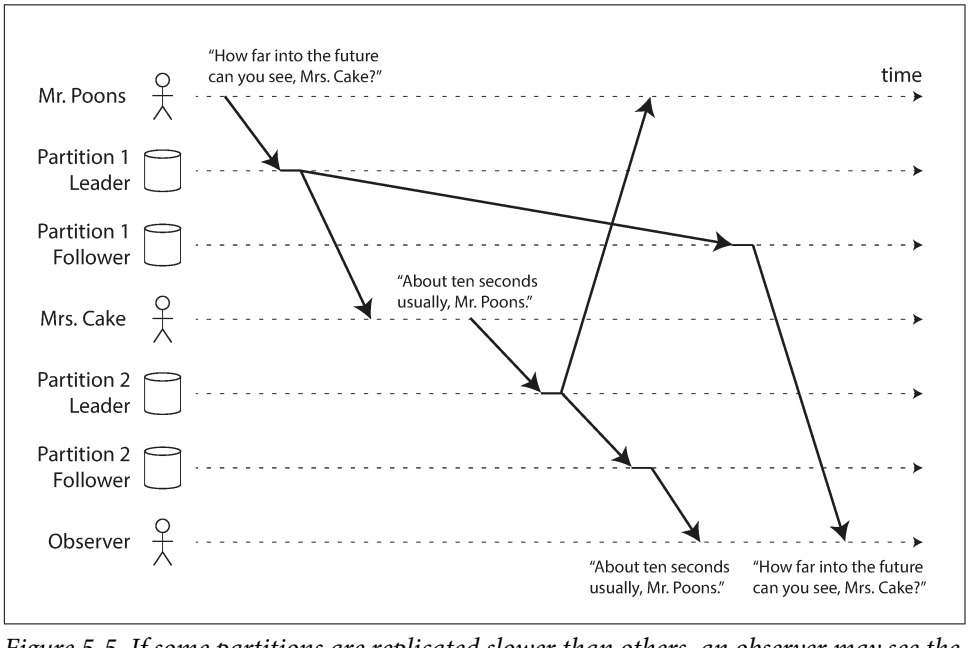



*Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.*

Preventing this kind of anomaly requires another type of guarantee: *consistent prefix reads* [23]. This guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order.

This is a particular problem in partitioned (sharded) databases, which we will discuss in Chapter 6. If the database always applies writes in the same order, reads always see a consistent prefix, so this anomaly cannot happen. However, in many distributed

databases, different partitions operate independently, so there is no global ordering of writes: when a user reads from the database, they may see some parts of the database in an older state and some in a newer state.

One solution is to make sure that any writes that are causally related to each other are written to the same partition—but in some applications that cannot be done efficiently. There are also algorithms that explicitly keep track of causal dependencies, a topic that we will return to in "The "happens-before" relationship and concurrency" on page 186.

## Solutions for Replication Lag

When working with an eventually consistent system, it is worth thinking about how the application behaves if the replication lag increases to several minutes or even hours. If the answer is "no problem," that's great. However, if the result is a bad experience for users, it's important to design the system to provide a stronger guarantee, such as read-after-write. Pretending that replication is synchronous when in fact it is asynchronous is a recipe for problems down the line.

As discussed earlier, there are ways in which an application can provide a stronger guarantee than the underlying database—for example, by performing certain kinds of reads on the leader. However, dealing with these issues in application code is complex and easy to get wrong.

It would be better if application developers didn't have to worry about subtle replication issues and could just trust their databases to "do the right thing." This is why *transactions* exist: they are a way for a database to provide stronger guarantees so that the application can be simpler.

Single-node transactions have existed for a long time. However, in the move to distributed (replicated and partitioned) databases, many systems have abandoned them, claiming that transactions are too expensive in terms of performance and availability, and asserting that eventual consistency is inevitable in a scalable system. There is some truth in that statement, but it is overly simplistic, and we will develop a more nuanced view over the course of the rest of this book. We will return to the topic of transactions in Chapters 7 and 9, and we will discuss some alternative mechanisms in Part III.

# Multi-Leader Replication

So far in this chapter we have only considered replication architectures using a single leader. Although that is a common approach, there are interesting alternatives.

Leader-based replication has one major downside: there is only one leader, and all writes must go through it.[iv] If you can't connect to the leader for any reason, for example due to a network interruption between you and the leader, you can't write to the database.

A natural extension of the leader-based replication model is to allow more than one node to accept writes. Replication still happens in the same way: each node that processes a write must forward that data change to all the other nodes. We call this a *multi-leader* configuration (also known as *master–master* or *active/active replication*). In this setup, each leader simultaneously acts as a follower to the other leaders.

## Use Cases for Multi-Leader Replication

It rarely makes sense to use a multi-leader setup within a single datacenter, because the benefits rarely outweigh the added complexity. However, there are some situations in which this configuration is reasonable.

### Multi-datacenter operation

Imagine you have a database with replicas in several different datacenters (perhaps so that you can tolerate failure of an entire datacenter, or perhaps in order to be closer to your users). With a normal leader-based replication setup, the leader has to be in *one* of the datacenters, and all writes must go through that datacenter.

In a multi-leader configuration, you can have a leader in *each* datacenter. Figure 5-6 shows what this architecture might look like. Within each datacenter, regular leader–follower replication is used; between datacenters, each datacenter's leader replicates its changes to the leaders in other datacenters.



iv. If the database is partitioned (see Chapter 6), each partition has one leader. Different partitions may have their leaders on different nodes, but each partition must nevertheless have one leader node.

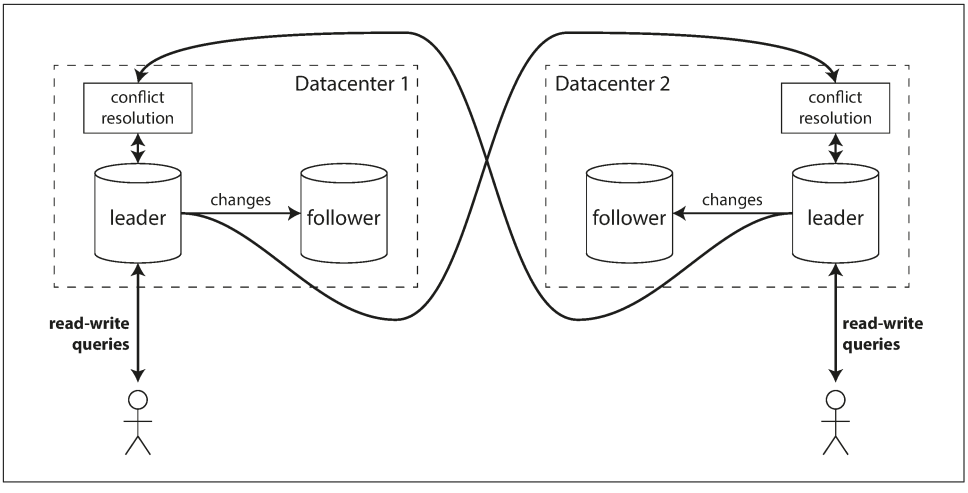



*Figure 5-6. Multi-leader replication across multiple datacenters.*

Let's compare how the single-leader and multi-leader configurations fare in a multi-datacenter deployment:

*Performance*
: In a single-leader configuration, every write must go over the internet to the datacenter with the leader. This can add significant latency to writes and might contravene the purpose of having multiple datacenters in the first place. In a multi-leader configuration, every write can be processed in the local datacenter and is replicated asynchronously to the other datacenters. Thus, the inter-datacenter network delay is hidden from users, which means the perceived performance may be better.

*Tolerance of datacenter outages*
: In a single-leader configuration, if the datacenter with the leader fails, failover can promote a follower in another datacenter to be leader. In a multi-leader configuration, each datacenter can continue operating independently of the others, and replication catches up when the failed datacenter comes back online.

*Tolerance of network problems*
: Traffic between datacenters usually goes over the public internet, which may be less reliable than the local network within a datacenter. A single-leader configuration is very sensitive to problems in this inter-datacenter link, because writes are made synchronously over this link. A multi-leader configuration with asynchronous replication can usually tolerate network problems better: a temporary network interruption does not prevent writes being processed.

Some databases support multi-leader configurations by default, but it is also often implemented with external tools, such as Tungsten Replicator for MySQL [26], BDR for PostgreSQL [27], and GoldenGate for Oracle [19].

Although multi-leader replication has advantages, it also has a big downside: the same data may be concurrently modified in two different datacenters, and those write conflicts must be resolved (indicated as "conflict resolution" in Figure 5-6). We will discuss this issue in "Handling Write Conflicts" on page 171.

As multi-leader replication is a somewhat retrofitted feature in many databases, there are often subtle configuration pitfalls and surprising interactions with other database features. For example, autoincrementing keys, triggers, and integrity constraints can be problematic. For this reason, multi-leader replication is often considered dangerous territory that should be avoided if possible [28].

### Clients with offline operation

Another situation in which multi-leader replication is appropriate is if you have an application that needs to continue to work while it is disconnected from the internet.

For example, consider the calendar apps on your mobile phone, your laptop, and other devices. You need to be able to see your meetings (make read requests) and enter new meetings (make write requests) at any time, regardless of whether your device currently has an internet connection. If you make any changes while you are offline, they need to be synced with a server and your other devices when the device is next online.

In this case, every device has a local database that acts as a leader (it accepts write requests), and there is an asynchronous multi-leader replication process (sync) between the replicas of your calendar on all of your devices. The replication lag may be hours or even days, depending on when you have internet access available.

From an architectural point of view, this setup is essentially the same as multi-leader replication between datacenters, taken to the extreme: each device is a "datacenter," and the network connection between them is extremely unreliable. As the rich history of broken calendar sync implementations demonstrates, multi-leader replication is a tricky thing to get right.

There are tools that aim to make this kind of multi-leader configuration easier. For example, CouchDB is designed for this mode of operation [29].

### Collaborative editing

*Real-time collaborative editing* applications allow several people to edit a document simultaneously. For example, Etherpad [30] and Google Docs [31] allow multiple people to concurrently edit a text document or spreadsheet (the algorithm is briefly discussed in "Automatic Conflict Resolution" on page 174).

We don't usually think of collaborative editing as a database replication problem, but it has a lot in common with the previously mentioned offline editing use case. When one user edits a document, the changes are instantly applied to their local replica (the state of the document in their web browser or client application) and asynchronously replicated to the server and any other users who are editing the same document.

If you want to guarantee that there will be no editing conflicts, the application must obtain a lock on the document before a user can edit it. If another user wants to edit the same document, they first have to wait until the first user has committed their changes and released the lock. This collaboration model is equivalent to single-leader replication with transactions on the leader.

However, for faster collaboration, you may want to make the unit of change very small (e.g., a single keystroke) and avoid locking. This approach allows multiple users to edit simultaneously, but it also brings all the challenges of multi-leader replication, including requiring conflict resolution [32].

## Handling Write Conflicts

The biggest problem with multi-leader replication is that write conflicts can occur, which means that conflict resolution is required.

For example, consider a wiki page that is simultaneously being edited by two users, as shown in Figure 5-7. User 1 changes the title of the page from A to B, and user 2 changes the title from A to C at the same time. Each user's change is successfully applied to their local leader. However, when the changes are asynchronously replicated, a conflict is detected [33]. This problem does not occur in a single-leader database.

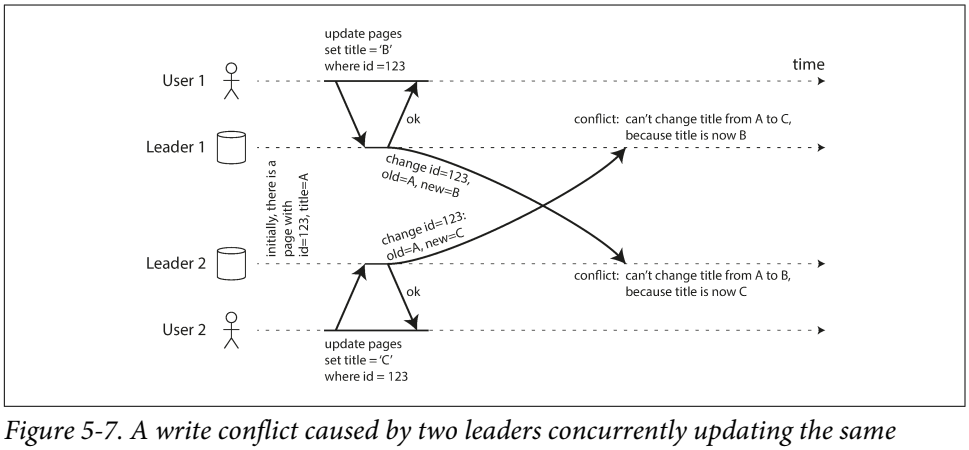



*Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.*

### Synchronous versus asynchronous conflict detection

In a single-leader database, the second writer will either block and wait for the first write to complete, or abort the second write transaction, forcing the user to retry the write. On the other hand, in a multi-leader setup, both writes are successful, and the conflict is only detected asynchronously at some later point in time. At that time, it may be too late to ask the user to resolve the conflict.

In principle, you could make the conflict detection synchronous—i.e., wait for the write to be replicated to all replicas before telling the user that the write was successful. However, by doing so, you would lose the main advantage of multi-leader replication: allowing each replica to accept writes independently. If you want synchronous conflict detection, you might as well just use single-leader replication.

### Conflict avoidance

The simplest strategy for dealing with conflicts is to avoid them: if the application can ensure that all writes for a particular record go through the same leader, then conflicts cannot occur. Since many implementations of multi-leader replication handle conflicts quite poorly, avoiding conflicts is a frequently recommended approach [34].

For example, in an application where a user can edit their own data, you can ensure that requests from a particular user are always routed to the same datacenter and use the leader in that datacenter for reading and writing. Different users may have different "home" datacenters (perhaps picked based on geographic proximity to the user), but from any one user's point of view the configuration is essentially single-leader.

However, sometimes you might want to change the designated leader for a record—perhaps because one datacenter has failed and you need to reroute traffic to another datacenter, or perhaps because a user has moved to a different location and is now closer to a different datacenter. In this situation, conflict avoidance breaks down, and you have to deal with the possibility of concurrent writes on different leaders.

### Converging toward a consistent state

A single-leader database applies writes in a sequential order: if there are several updates to the same field, the last write determines the final value of the field.

In a multi-leader configuration, there is no defined ordering of writes, so it's not clear what the final value should be. In Figure 5-7, at leader 1 the title is first updated to B and then to C; at leader 2 it is first updated to C and then to B. Neither order is "more correct" than the other.

If each replica simply applied writes in the order that it saw the writes, the database would end up in an inconsistent state: the final value would be C at leader 1 and B at leader 2. That is not acceptable—every replication system must ensure that the data is eventually the same in all replicas. Thus, the database must resolve the conflict in a

*convergent* way, which means that all replicas must arrive at the same final value when all changes have been replicated.

There are various ways of achieving convergent conflict resolution:

- Give each write a unique ID (e.g., a timestamp, a long random number, a UUID, or a hash of the key and value), pick the write with the highest ID as the *winner*, and throw away the other writes. If a timestamp is used, this technique is known as *last write wins* (LWW). Although this approach is popular, it is dangerously prone to data loss [35]. We will discuss LWW in more detail at the end of this chapter ("Detecting Concurrent Writes" on page 184).
- Give each replica a unique ID, and let writes that originated at a higher-numbered replica always take precedence over writes that originated at a lower-numbered replica. This approach also implies data loss.
- Somehow merge the values together—e.g., order them alphabetically and then concatenate them (in Figure 5-7, the merged title might be something like "B/C").
- Record the conflict in an explicit data structure that preserves all information, and write application code that resolves the conflict at some later time (perhaps by prompting the user).

### Custom conflict resolution logic

As the most appropriate way of resolving a conflict may depend on the application, most multi-leader replication tools let you write conflict resolution logic using application code. That code may be executed on write or on read:

*On write*
: As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler. For example, Bucardo allows you to write a snippet of Perl for this purpose. This handler typically cannot prompt a user—it runs in a background process and it must execute quickly.

*On read*
: When a conflict is detected, all the conflicting writes are stored. The next time the data is read, these multiple versions of the data are returned to the application. The application may prompt the user or automatically resolve the conflict, and write the result back to the database. CouchDB works this way, for example.

Note that conflict resolution usually applies at the level of an individual row or document, not for an entire transaction [36]. Thus, if you have a transaction that atomically makes several different writes (see Chapter 7), each write is still considered separately for the purposes of conflict resolution.

## Automatic Conflict Resolution

Conflict resolution rules can quickly become complicated, and custom code can be error-prone. Amazon is a frequently cited example of surprising effects due to a conflict resolution handler: for some time, the conflict resolution logic on the shopping cart would preserve items added to the cart, but not items removed from the cart. Thus, customers would sometimes see items reappearing in their carts even though they had previously been removed [37].

There has been some interesting research into automatically resolving conflicts caused by concurrent data modifications. A few lines of research are worth mentioning:

- *Conflict-free replicated datatypes* (CRDTs) [32, 38] are a family of data structures for sets, maps (dictionaries), ordered lists, counters, etc. that can be concurrently edited by multiple users, and which automatically resolve conflicts in sensible ways. Some CRDTs have been implemented in Riak 2.0 [39, 40].
- *Mergeable persistent data structures* [41] track history explicitly, similarly to the Git version control system, and use a three-way merge function (whereas CRDTs use two-way merges).
- *Operational transformation* [42] is the conflict resolution algorithm behind collaborative editing applications such as Etherpad [30] and Google Docs [31]. It was designed particularly for concurrent editing of an ordered list of items, such as the list of characters that constitute a text document.

Implementations of these algorithms in databases are still young, but it's likely that they will be integrated into more replicated data systems in the future. Automatic conflict resolution could make multi-leader data synchronization much simpler for applications to deal with.

### What is a conflict?

Some kinds of conflict are obvious. In the example in Figure 5-7, two writes concurrently modified the same field in the same record, setting it to two different values. There is little doubt that this is a conflict.

Other kinds of conflict can be more subtle to detect. For example, consider a meeting room booking system: it tracks which room is booked by which group of people at which time. This application needs to ensure that each room is only booked by one group of people at any one time (i.e., there must not be any overlapping bookings for the same room). In this case, a conflict may arise if two different bookings are created for the same room at the same time. Even if the application checks availability before

allowing a user to make a booking, there can be a conflict if the two bookings are made on two different leaders.

There isn't a quick ready-made answer, but in the following chapters we will trace a path toward a good understanding of this problem. We will see some more examples of conflicts in Chapter 7, and in Chapter 12 we will discuss scalable approaches for detecting and resolving conflicts in a replicated system.

## Multi-Leader Replication Topologies

A *replication topology* describes the communication paths along which writes are propagated from one node to another. If you have two leaders, like in Figure 5-7, there is only one plausible topology: leader 1 must send all of its writes to leader 2, and vice versa. With more than two leaders, various different topologies are possible. Some examples are illustrated in Figure 5-8.

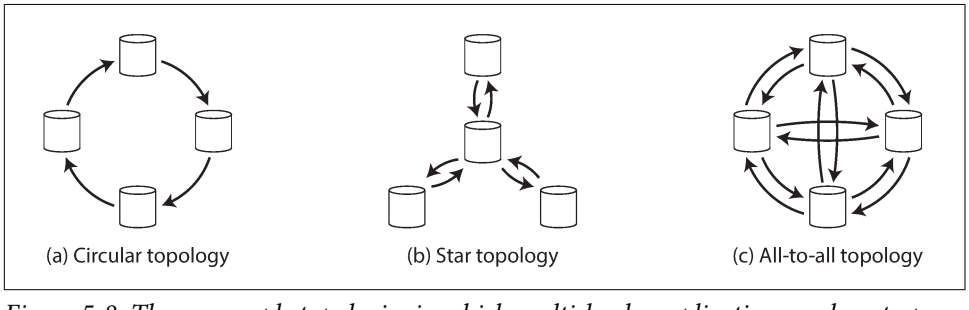



*Figure 5-8. Three example topologies in which multi-leader replication can be set up.*

The most general topology is *all-to-all* (Figure 5-8 [c]), in which every leader sends its writes to every other leader. However, more restricted topologies are also used: for example, MySQL by default supports only a *circular topology* [34], in which each node receives writes from one node and forwards those writes (plus any writes of its own) to one other node. Another popular topology has the shape of a *star*:[v] one designated root node forwards writes to all of the other nodes. The star topology can be generalized to a tree.

In circular and star topologies, a write may need to pass through several nodes before it reaches all replicas. Therefore, nodes need to forward data changes they receive from other nodes. To prevent infinite replication loops, each node is given a unique identifier, and in the replication log, each write is tagged with the identifiers of all the nodes it has passed through [43]. When a node receives a data change that is tagged

v. Not to be confused with a *star schema* (see "Stars and Snowflakes: Schemas for Analytics" on page 93), which describes the structure of a data model, not the communication topology between nodes.

with its own identifier, that data change is ignored, because the node knows that it has already been processed.

A problem with circular and star topologies is that if just one node fails, it can interrupt the flow of replication messages between other nodes, causing them to be unable to communicate until the node is fixed. The topology could be reconfigured to work around the failed node, but in most deployments such reconfiguration would have to be done manually. The fault tolerance of a more densely connected topology (such as all-to-all) is better because it allows messages to travel along different paths, avoiding a single point of failure.

On the other hand, all-to-all topologies can have issues too. In particular, some network links may be faster than others (e.g., due to network congestion), with the result that some replication messages may "overtake" others, as illustrated in Figure 5-9.

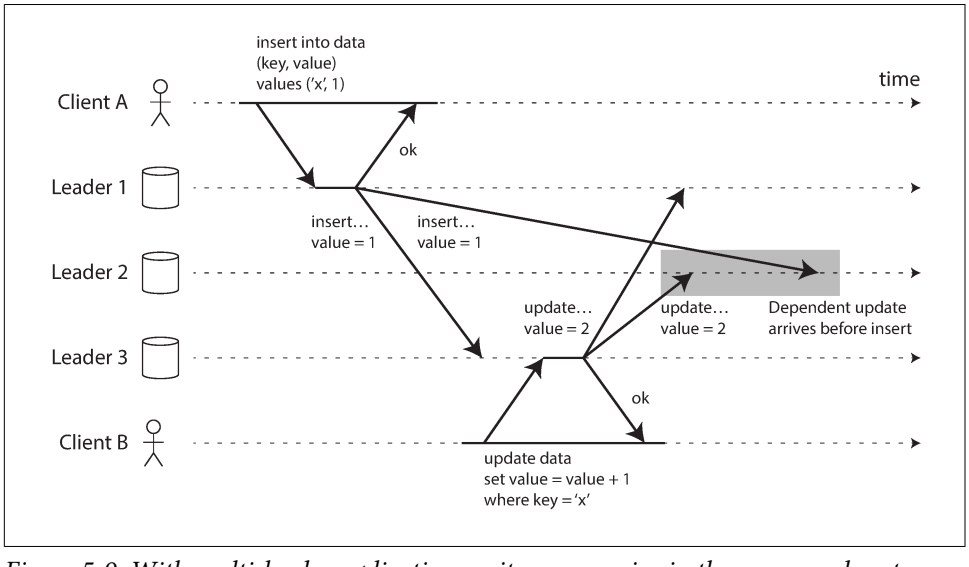



*Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.*

In Figure 5-9, client A inserts a row into a table on leader 1, and client B updates that row on leader 3. However, leader 2 may receive the writes in a different order: it may first receive the update (which, from its point of view, is an update to a row that does not exist in the database) and only later receive the corresponding insert (which should have preceded the update).

This is a problem of causality, similar to the one we saw in "Consistent Prefix Reads" on page 165: the update depends on the prior insert, so we need to make sure that all nodes process the insert first, and then the update. Simply attaching a timestamp to

every write is not sufficient, because clocks cannot be trusted to be sufficiently in sync to correctly order these events at leader 2 (see Chapter 8).

To order these events correctly, a technique called *version vectors* can be used, which we will discuss later in this chapter (see "Detecting Concurrent Writes" on page 184). However, conflict detection techniques are poorly implemented in many multi-leader replication systems. For example, at the time of writing, PostgreSQL BDR does not provide causal ordering of writes [27], and Tungsten Replicator for MySQL doesn't even try to detect conflicts [34].

If you are using a system with multi-leader replication, it is worth being aware of these issues, carefully reading the documentation, and thoroughly testing your database to ensure that it really does provide the guarantees you believe it to have.

# Leaderless Replication

The replication approaches we have discussed so far in this chapter—single-leader and multi-leader replication—are based on the idea that a client sends a write request to one node (the leader), and the database system takes care of copying that write to the other replicas. A leader determines the order in which writes should be processed, and followers apply the leader's writes in the same order.

Some data storage systems take a different approach, abandoning the concept of a leader and allowing any replica to directly accept writes from clients. Some of the earliest replicated data systems were leaderless [1, 44], but the idea was mostly forgotten during the era of dominance of relational databases. It once again became a fashionable architecture for databases after Amazon used it for its in-house *Dynamo* system [37].[vi] Riak, Cassandra, and Voldemort are open source datastores with leaderless replication models inspired by Dynamo, so this kind of database is also known as *Dynamo-style*.

In some leaderless implementations, the client directly sends its writes to several replicas, while in others, a coordinator node does this on behalf of the client. However, unlike a leader database, that coordinator does not enforce a particular ordering of writes. As we shall see, this difference in design has profound consequences for the way the database is used.

## Writing to the Database When a Node Is Down

Imagine you have a database with three replicas, and one of the replicas is currently unavailable—perhaps it is being rebooted to install a system update. In a leader-based

vi. Dynamo is not available to users outside of Amazon. Confusingly, AWS offers a hosted database product called *DynamoDB*, which uses a completely different architecture: it is based on single-leader replication.

configuration, if you want to continue processing writes, you may need to perform a failover (see "Handling Node Outages" on page 156).

On the other hand, in a leaderless configuration, failover does not exist. Figure 5-10 shows what happens: the client (user 1234) sends the write to all three replicas in parallel, and the two available replicas accept the write but the unavailable replica misses it. Let's say that it's sufficient for two out of three replicas to acknowledge the write: after user 1234 has received two *ok* responses, we consider the write to be successful. The client simply ignores the fact that one of the replicas missed the write.

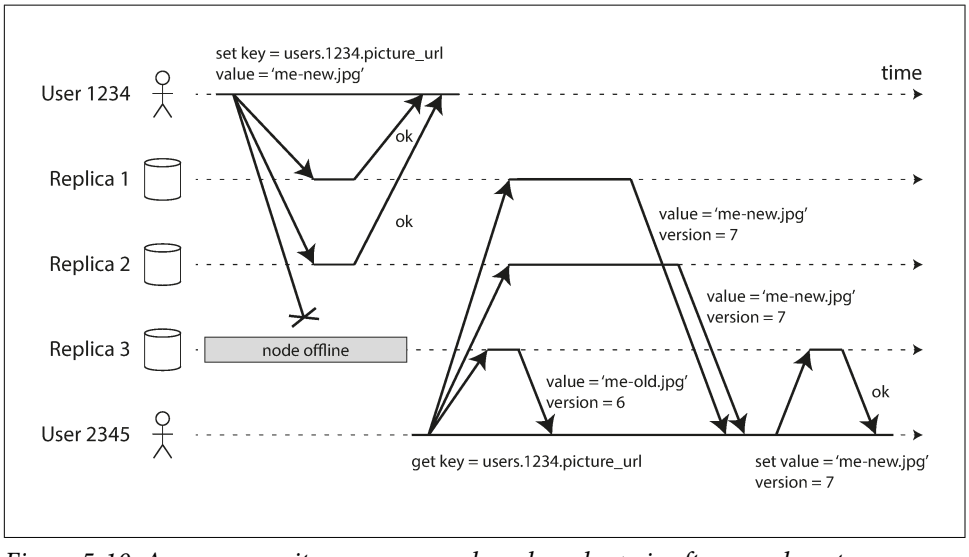



*Figure 5-10. A quorum write, quorum read, and read repair after a node outage.*

Now imagine that the unavailable node comes back online, and clients start reading from it. Any writes that happened while the node was down are missing from that node. Thus, if you read from that node, you may get *stale* (outdated) values as responses.

To solve that problem, when a client reads from the database, it doesn't just send its request to one replica: *read requests are also sent to several nodes in parallel*. The client may get different responses from different nodes; i.e., the up-to-date value from one node and a stale value from another. Version numbers are used to determine which value is newer (see "Detecting Concurrent Writes" on page 184).

### Read repair and anti-entropy

The replication system should ensure that eventually all the data is copied to every replica. After an unavailable node comes back online, how does it catch up on the writes that it missed?

Two mechanisms are often used in Dynamo-style datastores:

*Read repair*
: When a client makes a read from several nodes in parallel, it can detect any stale responses. For example, in Figure 5-10, user 2345 gets a version 6 value from replica 3 and a version 7 value from replicas 1 and 2. The client sees that replica 3 has a stale value and writes the newer value back to that replica. This approach works well for values that are frequently read.

*Anti-entropy process*
: In addition, some datastores have a background process that constantly looks for differences in the data between replicas and copies any missing data from one replica to another. Unlike the replication log in leader-based replication, this *anti-entropy process* does not copy writes in any particular order, and there may be a significant delay before data is copied.

Not all systems implement both of these; for example, Voldemort currently does not have an anti-entropy process. Note that without an anti-entropy process, values that are rarely read may be missing from some replicas and thus have reduced durability, because read repair is only performed when a value is read by the application.

### Quorums for reading and writing

In the example of Figure 5-10, we considered the write to be successful even though it was only processed on two out of three replicas. What if only one out of three replicas accepted the write? How far can we push this?

If we know that every successful write is guaranteed to be present on at least two out of three replicas, that means at most one replica can be stale. Thus, if we read from at least two replicas, we can be sure that at least one of the two is up to date. If the third replica is down or slow to respond, reads can nevertheless continue returning an up-to-date value.

More generally, if there are $n$ replicas, every write must be confirmed by $w$ nodes to be considered successful, and we must query at least $r$ nodes for each read. (In our example, $n = 3$, $w = 2$, $r = 2$.) As long as $w + r > n$, we expect to get an up-to-date value when reading, because at least one of the $r$ nodes we're reading from must be up to date. Reads and writes that obey these $r$ and $w$ values are called *quorum* reads and writes [44].[vii] You can think of $r$ and $w$ as the minimum number of votes required for the read or write to be valid.

vii. Sometimes this kind of quorum is called a *strict quorum*, to contrast with *sloppy quorums* (discussed in "Sloppy Quorums and Hinted Handoff" on page 183).

In Dynamo-style databases, the parameters $n$, $w$, and $r$ are typically configurable. A common choice is to make $n$ an odd number (typically 3 or 5) and to set $w = r = (n + 1) / 2$ (rounded up). However, you can vary the numbers as you see fit. For example, a workload with few writes and many reads may benefit from setting $w = n$ and $r = 1$. This makes reads faster, but has the disadvantage that just one failed node causes all database writes to fail.

There may be more than $n$ nodes in the cluster, but any given value is stored only on $n$ nodes. This allows the dataset to be partitioned, supporting datasets that are larger than you can fit on one node. We will return to partitioning in Chapter 6.

The quorum condition, $w + r > n$, allows the system to tolerate unavailable nodes as follows:

- If $w < n$, we can still process writes if a node is unavailable.
- If $r < n$, we can still process reads if a node is unavailable.
- With $n = 3$, $w = 2$, $r = 2$ we can tolerate one unavailable node.
- With $n = 5$, $w = 3$, $r = 3$ we can tolerate two unavailable nodes. This case is illustrated in Figure 5-11.
- Normally, reads and writes are always sent to all $n$ replicas in parallel. The parameters $w$ and $r$ determine how many nodes we wait for—i.e., how many of the $n$ nodes need to report success before we consider the read or write to be successful.

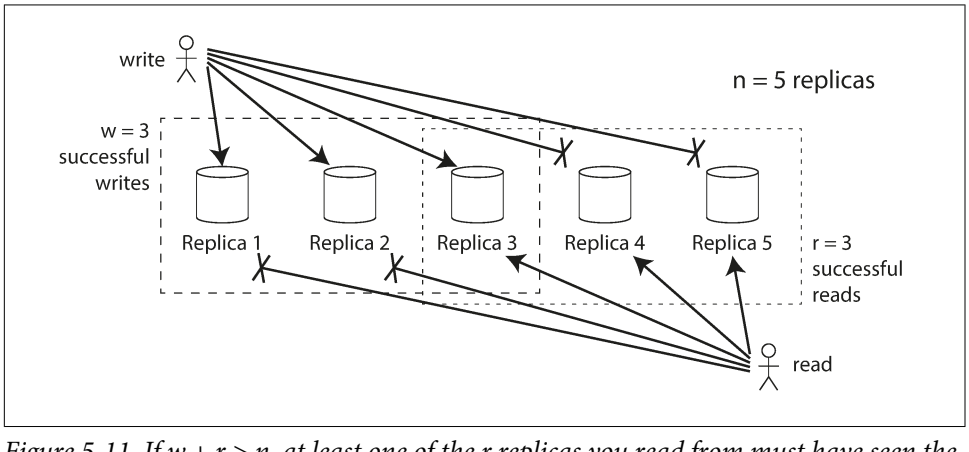



*Figure 5-11. If $w + r > n$, at least one of the r replicas you read from must have seen the most recent successful write.*

If fewer than the required $w$ or $r$ nodes are available, writes or reads return an error. A node could be unavailable for many reasons: because the node is down (crashed, powered down), due to an error executing the operation (can't write because the disk is full), due to a network interruption between the client and the node, or for any number of other reasons. We only care whether the node returned a successful response and don't need to distinguish between different kinds of fault.

## Limitations of Quorum Consistency

If you have $n$ replicas, and you choose $w$ and $r$ such that $w + r > n$, you can generally expect every read to return the most recent value written for a key. This is the case because the set of nodes to which you've written and the set of nodes from which you've read must overlap. That is, among the nodes you read there must be at least one node with the latest value (illustrated in Figure 5-11).

Often, $r$ and $w$ are chosen to be a majority (more than $n/2$) of nodes, because that ensures $w + r > n$ while still tolerating up to $n/2$ (rounded down) node failures. But quorums are not necessarily majorities—it only matters that the sets of nodes used by the read and write operations overlap in at least one node. Other quorum assignments are possible, which allows some flexibility in the design of distributed algorithms [45].

You may also set $w$ and $r$ to smaller numbers, so that $w + r \leq n$ (i.e., the quorum condition is not satisfied). In this case, reads and writes will still be sent to $n$ nodes, but a smaller number of successful responses is required for the operation to succeed.

With a smaller $w$ and $r$ you are more likely to read stale values, because it's more likely that your read didn't include the node with the latest value. On the upside, this configuration allows lower latency and higher availability: if there is a network interruption and many replicas become unreachable, there's a higher chance that you can continue processing reads and writes. Only after the number of reachable replicas falls below $w$ or $r$ does the database become unavailable for writing or reading, respectively.

However, even with $w + r > n$, there are likely to be edge cases where stale values are returned. These depend on the implementation, but possible scenarios include:

- If a sloppy quorum is used (see "Sloppy Quorums and Hinted Handoff" on page 183), the $w$ writes may end up on different nodes than the $r$ reads, so there is no longer a guaranteed overlap between the $r$ nodes and the $w$ nodes [46].
- If two writes occur concurrently, it is not clear which one happened first. In this case, the only safe solution is to merge the concurrent writes (see "Handling Write Conflicts" on page 171). If a winner is picked based on a timestamp (last write wins), writes can be lost due to clock skew [35]. We will return to this topic in "Detecting Concurrent Writes" on page 184.

- If a write happens concurrently with a read, the write may be reflected on only some of the replicas. In this case, it's undetermined whether the read returns the old or the new value.
- If a write succeeded on some replicas but failed on others (for example because the disks on some nodes are full), and overall succeeded on fewer than *w* replicas, it is not rolled back on the replicas where it succeeded. This means that if a write was reported as failed, subsequent reads may or may not return the value from that write [47].
- If a node carrying a new value fails, and its data is restored from a replica carrying an old value, the number of replicas storing the new value may fall below *w*, breaking the quorum condition.
- Even if everything is working correctly, there are edge cases in which you can get unlucky with the timing, as we shall see in "Linearizability and quorums" on page 334.

Thus, although quorums appear to guarantee that a read returns the latest written value, in practice it is not so simple. Dynamo-style databases are generally optimized for use cases that can tolerate eventual consistency. The parameters *w* and *r* allow you to adjust the probability of stale values being read, but it's wise to not take them as absolute guarantees.

In particular, you usually do not get the guarantees discussed in "Problems with Replication Lag" on page 161 (reading your writes, monotonic reads, or consistent prefix reads), so the previously mentioned anomalies can occur in applications. Stronger guarantees generally require transactions or consensus. We will return to these topics in Chapter 7 and Chapter 9.

### Monitoring staleness

From an operational perspective, it's important to monitor whether your databases are returning up-to-date results. Even if your application can tolerate stale reads, you need to be aware of the health of your replication. If it falls behind significantly, it should alert you so that you can investigate the cause (for example, a problem in the network or an overloaded node).

For leader-based replication, the database typically exposes metrics for the replication lag, which you can feed into a monitoring system. This is possible because writes are applied to the leader and to followers in the same order, and each node has a position in the replication log (the number of writes it has applied locally). By subtracting a follower's current position from the leader's current position, you can measure the amount of replication lag.

However, in systems with leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult. Moreover, if the database

only uses read repair (no anti-entropy), there is no limit to how old a value might be—if a value is only infrequently read, the value returned by a stale replica may be ancient.

There has been some research on measuring replica staleness in databases with leaderless replication and predicting the expected percentage of stale reads depending on the parameters *n*, *w*, and *r* [48]. This is unfortunately not yet common practice, but it would be good to include staleness measurements in the standard set of metrics for databases. Eventual consistency is a deliberately vague guarantee, but for operability it's important to be able to quantify "eventual."

## Sloppy Quorums and Hinted Handoff

Databases with appropriately configured quorums can tolerate the failure of individual nodes without the need for failover. They can also tolerate individual nodes going slow (e.g. due to overload), because requests don't have to wait for all *n* nodes to respond—they can return when *w* or *r* nodes have responded. These characteristics make databases with leaderless replication appealing for use cases that require high availability and low latency, and that can tolerate occasional stale reads.

However, quorums (as described so far) are not as fault-tolerant as they could be. A network interruption can easily cut off a client from a large number of database nodes. Although those nodes are alive, and other clients may be able to connect to them, to a client that is cut off from the database nodes, they might as well be dead. In this situation, it's likely that fewer than *w* or *r* reachable nodes remain, so the client can no longer reach a quorum.

In a large cluster (with significantly more than *n* nodes) it's likely that the client can connect to *some* database nodes during the network interruption, just not to the nodes that it needs to assemble a quorum for a particular value. In that case, database designers face a trade-off:

- Is it better to return errors to all requests for which we cannot reach a quorum of *w* or *r* nodes?
- Or should we accept writes anyway, and write them to some nodes that are reachable but aren't among the *n* nodes on which the value usually lives?

The latter is known as a *sloppy quorum* [37]: writes and reads still require *w* and *r* successful responses, but those may include nodes that are not among the designated *n* "home" nodes for a value. By analogy, if you lock yourself out of your house, you may knock on the neighbor's door and ask whether you may stay on their couch temporarily.

Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate "home" nodes. This is

called *hinted handoff*. (Once you find the keys to your house again, your neighbor politely asks you to get off their couch and go home.)

Sloppy quorums are particularly useful for increasing write availability: as long as *any* $w$ nodes are available, the database can accept writes. However, this means that even when $w + r > n$, you cannot be sure to read the latest value for a key, because the latest value may have been temporarily written to some nodes outside of $n$ [47].

Thus, a sloppy quorum actually isn't a quorum at all in the traditional sense. It's only an assurance of durability, namely that the data is stored on $w$ nodes somewhere. There is no guarantee that a read of $r$ nodes will see it until the hinted handoff has completed.

Sloppy quorums are optional in all common Dynamo implementations. In Riak they are enabled by default, and in Cassandra and Voldemort they are disabled by default [46, 49, 50].

### Multi-datacenter operation

We previously discussed cross-datacenter replication as a use case for multi-leader replication (see "Multi-Leader Replication" on page 168). Leaderless replication is also suitable for multi-datacenter operation, since it is designed to tolerate conflicting concurrent writes, network interruptions, and latency spikes.

Cassandra and Voldemort implement their multi-datacenter support within the normal leaderless model: the number of replicas $n$ includes nodes in all datacenters, and in the configuration you can specify how many of the $n$ replicas you want to have in each datacenter. Each write from a client is sent to all replicas, regardless of datacenter, but the client usually only waits for acknowledgment from a quorum of nodes within its local datacenter so that it is unaffected by delays and interruptions on the cross-datacenter link. The higher-latency writes to other datacenters are often configured to happen asynchronously, although there is some flexibility in the configuration [50, 51].

Riak keeps all communication between clients and database nodes local to one datacenter, so $n$ describes the number of replicas within one datacenter. Cross-datacenter replication between database clusters happens asynchronously in the background, in a style that is similar to multi-leader replication [52].

## Detecting Concurrent Writes

Dynamo-style databases allow several clients to concurrently write to the same key, which means that conflicts will occur even if strict quorums are used. The situation is similar to multi-leader replication (see "Handling Write Conflicts" on page 171), although in Dynamo-style databases conflicts can also arise during read repair or hinted handoff.

The problem is that events may arrive in a different order at different nodes, due to variable network delays and partial failures. For example, Figure 5-12 shows two clients, A and B, simultaneously writing to a key *X* in a three-node datastore:

- Node 1 receives the write from A, but never receives the write from B due to a transient outage.
- Node 2 first receives the write from A, then the write from B.
- Node 3 first receives the write from B, then the write from A.

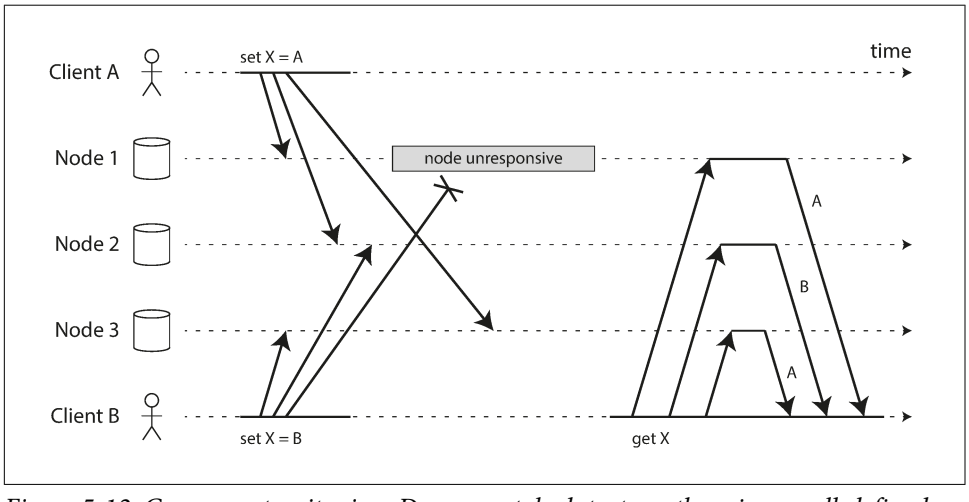



*Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.*

If each node simply overwrote the value for a key whenever it received a write request from a client, the nodes would become permanently inconsistent, as shown by the final *get* request in Figure 5-12: node 2 thinks that the final value of *X* is B, whereas the other nodes think that the value is A.

In order to become eventually consistent, the replicas should converge toward the same value. How do they do that? One might hope that replicated databases would handle this automatically, but unfortunately most implementations are quite poor: if you want to avoid losing data, you—the application developer—need to know a lot about the internals of your database's conflict handling.

We briefly touched on some techniques for conflict resolution in "Handling Write Conflicts" on page 171. Before we wrap up this chapter, let's explore the issue in a bit more detail.

### Last write wins (discarding concurrent writes)

One approach for achieving eventual convergence is to declare that each replica need only store the most "recent" value and allow "older" values to be overwritten and discarded. Then, as long as we have some way of unambiguously determining which write is more "recent," and every write is eventually copied to every replica, the replicas will eventually converge to the same value.

As indicated by the quotes around "recent," this idea is actually quite misleading. In the example of Figure 5-12, neither client knew about the other one when it sent its write requests to the database nodes, so it's not clear which one happened first. In fact, it doesn't really make sense to say that either happened "first": we say the writes are *concurrent*, so their order is undefined.

Even though the writes don't have a natural ordering, we can force an arbitrary order on them. For example, we can attach a timestamp to each write, pick the biggest timestamp as the most "recent," and discard any writes with an earlier timestamp. This conflict resolution algorithm, called *last write wins* (LWW), is the only supported conflict resolution method in Cassandra [53], and an optional feature in Riak [35].

LWW achieves the goal of eventual convergence, but at the cost of durability: if there are several concurrent writes to the same key, even if they were all reported as successful to the client (because they were written to *w* replicas), only one of the writes will survive and the others will be silently discarded. Moreover, LWW may even drop writes that are not concurrent, as we shall discuss in "Timestamps for ordering events" on page 291.

There are some situations, such as caching, in which lost writes are perhaps acceptable. If losing data is not acceptable, LWW is a poor choice for conflict resolution.

The only safe way of using a database with LWW is to ensure that a key is only written once and thereafter treated as immutable, thus avoiding any concurrent updates to the same key. For example, a recommended way of using Cassandra is to use a UUID as the key, thus giving each write operation a unique key [53].

### The "happens-before" relationship and concurrency

How do we decide whether two operations are concurrent or not? To develop an intuition, let's look at some examples:

- In Figure 5-9, the two writes are not concurrent: A's insert *happens before* B's increment, because the value incremented by B is the value inserted by A. In other words, B's operation builds upon A's operation, so B's operation must have happened later. We also say that B is *causally dependent* on A.

- On the other hand, the two writes in Figure 5-12 are concurrent: when each client starts the operation, it does not know that another client is also performing an operation on the same key. Thus, there is no causal dependency between the operations.

An operation A *happens before* another operation B if B knows about A, or depends on A, or builds upon A in some way. Whether one operation happens before another operation is the key to defining what concurrency means. In fact, we can simply say that two operations are *concurrent* if neither happens before the other (i.e., neither knows about the other) [54].

Thus, whenever you have two operations A and B, there are three possibilities: either A happened before B, or B happened before A, or A and B are concurrent. What we need is an algorithm to tell us whether two operations are concurrent or not. If one operation happened before another, the later operation should overwrite the earlier operation, but if the operations are concurrent, we have a conflict that needs to be resolved.

### Concurrency, Time, and Relativity

It may seem that two operations should be called concurrent if they occur "at the same time"—but in fact, it is not important whether they literally overlap in time. Because of problems with clocks in distributed systems, it is actually quite difficult to tell whether two things happened at exactly the same time—an issue we will discuss in more detail in Chapter 8.

For defining concurrency, exact time doesn't matter: we simply call two operations concurrent if they are both unaware of each other, regardless of the physical time at which they occurred. People sometimes make a connection between this principle and the special theory of relativity in physics [54], which introduced the idea that information cannot travel faster than the speed of light. Consequently, two events that occur some distance apart cannot possibly affect each other if the time between the events is shorter than the time it takes light to travel the distance between them.

In computer systems, two operations might be concurrent even though the speed of light would in principle have allowed one operation to affect the other. For example, if the network was slow or interrupted at the time, two operations can occur some time apart and still be concurrent, because the network problems prevented one operation from being able to know about the other.

#### Capturing the happens-before relationship

Let's look at an algorithm that determines whether two operations are concurrent, or whether one happened before another. To keep things simple, let's start with a data-

base that has only one replica. Once we have worked out how to do this on a single replica, we can generalize the approach to a leaderless database with multiple replicas.

Figure 5-13 shows two clients concurrently adding items to the same shopping cart. (If that example strikes you as too inane, imagine instead two air traffic controllers concurrently adding aircraft to the sector they are tracking.) Initially, the cart is empty. Between them, the clients make five writes to the database:

1. Client 1 adds `milk` to the cart. This is the first write to that key, so the server successfully stores it and assigns it version 1. The server also echoes the value back to the client, along with the version number.
2. Client 2 adds `eggs` to the cart, not knowing that client 1 concurrently added `milk` (client 2 thought that its `eggs` were the only item in the cart). The server assigns version 2 to this write, and stores `eggs` and `milk` as two separate values. It then returns *both* values to the client, along with the version number of 2.
3. Client 1, oblivious to client 2's write, wants to add `flour` to the cart, so it thinks the current cart contents should be `[milk, flour]`. It sends this value to the server, along with the version number 1 that the server gave client 1 previously. The server can tell from the version number that the write of `[milk, flour]` supersedes the prior value of `[milk]` but that it is concurrent with `[eggs]`. Thus, the server assigns version 3 to `[milk, flour]`, overwrites the version 1 value `[milk]`, but keeps the version 2 value `[eggs]` and returns both remaining values to the client.
4. Meanwhile, client 2 wants to add `ham` to the cart, unaware that client 1 just added `flour`. Client 2 received the two values `[milk]` and `[eggs]` from the server in the last response, so the client now merges those values and adds `ham` to form a new value, `[eggs, milk, ham]`. It sends that value to the server, along with the previous version number 2. The server detects that version 2 overwrites `[eggs]` but is concurrent with `[milk, flour]`, so the two remaining values are `[milk, flour]` with version 3, and `[eggs, milk, ham]` with version 4.
5. Finally, client 1 wants to add `bacon`. It previously received `[milk, flour]` and `[eggs]` from the server at version 3, so it merges those, adds `bacon`, and sends the final value `[milk, flour, eggs, bacon]` to the server, along with the version number 3. This overwrites `[milk, flour]` (note that `[eggs]` was already overwritten in the last step) but is concurrent with `[eggs, milk, ham]`, so the server keeps those two concurrent values.

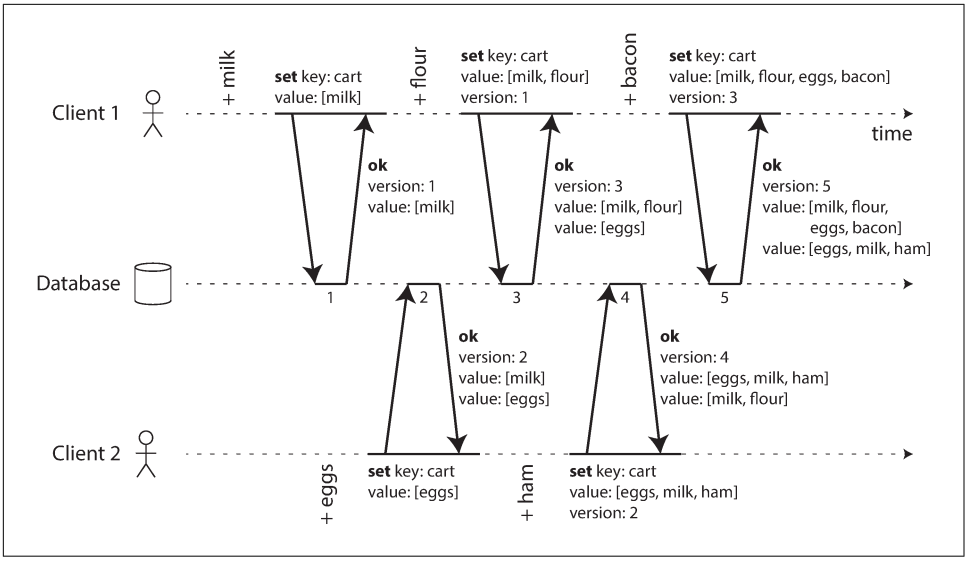



*Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.*

The dataflow between the operations in Figure 5-13 is illustrated graphically in Figure 5-14. The arrows indicate which operation *happened before* which other operation, in the sense that the later operation *knew about* or *depended on* the earlier one. In this example, the clients are never fully up to date with the data on the server, since there is always another operation going on concurrently. But old versions of the value do get overwritten eventually, and no writes are lost.

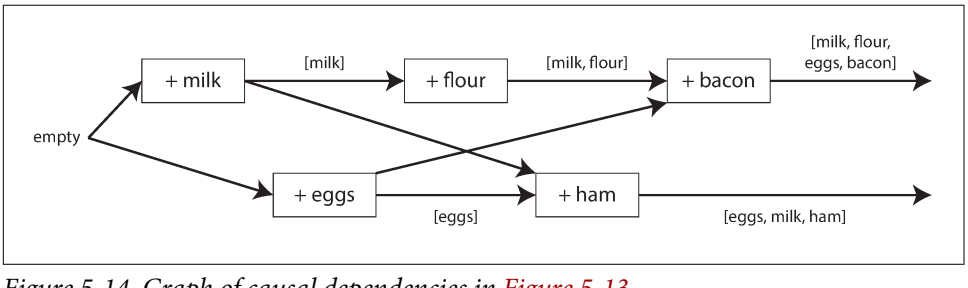



*Figure 5-14. Graph of causal dependencies in Figure 5-13.*

Note that the server can determine whether two operations are concurrent by looking at the version numbers—it does not need to interpret the value itself (so the value could be any data structure). The algorithm works as follows:

- The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written.
- When a client reads a key, the server returns all values that have not been overwritten, as well as the latest version number. A client must read a key before writing.
- When a client writes a key, it must include the version number from the prior read, and it must merge together all values that it received in the prior read. (The response from a write request can be like a read, returning all current values, which allows us to chain several writes like in the shopping cart example.)
- When the server receives a write with a particular version number, it can overwrite all values with that version number or below (since it knows that they have been merged into the new value), but it must keep all values with a higher version number (because those values are concurrent with the incoming write).

When a write includes the version number from a prior read, that tells us which previous state the write is based on. If you make a write without including a version number, it is concurrent with all other writes, so it will not overwrite anything—it will just be returned as one of the values on subsequent reads.

### Merging concurrently written values

This algorithm ensures that no data is silently dropped, but it unfortunately requires that the clients do some extra work: if several operations happen concurrently, clients have to clean up afterward by merging the concurrently written values. Riak calls these concurrent values *siblings*.

Merging sibling values is essentially the same problem as conflict resolution in multi-leader replication, which we discussed previously (see "Handling Write Conflicts" on page 171). A simple approach is to just pick one of the values based on a version number or timestamp (last write wins), but that implies losing data. So, you may need to do something more intelligent in application code.

With the example of a shopping cart, a reasonable approach to merging siblings is to just take the union. In Figure 5-14, the two final siblings are `[milk, flour, eggs, bacon]` and `[eggs, milk, ham]`; note that `milk` and `eggs` appear in both, even though they were each only written once. The merged value might be something like `[milk, flour, eggs, bacon, ham]`, without duplicates.

However, if you want to allow people to also *remove* things from their carts, and not just add things, then taking the union of siblings may not yield the right result: if you merge two sibling carts and an item has been removed in only one of them, then the removed item will reappear in the union of the siblings [37]. To prevent this prob-

lem, an item cannot simply be deleted from the database when it is removed; instead, the system must leave a marker with an appropriate version number to indicate that the item has been removed when merging siblings. Such a deletion marker is known as a *tombstone*. (We previously saw tombstones in the context of log compaction in "Hash Indexes" on page 72.)

As merging siblings in application code is complex and error-prone, there are some efforts to design data structures that can perform this merging automatically, as discussed in "Automatic Conflict Resolution" on page 174. For example, Riak's datatype support uses a family of data structures called CRDTs [38, 39, 55] that can automatically merge siblings in sensible ways, including preserving deletions.

### Version vectors

The example in Figure 5-13 used only a single replica. How does the algorithm change when there are multiple replicas, but no leader?

Figure 5-13 uses a single version number to capture dependencies between operations, but that is not sufficient when there are multiple replicas accepting writes concurrently. Instead, we need to use a version number *per replica* as well as per key. Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas. This information indicates which values to overwrite and which values to keep as siblings.

The collection of version numbers from all the replicas is called a *version vector* [56]. A few variants of this idea are in use, but the most interesting is probably the *dotted version vector* [57], which is used in Riak 2.0 [58, 59]. We won't go into the details, but the way it works is quite similar to what we saw in our cart example.

Like the version numbers in Figure 5-13, version vectors are sent from the database replicas to clients when values are read, and need to be sent back to the database when a value is subsequently written. (Riak encodes the version vector as a string that it calls *causal context*.) The version vector allows the database to distinguish between overwrites and concurrent writes.

Also, like in the single-replica example, the application may need to merge siblings. The version vector structure ensures that it is safe to read from one replica and subsequently write back to another replica. Doing so may result in siblings being created, but no data is lost as long as siblings are merged correctly.

#### Version vectors and vector clocks

A *version vector* is sometimes also called a *vector clock*, even though they are not quite the same. The difference is subtle—please see the references for details [57, 60, 61]. In brief, when comparing the state of replicas, version vectors are the right data structure to use.

## Summary

In this chapter we looked at the issue of replication. Replication can serve several purposes:

*High availability*
: Keeping the system running, even when one machine (or several machines, or an entire datacenter) goes down

*Disconnected operation*
: Allowing an application to continue working when there is a network interruption

*Latency*
: Placing data geographically close to users, so that users can interact with it faster

*Scalability*
: Being able to handle a higher volume of reads than a single machine could handle, by performing reads on replicas

Despite being a simple goal—keeping a copy of the same data on several machines—replication turns out to be a remarkably tricky problem. It requires carefully thinking about concurrency and about all the things that can go wrong, and dealing with the consequences of those faults. At a minimum, we need to deal with unavailable nodes and network interruptions (and that's not even considering the more insidious kinds of fault, such as silent data corruption due to software bugs).

We discussed three main approaches to replication:

*Single-leader replication*
: Clients send all writes to a single node (the leader), which sends a stream of data change events to the other replicas (followers). Reads can be performed on any replica, but reads from followers might be stale.

*Multi-leader replication*
: Clients send each write to one of several leader nodes, any of which can accept writes. The leaders send streams of data change events to each other and to any follower nodes.

*Leaderless replication*
: Clients send each write to several nodes, and read from several nodes in parallel in order to detect and correct nodes with stale data.

Each approach has advantages and disadvantages. Single-leader replication is popular because it is fairly easy to understand and there is no conflict resolution to worry about. Multi-leader and leaderless replication can be more robust in the presence of

faulty nodes, network interruptions, and latency spikes—at the cost of being harder to reason about and providing only very weak consistency guarantees.

Replication can be synchronous or asynchronous, which has a profound effect on the system behavior when there is a fault. Although asynchronous replication can be fast when the system is running smoothly, it's important to figure out what happens when replication lag increases and servers fail. If a leader fails and you promote an asynchronously updated follower to be the new leader, recently committed data may be lost.

We looked at some strange effects that can be caused by replication lag, and we discussed a few consistency models which are helpful for deciding how an application should behave under replication lag:

*Read-after-write consistency*
: Users should always see data that they submitted themselves.

*Monotonic reads*
: After users have seen the data at one point in time, they shouldn't later see the data from some earlier point in time.

*Consistent prefix reads*
: Users should see the data in a state that makes causal sense: for example, seeing a question and its reply in the correct order.

Finally, we discussed the concurrency issues that are inherent in multi-leader and leaderless replication approaches: because such systems allow multiple writes to happen concurrently, conflicts may occur. We examined an algorithm that a database might use to determine whether one operation happened before another, or whether they happened concurrently. We also touched on methods for resolving conflicts by merging together concurrently written values.

In the next chapter we will continue looking at data that is distributed across multiple machines, through the counterpart of replication: splitting a large dataset into *partitions*.

## References

[1] Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, et al.: "Notes on Distributed Databases," IBM Research, Research Report RJ2571(33471), July 1979.

[2] "Oracle Active Data Guard Real-Time Data Protection and Availability," Oracle White Paper, June 2013.

[3] "AlwaysOn Availability Groups," in *SQL Server Books Online*, Microsoft, 2012.

[4] Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: "On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

[5] Jun Rao: "Intra-Cluster Replication for Apache Kafka," at *ApacheCon North America*, February 2013.

[6] "Highly Available Queues," in *RabbitMQ Server Documentation*, Pivotal Software, Inc., 2014.

[7] Yoshinori Matsunobu: "Semi-Synchronous Replication at Facebook," *yoshinori-matsunobu.blogspot.co.uk*, April 1, 2014.

[8] Robbert van Renesse and Fred B. Schneider: "Chain Replication for Supporting High Throughput and Availability," at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

[9] Jeff Terrace and Michael J. Freedman: "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads," at *USENIX Annual Technical Conference* (ATC), June 2009.

[10] Brad Calder, Ju Wang, Aaron Ogus, et al.: "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," at *23rd ACM Symposium on Operating Systems Principles* (SOSP), October 2011.

[11] Andrew Wang: "Windows Azure Storage," *umbrant.com*, February 4, 2016.

[12] "Percona Xtrabackup - Documentation," Percona LLC, 2014.

[13] Jesse Newland: "GitHub Availability This Week," *github.com*, September 14, 2012.

[14] Mark Imbriaco: "Downtime Last Saturday," *github.com*, December 26, 2012.

[15] John Hugg: "'All in' with Determinism for Performance and Testing in Distributed Systems," at *Strange Loop*, September 2015.

[16] Amit Kapila: "WAL Internals of PostgreSQL," at *PostgreSQL Conference* (PGCon), May 2012.

[17] *MySQL Internals Manual*. Oracle, 2014.

[18] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: "Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

[19] "Oracle GoldenGate 12c: Real-Time Access to Real-Time Information," Oracle White Paper, October 2013.

[20] Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at *ACM Symposium on Cloud Computing* (SoCC), October 2012.

[21] Greg Sabino Mullane: "Version 5 of Bucardo Database Replication System," *blog.endpoint.com*, June 23, 2014.

[22] Werner Vogels: "Eventually Consistent," *ACM Queue*, volume 6, number 6, pages 14–19, October 2008. doi:10.1145/1466443.1466448

[23] Douglas B. Terry: "Replicated Data Consistency Explained Through Baseball," Microsoft Research, Technical Report MSR-TR-2011-137, October 2011.

[24] Douglas B. Terry, Alan J. Demers, Karin Petersen, et al.: "Session Guarantees for Weakly Consistent Replicated Data," at *3rd International Conference on Parallel and Distributed Information Systems* (PDIS), September 1994. doi:10.1109/PDIS.1994.331722

[25] Terry Pratchett: *Reaper Man: A Discworld Novel*. Victor Gollancz, 1991. ISBN: 978-0-575-04979-6

[26] "Tungsten Replicator," *github.com*.

[27] "BDR 0.10.0 Documentation," The PostgreSQL Global Development Group, *bdr-project.org*, 2015.

[28] Robert Hodges: "If You *Must* Deploy Multi-Master Replication, Read This First," *scale-out-blog.blogspot.co.uk*, March 30, 2012.

[29] J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6

[30] AppJet, Inc.: "Etherpad and EasySync Technical Manual," *github.com*, March 26, 2011.

[31] John Day-Richter: "What's Different About the New Google Docs: Making Collaboration Fast," *drive.googleblog.com*, September 23, 2010.

[32] Martin Kleppmann and Alastair R. Beresford: "A Conflict-Free Replicated JSON Datatype," arXiv:1608.03960, August 13, 2016.

[33] Frazer Clement: "Eventual Consistency – Detecting Conflicts," *messagepassing.blogspot.co.uk*, October 20, 2011.

[34] Robert Hodges: "State of the Art for MySQL Multi-Master Replication," at *Percona Live: MySQL Conference & Expo*, April 2013.

[35] John Daily: "Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems," *riak.com*, November 12, 2013.

[36] Riley Berton: "Is Bi-Directional Replication (BDR) in Postgres Transactional?," *sdf.org*, January 4, 2016.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: "Dynamo: Amazon's Highly Available Key-Value Store," at *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007.

[38] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski: "A Comprehensive Study of Convergent and Commutative Replicated Data Types," INRIA Research Report no. 7506, January 2011.

[39] Sam Elliott: "CRDTs: An UPDATE (or Maybe Just a PUT)," at *RICON West*, October 2013.

[40] Russell Brown: "A Bluffers Guide to CRDTs in Riak," *gist.github.com*, October 28, 2013.

[41] Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy: "Mergeable Persistent Data Structures," at *26es Journées Francophones des Langages Applicatifs* (JFLA), January 2015.

[42] Chengzheng Sun and Clarence Ellis: "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," at *ACM Conference on Computer Supported Cooperative Work* (CSCW), November 1998.

[43] Lars Hofhansl: "HBASE-7709: Infinite Loop Possible in Master/Master Replication," *issues.apache.org*, January 29, 2013.

[44] David K. Gifford: "Weighted Voting for Replicated Data," at *7th ACM Symposium on Operating Systems Principles* (SOSP), December 1979. doi:10.1145/800215.806583

[45] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman: "Flexible Paxos: Quorum Intersection Revisited," *arXiv:1608.06696*, August 24, 2016.

[46] Joseph Blomstedt: "Re: Absolute Consistency," email to *riak-users* mailing list, *lists.basho.com*, January 11, 2012.

[47] Joseph Blomstedt: "Bringing Consistency to Riak," at *RICON West*, October 2012.

[48] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, et al.: "Quantifying Eventual Consistency with PBS," *Communications of the ACM*, volume 57, number 8, pages 93–102, August 2014. doi:10.1145/2632792

[49] Jonathan Ellis: "Modern Hinted Handoff," *datastax.com*, December 11, 2012.

[50] "Project Voldemort Wiki," *github.com*, 2013.

[51] "Apache Cassandra Documentation," Apache Software Foundation, *cassandra.apache.org*.

[52] "Riak Enterprise: Multi-Datacenter Replication." Technical whitepaper, Basho Technologies, Inc., September 2014.

[53] Jonathan Ellis: "Why Cassandra Doesn't Need Vector Clocks," *datastax.com*, September 2, 2013.

[54] Leslie Lamport: "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563

[55] Joel Jacobson: "Riak 2.0: Data Types," *blog.joeljacobson.com*, March 23, 2014.

[56] D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, et al.: "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Transactions on Software Engineering*, volume 9, number 3, pages 240–247, May 1983. doi:10.1109/TSE.1983.236733

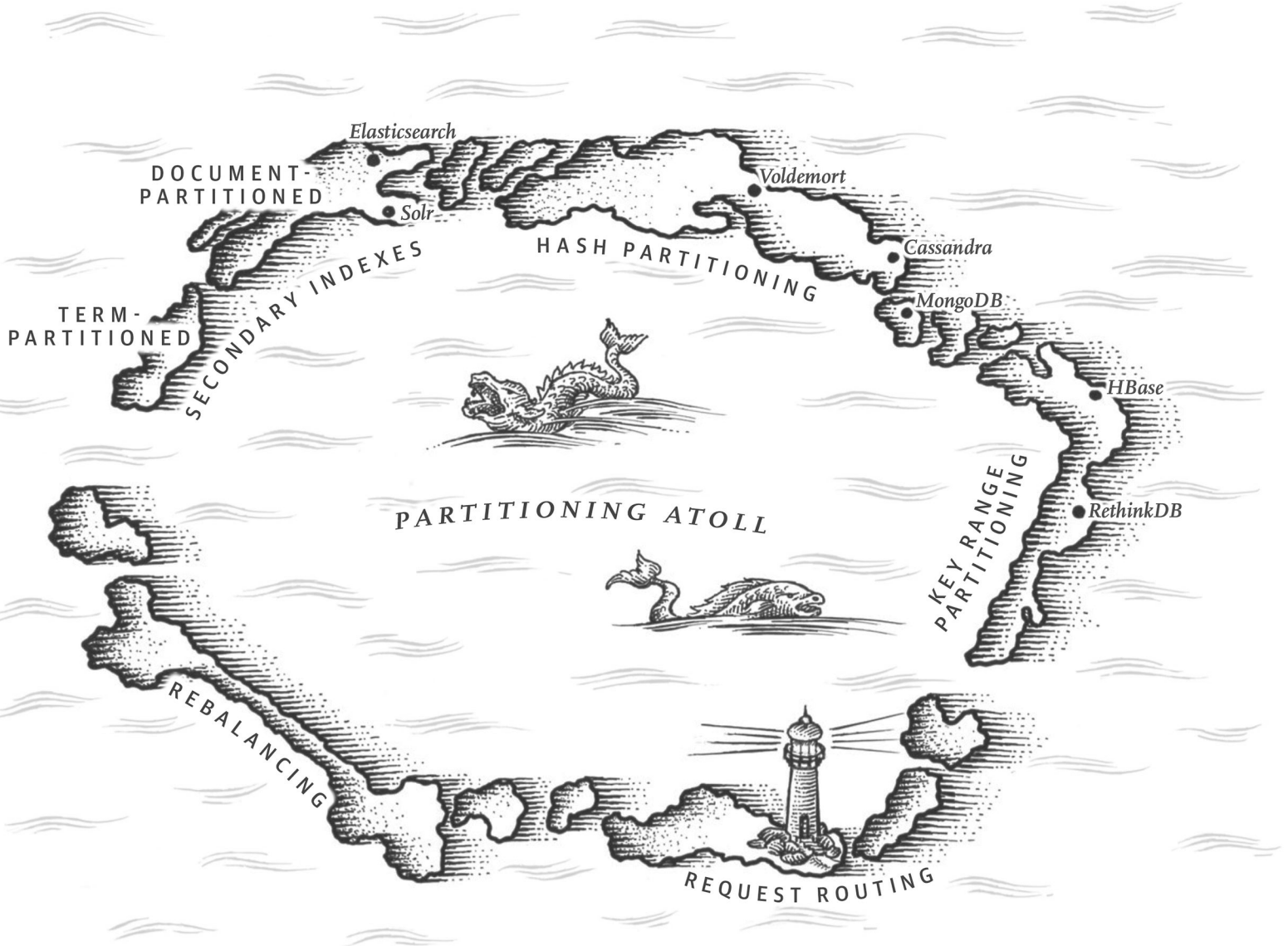
[57] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, et al.: "Dotted Version Vectors: Logical Clocks for Optimistic Replication," arXiv:1011.5808, November 26, 2010.

[58] Sean Cribbs: "A Brief History of Time in Riak," at *RICON*, October 2014.

[59] Russell Brown: "Vector Clocks Revisited Part 2: Dotted Version Vectors," *basho.com*, November 10, 2015.

[60] Carlos Baquero: "Version Vectors Are Not Vector Clocks," *haslab.wordpress.com*, July 8, 2011.

[61] Reinhard Schwarz and Friedemann Mattern: "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, volume 7, number 3, pages 149–174, March 1994. doi:10.1007/BF02277859



# CHAPTER 6
# Partitioning

> *Clearly, we must break away from the sequential and not limit the computers. We must state definitions and provide for priorities and descriptions of data. We must state relationships, not procedures.*
>
> —Grace Murray Hopper, *Management and the Computer of the Future* (1962)

In Chapter 5 we discussed replication—that is, having multiple copies of the same data on different nodes. For very large datasets, or very high query throughput, that is not sufficient: we need to break the data up into *partitions*, also known as *sharding*.[i]

### Terminological confusion

What we call a *partition* here is called a *shard* in MongoDB, Elasticsearch, and SolrCloud; it's known as a *region* in HBase, a *tablet* in Bigtable, a *vnode* in Cassandra and Riak, and a *vBucket* in Couchbase. However, *partitioning* is the most established term, so we'll stick with that.

Normally, partitions are defined in such a way that each piece of data (each record, row, or document) belongs to exactly one partition. There are various ways of achieving this, which we discuss in depth in this chapter. In effect, each partition is a small database of its own, although the database may support operations that touch multiple partitions at the same time.

The main reason for wanting to partition data is *scalability*. Different partitions can be placed on different nodes in a shared-nothing cluster (see the introduction to

i. Partitioning, as discussed in this chapter, is a way of intentionally breaking a large database down into smaller ones. It has nothing to do with *network partitions* (netsplits), a type of fault in the network between nodes. We will discuss such faults in Chapter 8.

Part II for a definition of *shared nothing*). Thus, a large dataset can be distributed across many disks, and the query load can be distributed across many processors.

For queries that operate on a single partition, each node can independently execute the queries for its own partition, so query throughput can be scaled by adding more nodes. Large, complex queries can potentially be parallelized across many nodes, although this gets significantly harder.

Partitioned databases were pioneered in the 1980s by products such as Teradata and Tandem NonStop SQL [1], and more recently rediscovered by NoSQL databases and Hadoop-based data warehouses. Some systems are designed for transactional workloads, and others for analytics (see "Transaction Processing or Analytics?" on page 90): this difference affects how the system is tuned, but the fundamentals of partitioning apply to both kinds of workloads.

In this chapter we will first look at different approaches for partitioning large datasets and observe how the indexing of data interacts with partitioning. We'll then talk about rebalancing, which is necessary if you want to add or remove nodes in your cluster. Finally, we'll get an overview of how databases route requests to the right partitions and execute queries.

## Partitioning and Replication

Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes. This means that, even though each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance.

A node may store more than one partition. If a leader–follower replication model is used, the combination of partitioning and replication can look like Figure 6-1, for example. Each partition's leader is assigned to one node, and its followers are assigned to other nodes. Each node may be the leader for some partitions and a follower for other partitions.

Everything we discussed in Chapter 5 about replication of databases applies equally to replication of partitions. The choice of partitioning scheme is mostly independent of the choice of replication scheme, so we will keep things simple and ignore replication in this chapter.

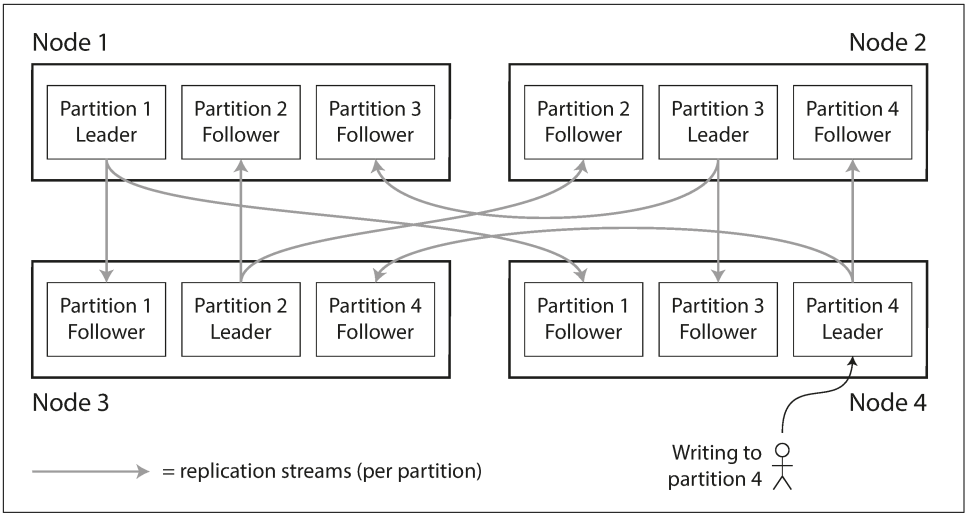



*Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.*

# Partitioning of Key-Value Data

Say you have a large amount of data, and you want to partition it. How do you decide which records to store on which nodes?

Our goal with partitioning is to spread the data and the query load evenly across nodes. If every node takes a fair share, then—in theory—10 nodes should be able to handle 10 times as much data and 10 times the read and write throughput of a single node (ignoring replication for now).

If the partitioning is unfair, so that some partitions have more data or queries than others, we call it *skewed*. The presence of skew makes partitioning much less effective. In an extreme case, all the load could end up on one partition, so 9 out of 10 nodes are idle and your bottleneck is the single busy node. A partition with disproportionately high load is called a *hot spot*.

The simplest approach for avoiding hot spots would be to assign records to nodes randomly. That would distribute the data quite evenly across the nodes, but it has a big disadvantage: when you're trying to read a particular item, you have no way of knowing which node it is on, so you have to query all nodes in parallel.

We can do better. Let's assume for now that you have a simple key-value data model, in which you always access a record by its primary key. For example, in an old-fashioned paper encyclopedia, you look up an entry by its title; since all the entries are alphabetically sorted by title, you can quickly find the one you're looking for.

## Partitioning by Key Range

One way of partitioning is to assign a continuous range of keys (from some minimum to some maximum) to each partition, like the volumes of a paper encyclopedia (Figure 6-2). If you know the boundaries between the ranges, you can easily determine which partition contains a given key. If you also know which partition is assigned to which node, then you can make your request directly to the appropriate node (or, in the case of the encyclopedia, pick the correct book off the shelf).

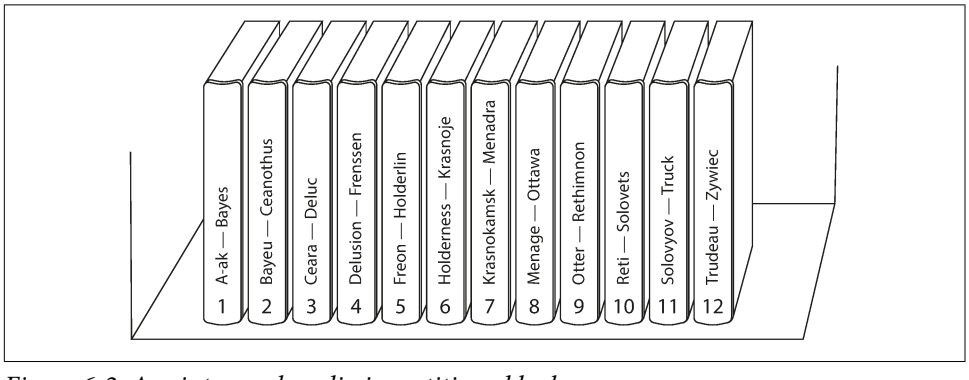



*Figure 6-2. A print encyclopedia is partitioned by key range.*

The ranges of keys are not necessarily evenly spaced, because your data may not be evenly distributed. For example, in Figure 6-2, volume 1 contains words starting with A and B, but volume 12 contains words starting with T, U, V, W, X, Y, and Z. Simply having one volume per two letters of the alphabet would lead to some volumes being much bigger than others. In order to distribute the data evenly, the partition boundaries need to adapt to the data.

The partition boundaries might be chosen manually by an administrator, or the database can choose them automatically (we will discuss choices of partition boundaries in more detail in "Rebalancing Partitions" on page 209). This partitioning strategy is used by Bigtable, its open source equivalent HBase [2, 3], RethinkDB, and MongoDB before version 2.4 [4].

Within each partition, we can keep keys in sorted order (see "SSTables and LSM-Trees" on page 76). This has the advantage that range scans are easy, and you can treat the key as a concatenated index in order to fetch several related records in one query (see "Multi-column indexes" on page 87). For example, consider an application that stores data from a network of sensors, where the key is the timestamp of the measurement (*year-month-day-hour-minute-second*). Range scans are very useful in this case, because they let you easily fetch, say, all the readings from a particular month.

However, the downside of key range partitioning is that certain access patterns can lead to hot spots. If the key is a timestamp, then the partitions correspond to ranges of time—e.g., one partition per day. Unfortunately, because we write data from the sensors to the database as the measurements happen, all the writes end up going to the same partition (the one for today), so that partition can be overloaded with writes while others sit idle [5].

To avoid this problem in the sensor database, you need to use something other than the timestamp as the first element of the key. For example, you could prefix each timestamp with the sensor name so that the partitioning is first by sensor name and then by time. Assuming you have many sensors active at the same time, the write load will end up more evenly spread across the partitions. Now, when you want to fetch the values of multiple sensors within a time range, you need to perform a separate range query for each sensor name.

## Partitioning by Hash of Key

Because of this risk of skew and hot spots, many distributed datastores use a hash function to determine the partition for a given key.

A good hash function takes skewed data and makes it uniformly distributed. Say you have a 32-bit hash function that takes a string. Whenever you give it a new string, it returns a seemingly random number between 0 and $2^{32} - 1$. Even if the input strings are very similar, their hashes are evenly distributed across that range of numbers.

For partitioning purposes, the hash function need not be cryptographically strong: for example, MongoDB uses MD5, Cassandra uses Murmur3, and Voldemort uses the Fowler–Noll–Vo function. Many programming languages have simple hash functions built in (as they are used for hash tables), but they may not be suitable for partitioning: for example, in Java's `Object.hashCode()` and Ruby's `Object#hash`, the same key may have a different hash value in different processes, making them unsuitable for partitioning [6].

Once you have a suitable hash function for keys, you can assign each partition a range of hashes (rather than a range of keys), and every key whose hash falls within a partition's range will be stored in that partition. This is illustrated in Figure 6-3.

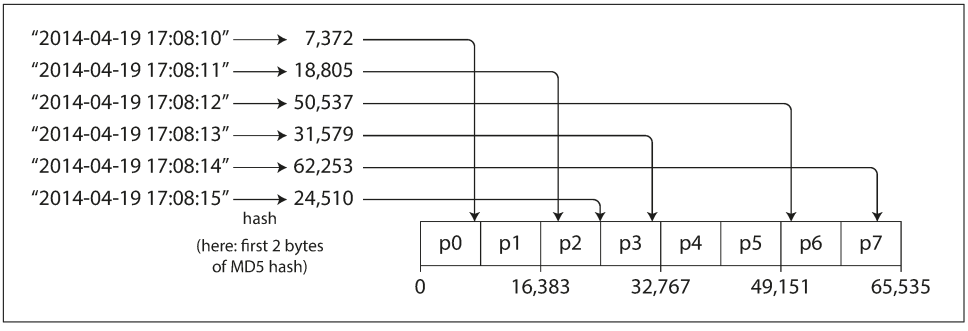



*Figure 6-3. Partitioning by hash of key.*

This technique is good at distributing keys fairly among the partitions. The partition boundaries can be evenly spaced, or they can be chosen pseudorandomly (in which case the technique is sometimes known as *consistent hashing*).

## Consistent Hashing

Consistent hashing, as defined by Karger et al. [7], is a way of evenly distributing load across an internet-wide system of caches such as a content delivery network (CDN). It uses randomly chosen partition boundaries to avoid the need for central control or distributed consensus. Note that *consistent* here has nothing to do with replica consistency (see Chapter 5) or ACID consistency (see Chapter 7), but rather describes a particular approach to rebalancing.

As we shall see in "Rebalancing Partitions" on page 209, this particular approach actually doesn't work very well for databases [8], so it is rarely used in practice (the documentation of some databases still refers to consistent hashing, but it is often inaccurate). Because this is so confusing, it's best to avoid the term *consistent hashing* and just call it *hash partitioning* instead.

Unfortunately however, by using the hash of the key for partitioning we lose a nice property of key-range partitioning: the ability to do efficient range queries. Keys that were once adjacent are now scattered across all the partitions, so their sort order is lost. In MongoDB, if you have enabled hash-based sharding mode, any range query has to be sent to all partitions [4]. Range queries on the primary key are not supported by Riak [9], Couchbase [10], or Voldemort.

Cassandra achieves a compromise between the two partitioning strategies [11, 12, 13]. A table in Cassandra can be declared with a *compound primary key* consisting of several columns. Only the first part of that key is hashed to determine the partition, but the other columns are used as a concatenated index for sorting the data in Cassandra's SSTables. A query therefore cannot search for a range of values within the

first column of a compound key, but if it specifies a fixed value for the first column, it can perform an efficient range scan over the other columns of the key.

The concatenated index approach enables an elegant data model for one-to-many relationships. For example, on a social media site, one user may post many updates. If the primary key for updates is chosen to be `(user_id, update_timestamp)`, then you can efficiently retrieve all updates made by a particular user within some time interval, sorted by timestamp. Different users may be stored on different partitions, but within each user, the updates are stored ordered by timestamp on a single partition.

## Skewed Workloads and Relieving Hot Spots

As discussed, hashing a key to determine its partition can help reduce hot spots. However, it can't avoid them entirely: in the extreme case where all reads and writes are for the same key, you still end up with all requests being routed to the same partition.

This kind of workload is perhaps unusual, but not unheard of: for example, on a social media site, a celebrity user with millions of followers may cause a storm of activity when they do something [14]. This event can result in a large volume of writes to the same key (where the key is perhaps the user ID of the celebrity, or the ID of the action that people are commenting on). Hashing the key doesn't help, as the hash of two identical IDs is still the same.

Today, most data systems are not able to automatically compensate for such a highly skewed workload, so it's the responsibility of the application to reduce the skew. For example, if one key is known to be very hot, a simple technique is to add a random number to the beginning or end of the key. Just a two-digit decimal random number would split the writes to the key evenly across 100 different keys, allowing those keys to be distributed to different partitions.

However, having split the writes across different keys, any reads now have to do additional work, as they have to read the data from all 100 keys and combine it. This technique also requires additional bookkeeping: it only makes sense to append the random number for the small number of hot keys; for the vast majority of keys with low write throughput this would be unnecessary overhead. Thus, you also need some way of keeping track of which keys are being split.

Perhaps in the future, data systems will be able to automatically detect and compensate for skewed workloads; but for now, you need to think through the trade-offs for your own application.

# Partitioning and Secondary Indexes

The partitioning schemes we have discussed so far rely on a key-value data model. If records are only ever accessed via their primary key, we can determine the partition from that key and use it to route read and write requests to the partition responsible for that key.

The situation becomes more complicated if secondary indexes are involved (see also "Other Indexing Structures" on page 85). A secondary index usually doesn't identify a record uniquely but rather is a way of searching for occurrences of a particular value: find all actions by user `123`, find all articles containing the word `hogwash`, find all cars whose color is `red`, and so on.

Secondary indexes are the bread and butter of relational databases, and they are common in document databases too. Many key-value stores (such as HBase and Voldemort) have avoided secondary indexes because of their added implementation complexity, but some (such as Riak) have started adding them because they are so useful for data modeling. And finally, secondary indexes are the *raison d'être* of search servers such as Solr and Elasticsearch.

The problem with secondary indexes is that they don't map neatly to partitions. There are two main approaches to partitioning a database with secondary indexes: document-based partitioning and term-based partitioning.

## Partitioning Secondary Indexes by Document

For example, imagine you are operating a website for selling used cars (illustrated in Figure 6-4). Each listing has a unique ID—call it the *document ID*—and you partition the database by the document ID (for example, IDs 0 to 499 in partition 0, IDs 500 to 999 in partition 1, etc.).

You want to let users search for cars, allowing them to filter by color and by make, so you need a secondary index on `color` and `make` (in a document database these would be fields; in a relational database they would be columns). If you have declared the index, the database can perform the indexing automatically.[ii] For example, whenever a red car is added to the database, the database partition automatically adds it to the list of document IDs for the index entry `color:red`.

ii. If your database only supports a key-value model, you might be tempted to implement a secondary index yourself by creating a mapping from values to document IDs in application code. If you go down this route, you need to take great care to ensure your indexes remain consistent with the underlying data. Race conditions and intermittent write failures (where some changes were saved but others weren't) can very easily cause the data to go out of sync—see "The need for multi-object transactions" on page 231.

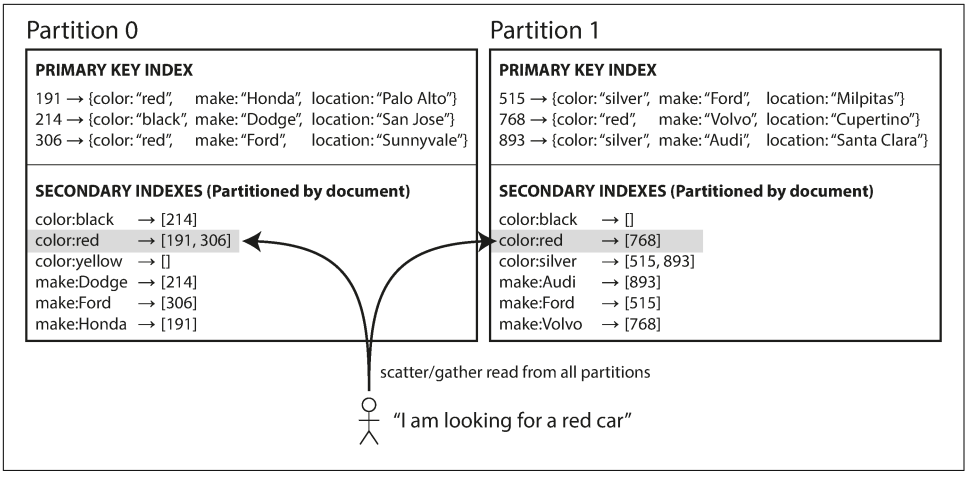



*Figure 6-4. Partitioning secondary indexes by document.*

In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition. It doesn't care what data is stored in other partitions. Whenever you write to the database—to add, remove, or update a document—you only need to deal with the partition that contains the document ID that you are writing. For that reason, a document-partitioned index is also known as a *local index* (as opposed to a *global index*, described in the next section).

However, reading from a document-partitioned index requires care: unless you have done something special with the document IDs, there is no reason why all the cars with a particular color or a particular make would be in the same partition. In Figure 6-4, red cars appear in both partition 0 and partition 1. Thus, if you want to search for red cars, you need to send the query to *all* partitions, and combine all the results you get back.

This approach to querying a partitioned database is sometimes known as *scatter/gather*, and it can make read queries on secondary indexes quite expensive. Even if you query the partitions in parallel, scatter/gather is prone to tail latency amplification (see "Percentiles in Practice" on page 16). Nevertheless, it is widely used: MongoDB, Riak [15], Cassandra [16], Elasticsearch [17], SolrCloud [18], and VoltDB [19] all use document-partitioned secondary indexes. Most database vendors recommend that you structure your partitioning scheme so that secondary index queries can be served from a single partition, but that is not always possible, especially when you're using multiple secondary indexes in a single query (such as filtering cars by color and by make at the same time).

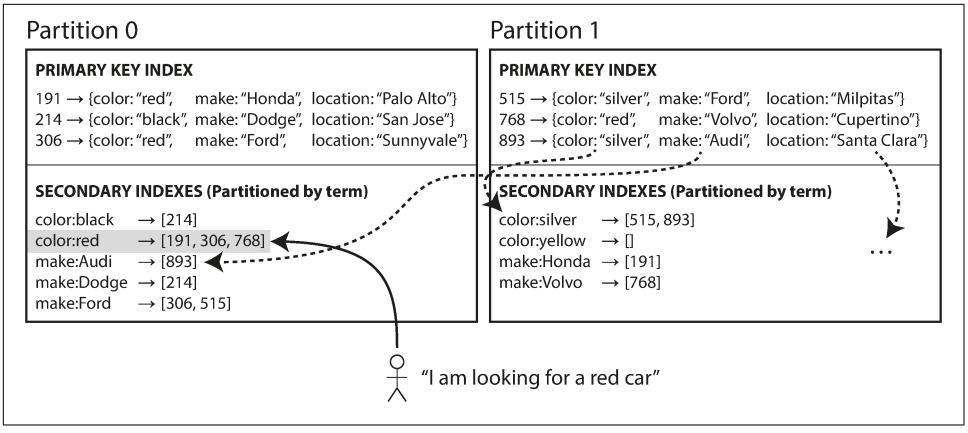



*Figure 6-5. Partitioning secondary indexes by term.*

## Partitioning Secondary Indexes by Term

Rather than each partition having its own secondary index (a *local index*), we can construct a *global index* that covers data in all partitions. However, we can't just store that index on one node, since it would likely become a bottleneck and defeat the purpose of partitioning. A global index must also be partitioned, but it can be partitioned differently from the primary key index.

Figure 6-5 illustrates what this could look like: red cars from all partitions appear under `color:red` in the index, but the index is partitioned so that colors starting with the letters *a* to *r* appear in partition 0 and colors starting with *s* to *z* appear in partition 1. The index on the make of car is partitioned similarly (with the partition boundary being between *f* and *h*).

We call this kind of index *term-partitioned*, because the term we're looking for determines the partition of the index. Here, a term would be `color:red`, for example. The name *term* comes from full-text indexes (a particular kind of secondary index), where the terms are all the words that occur in a document.

As before, we can partition the index by the term itself, or using a hash of the term. Partitioning by the term itself can be useful for range scans (e.g., on a numeric property, such as the asking price of the car), whereas partitioning on a hash of the term gives a more even distribution of load.

The advantage of a global (term-partitioned) index over a document-partitioned index is that it can make reads more efficient: rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants. However, the downside of a global index is that writes are slower and more complicated, because a write to a single document may now affect multiple

partitions of the index (every term in the document might be on a different partition, on a different node).

In an ideal world, the index would always be up to date, and every document written to the database would immediately be reflected in the index. However, in a term-partitioned index, that would require a distributed transaction across all partitions affected by a write, which is not supported in all databases (see Chapter 7 and Chapter 9).

In practice, updates to global secondary indexes are often asynchronous (that is, if you read the index shortly after a write, the change you just made may not yet be reflected in the index). For example, Amazon DynamoDB states that its global secondary indexes are updated within a fraction of a second in normal circumstances, but may experience longer propagation delays in cases of faults in the infrastructure [20].

Other uses of global term-partitioned indexes include Riak's search feature [21] and the Oracle data warehouse, which lets you choose between local and global indexing [22]. We will return to the topic of implementing term-partitioned secondary indexes in Chapter 12.

## Rebalancing Partitions

Over time, things change in a database:

- The query throughput increases, so you want to add more CPUs to handle the load.
- The dataset size increases, so you want to add more disks and RAM to store it.
- A machine fails, and other machines need to take over the failed machine's responsibilities.

All of these changes call for data and requests to be moved from one node to another. The process of moving load from one node in the cluster to another is called *rebalancing*.

No matter which partitioning scheme is used, rebalancing is usually expected to meet some minimum requirements:

- After rebalancing, the load (data storage, read and write requests) should be shared fairly between the nodes in the cluster.
- While rebalancing is happening, the database should continue accepting reads and writes.
- No more data than necessary should be moved between nodes, to make rebalancing fast and to minimize the network and disk I/O load.

## Strategies for Rebalancing

There are a few different ways of assigning partitions to nodes [23]. Let's briefly discuss each in turn.

### How not to do it: hash mod N

When partitioning by the hash of a key, we said earlier (Figure 6-3) that it's best to divide the possible hashes into ranges and assign each range to a partition (e.g., assign *key* to partition 0 if $0 \le hash(key) < b_0$, to partition 1 if $b_0 \le hash(key) < b_1$, etc.).

Perhaps you wondered why we don't just use *mod* (the `%` operator in many programming languages). For example, *hash*(*key*) *mod* 10 would return a number between 0 and 9 (if we write the hash as a decimal number, the hash *mod* 10 would be the last digit). If we have 10 nodes, numbered 0 to 9, that seems like an easy way of assigning each key to a node.

The problem with the *mod N* approach is that if the number of nodes *N* changes, most of the keys will need to be moved from one node to another. For example, say *hash*(*key*) = 123456. If you initially have 10 nodes, that key starts out on node 6 (because 123456 *mod* 10 = 6). When you grow to 11 nodes, the key needs to move to node 3 (123456 *mod* 11 = 3), and when you grow to 12 nodes, it needs to move to node 0 (123456 *mod* 12 = 0). Such frequent moves make rebalancing excessively expensive.

We need an approach that doesn't move data around more than necessary.

### Fixed number of partitions

Fortunately, there is a fairly simple solution: create many more partitions than there are nodes, and assign several partitions to each node. For example, a database running on a cluster of 10 nodes may be split into 1,000 partitions from the outset so that approximately 100 partitions are assigned to each node.

Now, if a node is added to the cluster, the new node can *steal* a few partitions from every existing node until partitions are fairly distributed once again. This process is illustrated in Figure 6-6. If a node is removed from the cluster, the same happens in reverse.

Only entire partitions are moved between nodes. The number of partitions does not change, nor does the assignment of keys to partitions. The only thing that changes is the assignment of partitions to nodes. This change of assignment is not immediate—it takes some time to transfer a large amount of data over the network—so the old assignment of partitions is used for any reads and writes that happen while the transfer is in progress.

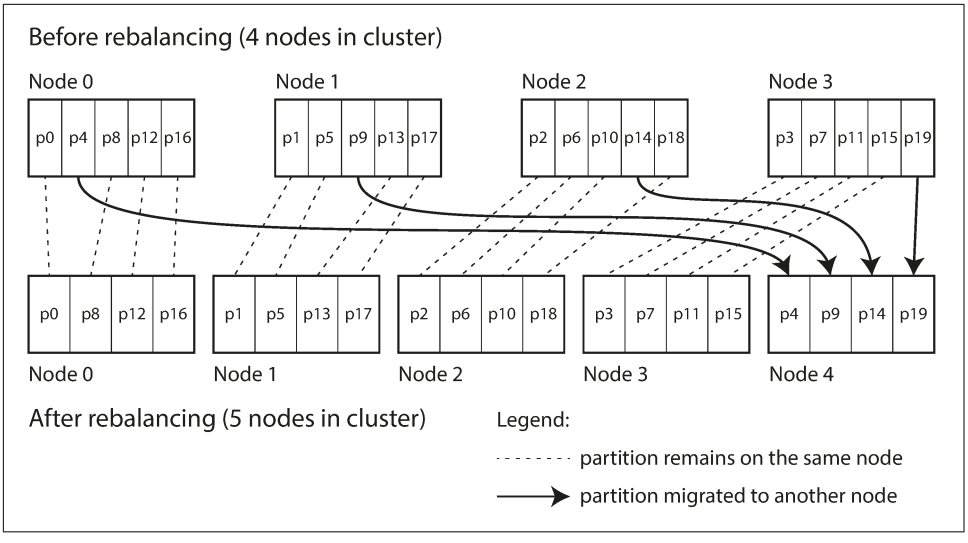



*Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.*

In principle, you can even account for mismatched hardware in your cluster: by assigning more partitions to nodes that are more powerful, you can force those nodes to take a greater share of the load.

This approach to rebalancing is used in Riak [15], Elasticsearch [24], Couchbase [10], and Voldemort [25].

In this configuration, the number of partitions is usually fixed when the database is first set up and not changed afterward. Although in principle it's possible to split and merge partitions (see the next section), a fixed number of partitions is operationally simpler, and so many fixed-partition databases choose not to implement partition splitting. Thus, the number of partitions configured at the outset is the maximum number of nodes you can have, so you need to choose it high enough to accommodate future growth. However, each partition also has management overhead, so it's counterproductive to choose too high a number.

Choosing the right number of partitions is difficult if the total size of the dataset is highly variable (for example, if it starts small but may grow much larger over time). Since each partition contains a fixed fraction of the total data, the size of each partition grows proportionally to the total amount of data in the cluster. If partitions are very large, rebalancing and recovery from node failures become expensive. But if partitions are too small, they incur too much overhead. The best performance is achieved when the size of partitions is "just right," neither too big nor too small, which can be hard to achieve if the number of partitions is fixed but the dataset size varies.

### Dynamic partitioning

For databases that use key range partitioning (see "Partitioning by Key Range" on page 202), a fixed number of partitions with fixed boundaries would be very inconvenient: if you got the boundaries wrong, you could end up with all of the data in one partition and all of the other partitions empty. Reconfiguring the partition boundaries manually would be very tedious.

For that reason, key range–partitioned databases such as HBase and RethinkDB create partitions dynamically. When a partition grows to exceed a configured size (on HBase, the default is 10 GB), it is split into two partitions so that approximately half of the data ends up on each side of the split [26]. Conversely, if lots of data is deleted and a partition shrinks below some threshold, it can be merged with an adjacent partition. This process is similar to what happens at the top level of a B-tree (see "B-Trees" on page 79).

Each partition is assigned to one node, and each node can handle multiple partitions, like in the case of a fixed number of partitions. After a large partition has been split, one of its two halves can be transferred to another node in order to balance the load. In the case of HBase, the transfer of partition files happens through HDFS, the underlying distributed filesystem [3].

An advantage of dynamic partitioning is that the number of partitions adapts to the total data volume. If there is only a small amount of data, a small number of partitions is sufficient, so overheads are small; if there is a huge amount of data, the size of each individual partition is limited to a configurable maximum [23].

However, a caveat is that an empty database starts off with a single partition, since there is no *a priori* information about where to draw the partition boundaries. While the dataset is small—until it hits the point at which the first partition is split—all writes have to be processed by a single node while the other nodes sit idle. To mitigate this issue, HBase and MongoDB allow an initial set of partitions to be configured on an empty database (this is called *pre-splitting*). In the case of key-range partitioning, pre-splitting requires that you already know what the key distribution is going to look like [4, 26].

Dynamic partitioning is not only suitable for key range–partitioned data, but can equally well be used with hash-partitioned data. MongoDB since version 2.4 supports both key-range and hash partitioning, and it splits partitions dynamically in either case.

### Partitioning proportionally to nodes

With dynamic partitioning, the number of partitions is proportional to the size of the dataset, since the splitting and merging processes keep the size of each partition between some fixed minimum and maximum. On the other hand, with a fixed num-

ber of partitions, the size of each partition is proportional to the size of the dataset. In both of these cases, the number of partitions is independent of the number of nodes.

A third option, used by Cassandra and Ketama, is to make the number of partitions proportional to the number of nodes—in other words, to have a fixed number of partitions *per node* [23, 27, 28]. In this case, the size of each partition grows proportionally to the dataset size while the number of nodes remains unchanged, but when you increase the number of nodes, the partitions become smaller again. Since a larger data volume generally requires a larger number of nodes to store, this approach also keeps the size of each partition fairly stable.

When a new node joins the cluster, it randomly chooses a fixed number of existing partitions to split, and then takes ownership of one half of each of those split partitions while leaving the other half of each partition in place. The randomization can produce unfair splits, but when averaged over a larger number of partitions (in Cassandra, 256 partitions per node by default), the new node ends up taking a fair share of the load from the existing nodes. Cassandra 3.0 introduced an alternative rebalancing algorithm that avoids unfair splits [29].

Picking partition boundaries randomly requires that hash-based partitioning is used (so the boundaries can be picked from the range of numbers produced by the hash function). Indeed, this approach corresponds most closely to the original definition of consistent hashing [7] (see "Consistent Hashing" on page 204). Newer hash functions can achieve a similar effect with lower metadata overhead [8].

## Operations: Automatic or Manual Rebalancing

There is one important question with regard to rebalancing that we have glossed over: does the rebalancing happen automatically or manually?

There is a gradient between fully automatic rebalancing (the system decides automatically when to move partitions from one node to another, without any administrator interaction) and fully manual (the assignment of partitions to nodes is explicitly configured by an administrator, and only changes when the administrator explicitly reconfigures it). For example, Couchbase, Riak, and Voldemort generate a suggested partition assignment automatically, but require an administrator to commit it before it takes effect.

Fully automated rebalancing can be convenient, because there is less operational work to do for normal maintenance. However, it can be unpredictable. Rebalancing is an expensive operation, because it requires rerouting requests and moving a large amount of data from one node to another. If it is not done carefully, this process can overload the network or the nodes and harm the performance of other requests while the rebalancing is in progress.

Such automation can be dangerous in combination with automatic failure detection. For example, say one node is overloaded and is temporarily slow to respond to requests. The other nodes conclude that the overloaded node is dead, and automatically rebalance the cluster to move load away from it. This puts additional load on the overloaded node, other nodes, and the network—making the situation worse and potentially causing a cascading failure.

For that reason, it can be a good thing to have a human in the loop for rebalancing. It's slower than a fully automatic process, but it can help prevent operational surprises.

## Request Routing

We have now partitioned our dataset across multiple nodes running on multiple machines. But there remains an open question: when a client wants to make a request, how does it know which node to connect to? As partitions are rebalanced, the assignment of partitions to nodes changes. Somebody needs to stay on top of those changes in order to answer the question: if I want to read or write the key "foo", which IP address and port number do I need to connect to?

This is an instance of a more general problem called *service discovery*, which isn't limited to just databases. Any piece of software that is accessible over a network has this problem, especially if it is aiming for high availability (running in a redundant configuration on multiple machines). Many companies have written their own in-house service discovery tools, and many of these have been released as open source [30].

On a high level, there are a few different approaches to this problem (illustrated in Figure 6-7):

1. Allow clients to contact any node (e.g., via a round-robin load balancer). If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client.
2. Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer.
3. Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node, without any intermediary.

In all cases, the key problem is: how does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?

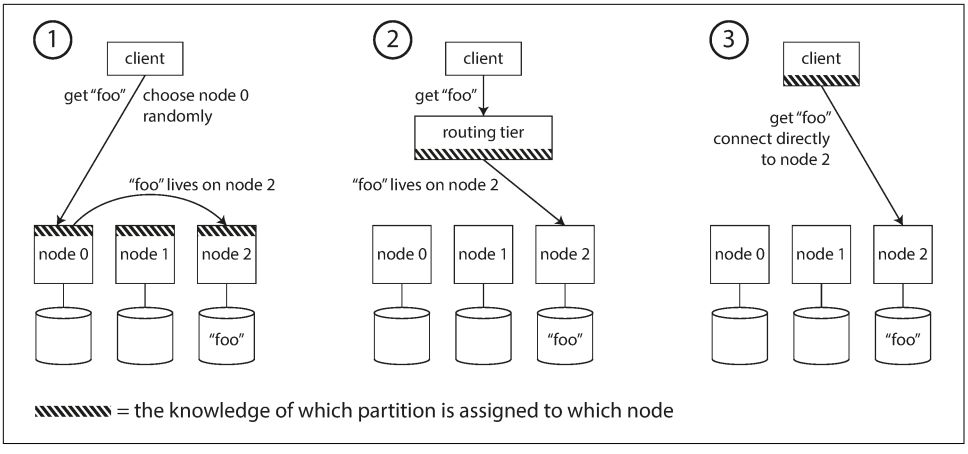
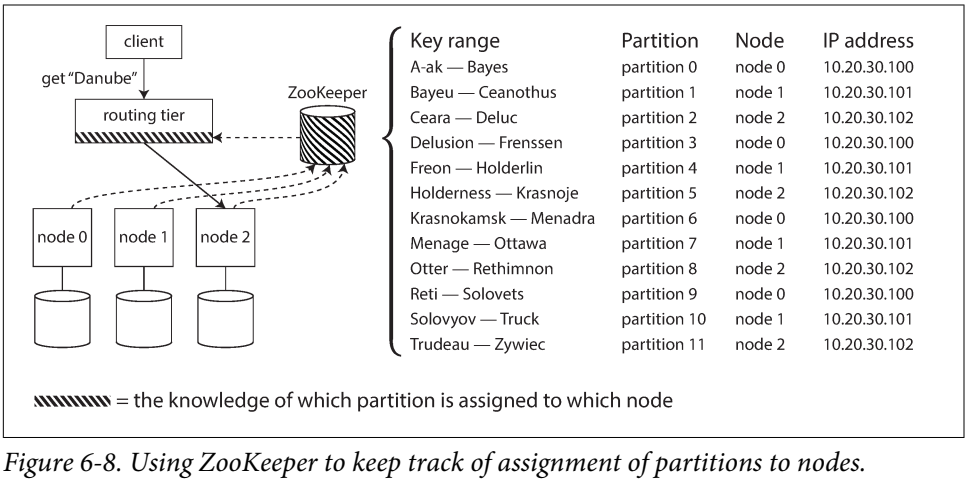



*Figure 6-7. Three different ways of routing a request to the right node.*

This is a challenging problem, because it is important that all participants agree—otherwise requests would be sent to the wrong nodes and not handled correctly. There are protocols for achieving consensus in a distributed system, but they are hard to implement correctly (see Chapter 9).

Many distributed data systems rely on a separate coordination service such as ZooKeeper to keep track of this cluster metadata, as illustrated in Figure 6-8. Each node registers itself in ZooKeeper, and ZooKeeper maintains the authoritative mapping of partitions to nodes. Other actors, such as the routing tier or the partitioning-aware client, can subscribe to this information in ZooKeeper. Whenever a partition changes ownership, or a node is added or removed, ZooKeeper notifies the routing tier so that it can keep its routing information up to date.




| Key range | Partition | Node | IP address |
|---|---|---|---|
| A-ak — Bayes | partition 0 | node 0 | 10.20.30.100 |
| Bayeu — Ceanothus | partition 1 | node 1 | 10.20.30.101 |
| Ceara — Deluc | partition 2 | node 2 | 10.20.30.102 |
| Delusion — Frenssen | partition 3 | node 0 | 10.20.30.100 |
| Freon — Holderlin | partition 4 | node 1 | 10.20.30.101 |
| Holderness — Krasnoje | partition 5 | node 2 | 10.20.30.102 |
| Krasnokamsk — Menadra | partition 6 | node 0 | 10.20.30.100 |
| Menage — Ottawa | partition 7 | node 1 | 10.20.30.101 |
| Otter — Rethimnon | partition 8 | node 2 | 10.20.30.102 |
| Reti — Solovets | partition 9 | node 0 | 10.20.30.100 |
| Solovyov — Truck | partition 10 | node 1 | 10.20.30.101 |
| Trudeau — Zywiec | partition 11 | node 2 | 10.20.30.102 |

*Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.*

For example, LinkedIn's Espresso uses Helix [31] for cluster management (which in turn relies on ZooKeeper), implementing a routing tier as shown in Figure 6-8. HBase, SolrCloud, and Kafka also use ZooKeeper to track partition assignment. MongoDB has a similar architecture, but it relies on its own *config server* implementation and *mongos* daemons as the routing tier.

Cassandra and Riak take a different approach: they use a *gossip protocol* among the nodes to disseminate any changes in cluster state. Requests can be sent to any node, and that node forwards them to the appropriate node for the requested partition (approach 1 in Figure 6-7). This model puts more complexity in the database nodes but avoids the dependency on an external coordination service such as ZooKeeper.

Couchbase does not rebalance automatically, which simplifies the design. Normally it is configured with a routing tier called *moxi*, which learns about routing changes from the cluster nodes [32].

When using a routing tier or when sending requests to a random node, clients still need to find the IP addresses to connect to. These are not as fast-changing as the assignment of partitions to nodes, so it is often sufficient to use DNS for this purpose.

## Parallel Query Execution

So far we have focused on very simple queries that read or write a single key (plus scatter/gather queries in the case of document-partitioned secondary indexes). This is about the level of access supported by most NoSQL distributed datastores.

However, *massively parallel processing* (MPP) relational database products, often used for analytics, are much more sophisticated in the types of queries they support. A typical data warehouse query contains several join, filtering, grouping, and aggregation operations. The MPP query optimizer breaks this complex query into a number of execution stages and partitions, many of which can be executed in parallel on different nodes of the database cluster. Queries that involve scanning over large parts of the dataset particularly benefit from such parallel execution.

Fast parallel execution of data warehouse queries is a specialized topic, and given the business importance of analytics, it receives a lot of commercial interest. We will discuss some techniques for parallel query execution in Chapter 10. For a more detailed overview of techniques used in parallel databases, please see the references [1, 33].

# Summary

In this chapter we explored different ways of partitioning a large dataset into smaller subsets. Partitioning is necessary when you have so much data that storing and processing it on a single machine is no longer feasible.

The goal of partitioning is to spread the data and query load evenly across multiple machines, avoiding hot spots (nodes with disproportionately high load). This requires choosing a partitioning scheme that is appropriate to your data, and rebalancing the partitions when nodes are added to or removed from the cluster.

We discussed two main approaches to partitioning:

- *Key range partitioning*, where keys are sorted, and a partition owns all the keys from some minimum up to some maximum. Sorting has the advantage that efficient range queries are possible, but there is a risk of hot spots if the application often accesses keys that are close together in the sorted order.

  In this approach, partitions are typically rebalanced dynamically by splitting the range into two subranges when a partition gets too big.
- *Hash partitioning*, where a hash function is applied to each key, and a partition owns a range of hashes. This method destroys the ordering of keys, making range queries inefficient, but may distribute load more evenly.

  When partitioning by hash, it is common to create a fixed number of partitions in advance, to assign several partitions to each node, and to move entire partitions from one node to another when nodes are added or removed. Dynamic partitioning can also be used.

Hybrid approaches are also possible, for example with a compound key: using one part of the key to identify the partition and another part for the sort order.

We also discussed the interaction between partitioning and secondary indexes. A secondary index also needs to be partitioned, and there are two methods:

- *Document-partitioned indexes* (local indexes), where the secondary indexes are stored in the same partition as the primary key and value. This means that only a single partition needs to be updated on write, but a read of the secondary index requires a scatter/gather across all partitions.
- *Term-partitioned indexes* (global indexes), where the secondary indexes are partitioned separately, using the indexed values. An entry in the secondary index may include records from all partitions of the primary key. When a document is written, several partitions of the secondary index need to be updated; however, a read can be served from a single partition.

Finally, we discussed techniques for routing queries to the appropriate partition, which range from simple partition-aware load balancing to sophisticated parallel query execution engines.

By design, every partition operates mostly independently—that's what allows a partitioned database to scale to multiple machines. However, operations that need to write

to several partitions can be difficult to reason about: for example, what happens if the write to one partition succeeds, but another fails? We will address that question in the following chapters.

## References


[1] David J. DeWitt and Jim N. Gray: "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM*, volume 35, number 6, pages 85–98, June 1992. doi:10.1145/129888.129894

[2] Lars George: "HBase vs. BigTable Comparison," *larsgeorge.com*, November 2009.

[3] "The Apache HBase Reference Guide," Apache Software Foundation, *hbase.apache.org*, 2014.

[4] MongoDB, Inc.: "New Hash-Based Sharding Feature in MongoDB 2.4," *blog.mongodb.org*, April 10, 2013.

[5] Ikai Lan: "App Engine Datastore Tip: Monotonically Increasing Values Are Bad," *ikaisays.com*, January 25, 2011.

[6] Martin Kleppmann: "Java's hashCode Is Not Safe for Distributed Systems," *martin.kleppmann.com*, June 18, 2012.

[7] David Karger, Eric Lehman, Tom Leighton, et al.: "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," at *29th Annual ACM Symposium on Theory of Computing* (STOC), pages 654–663, 1997. doi:10.1145/258533.258660

[8] John Lamping and Eric Veach: "A Fast, Minimal Memory, Consistent Hash Algorithm," *arxiv.org*, June 2014.

[9] Eric Redmond: "A Little Riak Book," Version 1.4.0, Basho Technologies, September 2013.

[10] "Couchbase 2.5 Administrator Guide," Couchbase, Inc., 2014.

[11] Avinash Lakshman and Prashant Malik: "Cassandra – A Decentralized Structured Storage System," at *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (LADIS), October 2009.

[12] Jonathan Ellis: "Facebook's Cassandra Paper, Annotated and Compared to Apache Cassandra 2.0," *docs.datastax.com*, September 12, 2013.

[13] "Introduction to Cassandra Query Language," DataStax, Inc., 2014.

[14] Samuel Axon: "3% of Twitter's Servers Dedicated to Justin Bieber," *mashable.com*, September 7, 2010.

[15] "Riak KV Docs," *docs.riak.com*.

[16] Richard Low: "The Sweet Spot for Cassandra Secondary Indexing," *wentnet.com*, October 21, 2013.

[17] Zachary Tong: "Customizing Your Document Routing," *elastic.co*, June 3, 2013.

[18] "Apache Solr Reference Guide," Apache Software Foundation, 2014.

[19] Andrew Pavlo: "H-Store Frequently Asked Questions," *hstore.cs.brown.edu*, October 2013.

[20] "Amazon DynamoDB Developer Guide," Amazon Web Services, Inc., 2014.

[21] Rusty Klophaus: "Difference Between 2I and Search," email to *riak-users* mailing list, *lists.basho.com*, October 25, 2011.

[22] Donald K. Burleson: "Object Partitioning in Oracle," *dba-oracle.com*, November 8, 2000.

[23] Eric Evans: "Rethinking Topology in Cassandra," at *ApacheCon Europe*, November 2012.

[24] Rafał Kuć: "Reroute API Explained," *elasticsearchserverbook.com*, September 30, 2013.

[25] "Project Voldemort Documentation," *project-voldemort.com*.

[26] Enis Soztutar: "Apache HBase Region Splitting and Merging," *hortonworks.com*, February 1, 2013.

[27] Brandon Williams: "Virtual Nodes in Cassandra 1.2," *datastax.com*, December 4, 2012.

[28] Richard Jones: "libketama: Consistent Hashing Library for Memcached Clients," *metabrew.com*, April 10, 2007.

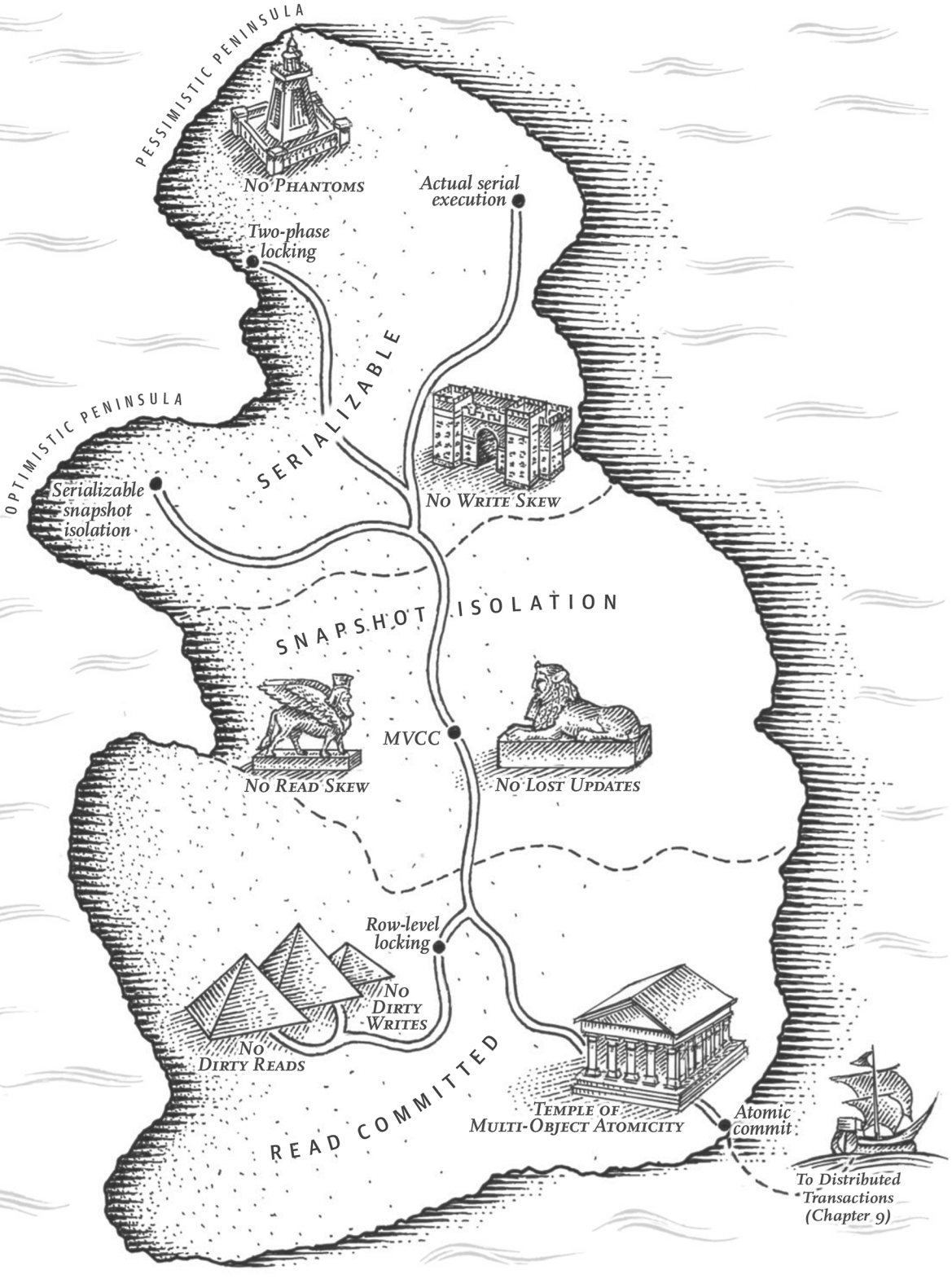
[29] Branimir Lambov: "New Token Allocation Algorithm in Cassandra 3.0," *datastax.com*, January 28, 2016.

[30] Jason Wilder: "Open-Source Service Discovery," *jasonwilder.com*, February 2014.

[31] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, et al.: "Untangling Cluster Management with Helix," at *ACM Symposium on Cloud Computing* (SoCC), October 2012. doi:10.1145/2391229.2391248

[32] "Moxi 1.8 Manual," Couchbase, Inc., 2014.

[33] Shivnath Babu and Herodotos Herodotou: "Massively Parallel Databases and MapReduce Systems," *Foundations and Trends in Databases*, volume 5, number 1, pages 1–104, November 2013. doi:10.1561/1900000036



# CHAPTER 7
# 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)

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.

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*.

This chapter applies to both single-node and distributed databases; in Chapter 8 we will focus the discussion on the particular challenges that arise only in distributed systems.

# The Slippery Concept of 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 [1, 2, 3]. Although some implementation details have changed, the general idea has remained virtually the same for 40 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 2), and by including replication (Chapter 5) and partitioning (Chapter 6) by default. Transactions were the main casualty of this movement: many of this new generation of databases abandoned transactions entirely, or redefined the word to describe a much weaker set of guarantees than had previously been understood [4].

With the hype around this new crop of distributed databases, there emerged a popular belief that transactions were the antithesis of scalability, and that any large-scale system would have to abandon transactions in order to maintain good performance and high availability [5, 6]. On the other hand, transactional guarantees are sometimes presented by database vendors as an essential requirement for "serious applications" with "valuable data." Both viewpoints are pure hyperbole.

The truth is not that simple: 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 [7] 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* [8]. 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* [9]. 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 comput-

ing. 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" on page 225).

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 5 we discussed *replica consistency* and the issue of *eventual consistency* that arises in asynchronously replicated systems (see "Problems with Replication Lag" on page 161).
- *Consistent hashing* is an approach to partitioning that some systems use for rebalancing (see "Consistent Hashing" on page 204).
- In the CAP theorem (see Chapter 9), the word *consistency* is used to mean *linearizability* (see "Linearizability" on page 324).
- 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 four 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.

However, this idea of consistency depends on the application's notion of invariants, and it's the application's responsibility to define its transactions correctly so that they preserve consistency. This is not something that the database can guarantee: if you write bad data that violates your invariants, the database can't stop you. (Some specific kinds of invariants can be checked by the database, for example using foreign key constraints or uniqueness constraints. However, in general, the application defines what data is valid or invalid—the database only stores it.)

Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application. The application may rely on the database's atomicity and isolation properties in order to achieve consistency, but it's not up to the database alone. Thus, the letter C doesn't really belong in ACID.[i]

### 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 7-1 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 7-1 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.

*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 [10].



i. Joe Hellerstein has remarked that the C in ACID was "tossed in to make the acronym work" in Härder and Reuter's paper [7], and that it wasn't considered important at the time.

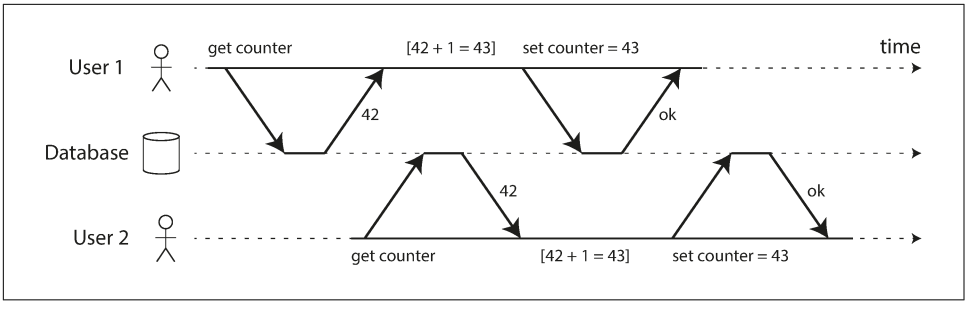



*Figure 7-1. A race condition between two clients concurrently incrementing a counter.*

However, in practice, serializable isolation is rarely used, because it carries a performance penalty. Some popular databases, such as Oracle 11g, don't even implement it. In Oracle there is an isolation level called "serializable," but it actually implements something called *snapshot isolation*, which is a weaker guarantee than serializability [8, 11]. We will explore snapshot isolation and other forms of isolation in "Weak Isolation Levels" on page 233.

### 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. It usually also involves a write-ahead log or similar (see "Making B-trees reliable" on page 82), which allows recovery in the event that the data structures on disk are corrupted. 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.

As discussed in "Reliability" on page 6, 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.

## Replication and Durability

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" on page 6), losing any data that is only in memory. Writing to disk is therefore still relevant for in-memory databases.
- In an asynchronously replicated system, recent writes may be lost when the leader becomes unavailable (see "Handling Node Outages" on page 156).
- 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 [12]. Disk firmware can have bugs, just like any other kind of software [13, 14].
- 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 [15, 16].
- Data on disk can gradually become corrupted without this being detected [17]. 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 [18]. 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 [19].

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 7-2 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 unread_flag = true
```

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). 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 7-2, 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.[ii] 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.

ii. Arguably, an incorrect counter in an email application is not a particularly critical problem. Alternatively, think of a customer account balance instead of an unread counter, and a payment transaction instead of an email.

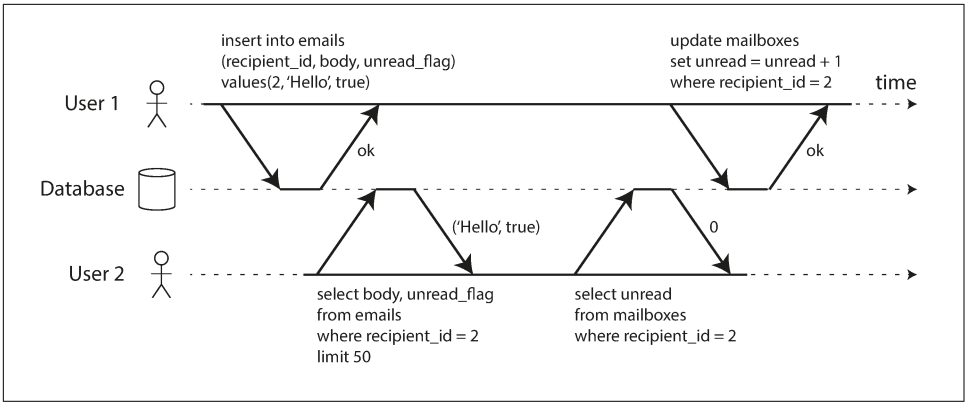



*Figure 7-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a "dirty read").*

Figure 7-3 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.

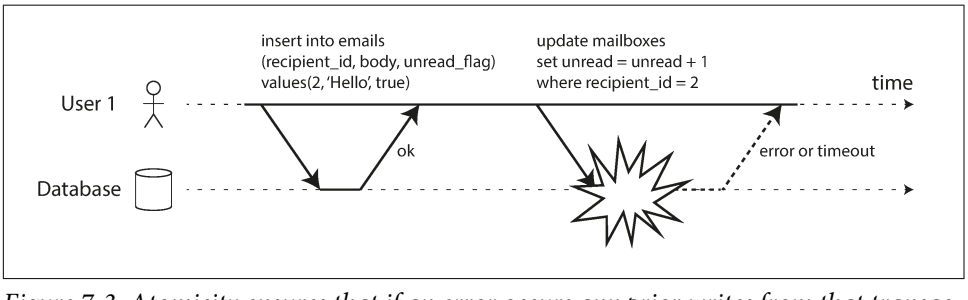



*Figure 7-3. Atomicity ensures that if an error occurs any prior writes from that transaction are undone, to avoid an inconsistent state.*

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.[iii]

iii. This is not ideal. If the TCP connection is interrupted, the transaction must be aborted. If the interruption happens after the client has requested a commit but before the server acknowledges that the commit happened, the client doesn't know whether the transaction was committed or not. To solve this issue, a transaction manager can group operations by a unique transaction identifier that is not bound to a particular TCP connection. We will return to this topic in "The End-to-End Argument for Databases" on page 516.

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" on page 82), 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,[iv] such as an increment operation, which removes the need for a read-modify-write cycle like that in Figure 7-1. Similarly popular is a compare-and-set operation, which allows a write to happen only if the value has not been concurrently changed by someone else (see "Compare-and-set" on page 245).

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" on page 242). However, they are not transactions in the usual sense of the word. Compare-and-set and other single-object operations have been dubbed "lightweight transactions" or even "ACID" for marketing purposes [20, 21, 22], but that terminology is misleading. A transaction is usually understood as a mechanism for grouping multiple operations on multiple objects into one unit of execution.

iv. 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 *isolated* or *serializable* increment. But that's getting nitpicky.

### The need for multi-object transactions

Many distributed datastores have abandoned multi-object transactions because they are difficult to implement across partitions, and they can get in the way in some scenarios where very high availability or performance is required. However, there is nothing that fundamentally prevents transactions in a distributed database, and we will discuss implementations of distributed transactions in Chapter 9.

But 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 "Relational Versus Document Databases Today" on page 38). When denormalized information needs to be updated, like in the example of Figure 7-2, 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.

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" on page 233, and explore alternative approaches in Chapter 12.

### 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" on page 177) 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 failed while the server tried to acknowledge the successful commit to the client (so the client thinks it failed), then retrying the transaction causes it to be performed twice—unless you have an additional application-level deduplication mechanism in place.
- If the error is due to overload, retrying the transaction will make the problem worse, not better. To avoid such feedback cycles, you can limit the number of retries, use exponential backoff, and handle overload-related errors differently from other errors (if possible).
- 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 "Atomic Commit and Two-Phase Commit (2PC)" on page 354).
- If the client process fails while retrying, any data it was trying to write to the database is lost.

# Weak Isolation Levels

If two transactions don't touch the same data, 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 two transactions try to simultaneously 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 [8]. 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 [23].

Concurrency bugs caused by weak transaction isolation are not just a theoretical problem. They have caused substantial loss of money [24, 25], led to investigation by financial auditors [26], and caused customer data to be corrupted [27]. 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.

Rather than blindly relying on tools, we need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them. Then we can build applications that are reliable and correct, using the tools at our disposal.

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" on page

251). 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 [28, 29, 30].

## Read Committed

The most basic level of transaction isolation is *read committed*.[v] 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* [2].

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 7-4, 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.

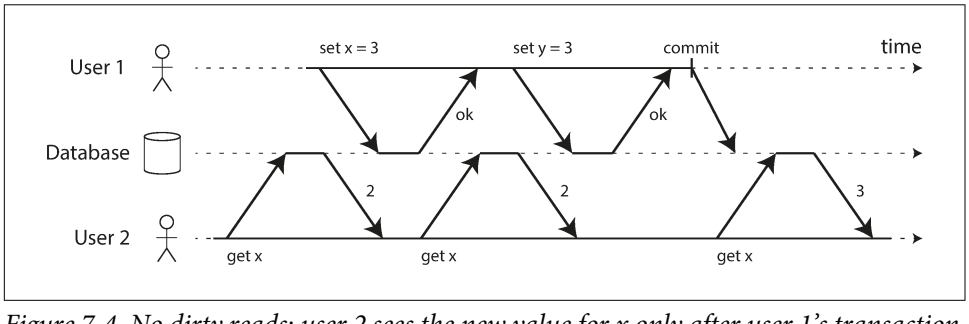



*Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.*

v. Some databases support an even weaker isolation level called *read uncommitted*. It prevents dirty writes, but does not prevent dirty reads.

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

- If a transaction needs to update several objects, a dirty read means that another transaction may see some of the updates but not others. For example, in Figure 7-2, 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 7-3). 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. Reasoning about the consequences quickly becomes mind-bending.

### No dirty writes

What happens if two transactions concurrently try to update the same object 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* [28]. 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 objects, dirty writes can lead to a bad outcome. For example, consider Figure 7-5, which illustrates a used car sales website on which two people, Alice and Bob, 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 7-5, the sale is awarded to Bob (because he performs the winning update to the `listings` table), but the invoice is sent to Alice (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 7-1. 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" on page 242 we will discuss how to make such counter increments safe.

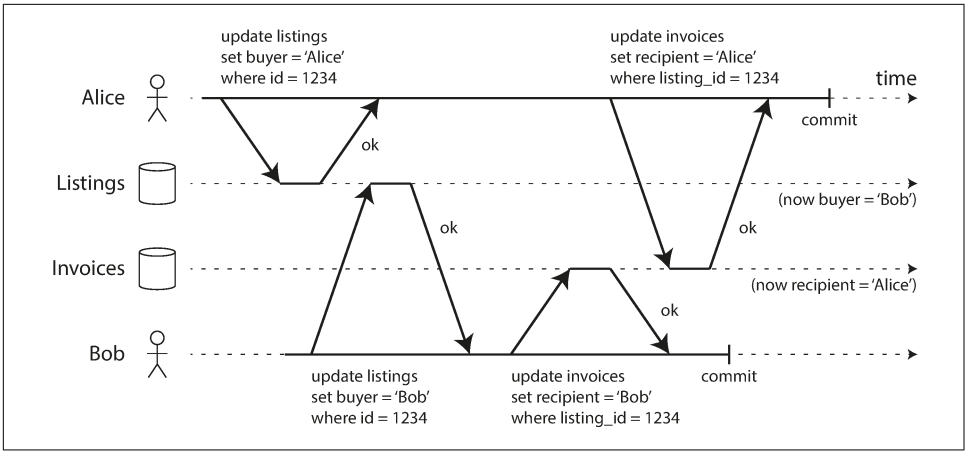



*Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.*

### Implementing read committed

Read committed is a very popular isolation level. It is the default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL, and many other databases [8].

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

How do we prevent dirty reads? One option would be to use the same lock, and to require any transaction that wants to read an object to briefly acquire the lock and then release it again immediately after reading. This would ensure that a read couldn't happen while an object 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.

For that reason, most databases[vi] prevent dirty reads using the approach illustrated in Figure 7-4: for every object that is written, the database remembers both the old committed value and the new value set by the transaction that currently holds the write lock. While the transaction is ongoing, any other transactions that read the object are simply given the old value. Only when the new value is committed do transactions switch over to reading the new value.

## Snapshot Isolation and Repeatable Read

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 7-6 illustrates a problem that can occur with read committed.

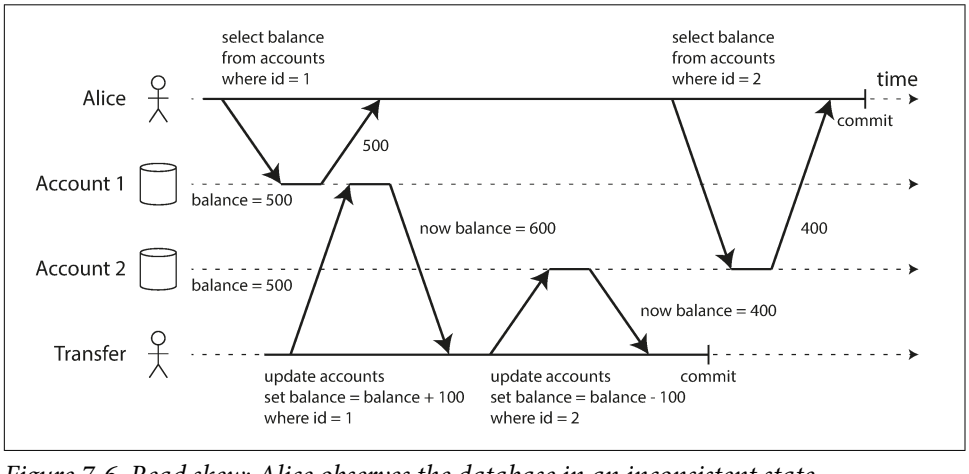



*Figure 7-6. Read skew: Alice observes the database in an inconsistent state.*

Say Alice 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

vi. At the time of writing, the only mainstream databases that use locks to prevent dirty reads are IBM DB2 and Microsoft SQL Server in the `read_committed_snapshot=off` configuration [23, 36].

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 Alice 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 Alice 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 Alice saw were indeed committed at the time when she read them.

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" on page 205), whereas here it means *timing anomaly*.

In Alice'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 "Transaction Processing or Analytics?" on page 90), 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* [28] 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: it is supported by PostgreSQL, MySQL with the InnoDB storage engine, Oracle, SQL Server, and others [23, 31, 32].

### Implementing snapshot isolation

Like read committed isolation, implementations of snapshot isolation typically use write locks to prevent dirty writes (see "Implementing read committed" on page 236), which means that a transaction that makes a write can block the progress of another transaction that writes to the same object. 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 7-4. The database must potentially keep several different committed versions of an object, 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 an object side by side, this technique is known as *multi-version concurrency control* (MVCC).

If a database only needed to provide read committed isolation, but not snapshot isolation, it would be sufficient to keep two versions of an object: the committed version and the overwritten-but-not-yet-committed version. However, storage engines that support snapshot isolation typically use MVCC for their read committed isolation level as well. A typical approach is that read committed uses a separate snapshot for each query, while snapshot isolation uses the same snapshot for an entire transaction.

Figure 7-7 illustrates how MVCC-based snapshot isolation is implemented in PostgreSQL [31] (other implementations are similar). When a transaction is started, it is given a unique, always-increasing[vii] transaction ID (`txid`). Whenever a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer.



vii. To be precise, transaction IDs are 32-bit integers, so they overflow after approximately 4 billion transactions. PostgreSQL's vacuum process performs cleanup which ensures that overflow does not affect the data.

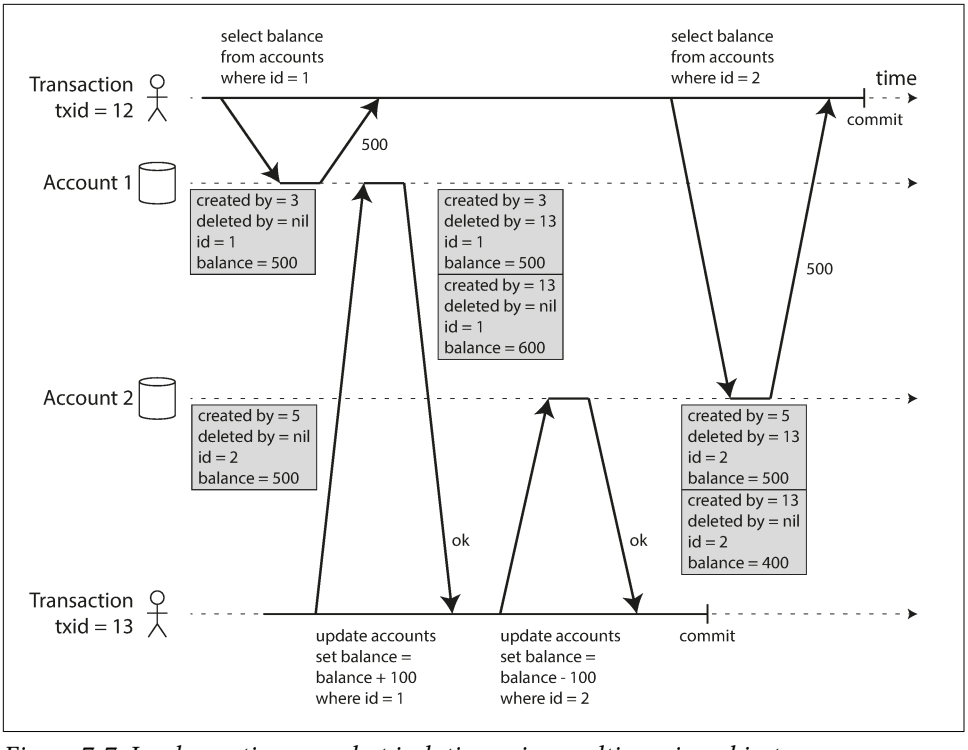



*Figure 7-7. Implementing snapshot isolation using multi-version objects.*

Each row in a table has a `created_by` field, containing the ID of the transaction that inserted this row into the table. Moreover, each row has a `deleted_by` field, which is initially empty. If a transaction deletes a row, the row isn't actually deleted from the database, but it is marked for deletion by setting the `deleted_by` field to the ID of the transaction that requested the deletion. At some later time, when it is certain that no transaction can any longer access the deleted data, a garbage collection process in the database removes any rows marked for deletion and frees their space.

An update is internally translated into a delete and a create. For example, in Figure 7-7, 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 created by transaction 13.

### Visibility rules for observing a consistent snapshot

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

the database can present a consistent snapshot of the database to the application. This works as follows:

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

These rules apply to both creation and deletion of objects. In Figure 7-7, 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 3, transaction 12 cannot see a deletion made by transaction 13), and the creation of the $400 balance is not yet visible (by the same rule).

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

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

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

### Indexes and snapshot isolation

How do indexes work in a multi-version database? One option is to have the index simply point to all versions of an object and require an index query to filter out any object versions that are not visible to the current transaction. When garbage collection removes old object versions that are no longer visible to any transaction, the corresponding index entries can also be removed.

In practice, many implementation details determine the performance of multi-version concurrency control. For example, PostgreSQL has optimizations for avoiding index updates if different versions of the same object can fit on the same page [31].

Another approach is used in CouchDB, Datomic, and LMDB. Although they also use B-trees (see "B-Trees" on page 79), they use an *append-only/copy-on-write* variant that does not overwrite pages of the tree when they are updated, but instead creates a new copy of each modified page. Parent pages, up to the root of the tree, are copied and updated to point to the new versions of their child pages. Any pages that are not affected by a write do not need to be copied, and remain immutable [33, 34, 35].

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

### Repeatable read and naming confusion

Snapshot isolation is a useful isolation level, especially for read-only transactions. However, many databases that implement it call it by different names. In Oracle it is called *serializable*, and in PostgreSQL and MySQL it is called *repeatable read* [23].

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 [2] and snapshot isolation hadn't yet been invented then. Instead, it defines repeatable read, which looks superficially similar to snapshot isolation. PostgreSQL and MySQL call their 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 [28]. Even though several databases implement repeatable read, there are big differences in the guarantees they actually provide, despite being ostensibly standardized [23]. There has been a formal definition of repeatable read in the research literature [29, 30], but most implementations don't satisfy that formal definition. And to top it off, IBM DB2 uses "repeatable read" to refer to serializability [8].

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" on page 235), 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 7-1 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.

### 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[viii]—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

viii. It is possible, albeit fairly complicated, to express the editing of a text document as a stream of atomic mutations. See "Automatic Conflict Resolution" on page 174 for some pointers.

applied. This technique is sometimes known as *cursor stability* [36, 37]. Another option is to simply force all atomic operations to be executed on a single thread.

Unfortunately, object-relational mapping frameworks make it easy to accidentally write code that performs unsafe read-modify-write cycles instead of using atomic operations provided by the database [38]. That's not a problem if you know what you are doing, but it is potentially 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 read 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 7-1.

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

```
BEGIN TRANSACTION;

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

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

COMMIT;
```

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.

### 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 [23]. Some authors [28, 30] 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.

### Compare-and-set

In databases that don't provide transactions, you sometimes find an atomic compare-and-set operation (previously mentioned in "Single-object writes" on page 230). The purpose of this operation is to avoid lost updates by allowing an update to happen only if the value has not changed since you last read it. If the current value does not match what you previously read, the update has no effect, and the read-modify-write cycle must be retried.

For example, to prevent two users concurrently updating the same wiki page, you might try something like this, expecting the update to occur only if the content of the page hasn't changed since the user started editing it:

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

If the content has changed and no longer matches `'old content'`, this update will have no effect, so you need to check whether the update took effect and retry if necessary. However, if the database allows the `WHERE` clause to read from an old snapshot, this statement may not prevent lost updates, because the condition may be true even though another concurrent write is occurring. Check whether your database's compare-and-set operation is safe before relying on it.

### Conflict resolution and replication

In replicated databases (see Chapter 5), preventing lost updates takes on another dimension: since they have copies of the data on multiple nodes, and the data can potentially be modified concurrently on different nodes, some additional steps need to be taken to prevent lost updates.

Locks and compare-and-set operations assume that there is a single up-to-date copy of the data. However, databases with multi-leader or leaderless replication usually allow several writes to happen concurrently and replicate them asynchronously, so they cannot guarantee that there is a single up-to-date copy of the data. Thus, techniques based on locks or compare-and-set do not apply in this context. (We will revisit this issue in more detail in "Linearizability" on page 324.)

Instead, as discussed in "Detecting Concurrent Writes" on page 184, a common approach in such replicated databases is to allow concurrent writes to create several conflicting versions of a value (also known as *siblings*), and to use application code or special data structures to resolve and merge these versions after the fact.

Atomic operations can work well in a replicated context, especially if they are commutative (i.e., you can apply them in a different order on different replicas, and still get the same result). For example, incrementing a counter or adding an element to a set are commutative operations. That is the idea behind Riak 2.0 datatypes, which prevent lost updates across replicas. When a value is concurrently updated by different clients, Riak automatically merges together the updates in such a way that no updates are lost [39].

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)" on page 186. 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 [40, 41].

Now imagine that Alice and Bob 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 7-8.

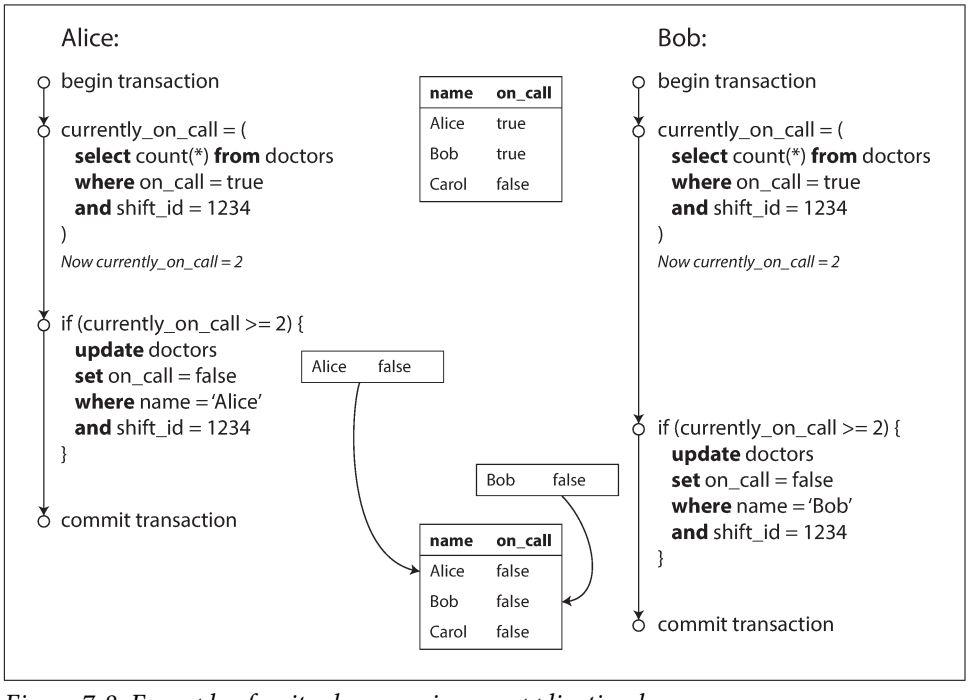



*Figure 7-8. Example of write skew causing an application bug.*

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. Alice updates her own record to take herself off call, and Bob 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* [28]. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects (Alice's and Bob'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 [23]. Automatically preventing write skew requires true serializable isolation (see "Serializability" on page 251).
- 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, depending on the database [42].
- 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 = 'Alice'
  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 [43]. 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 7-2).[ix]

*Example 7-2. A meeting room booking system tries to avoid double-booking (not safe under snapshot isolation)*

```
BEGIN TRANSACTION;

-- Check for any existing bookings that overlap with the period of noon-1pm
SELECT COUNT(*) FROM bookings
  WHERE room_id = 123 AND
    end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';

-- If the previous query returned zero:
INSERT INTO bookings
  (room_id, start_time, end_time, user_id)
  VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);

COMMIT;
```

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 7-1, 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.

ix. In PostgreSQL you can do this more elegantly using range types, but they are not widely supported in other databases.

*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 [44]. 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.

This effect, where a write in one transaction changes the result of a search query in another transaction, is called a *phantom* [3]. 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.

### 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 [11]. 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 [26], 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 [2]. All along, the answer from researchers has been simple: use *serializable* isolation!

Serializable isolation is usually regarded as the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, *serially*, without any concurrency. 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" on page 252)
- Two-phase locking (see "Two-Phase Locking (2PL)" on page 257), which for several decades was the only viable option
- Optimistic concurrency control techniques such as serializable snapshot isolation (see "Serializable Snapshot Isolation (SSI)" on page 261)

For now, we will discuss these techniques primarily in the context of single-node databases; in Chapter 9 we will examine how they can be generalized to transactions that involve multiple nodes in a distributed system.

## Actual Serial Execution

The simplest way of avoiding concurrency problems is to remove the concurrency entirely: to execute only one transaction at a time, in serial order, on a single thread. 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, database designers only fairly recently—around 2007—decided that a single-threaded loop for executing transactions was feasible [45]. 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" on page 88). 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 "Transaction Processing or Analytics?" on page 90). 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 [46, 47, 48]. 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 submit the entire transaction code to the database ahead of time, as a *stored procedure*. The differences between these approaches is illustrated in Figure 7-9. Provided that all data required by a transaction is in memory, the stored procedure can execute very fast, without waiting for any network or disk I/O.

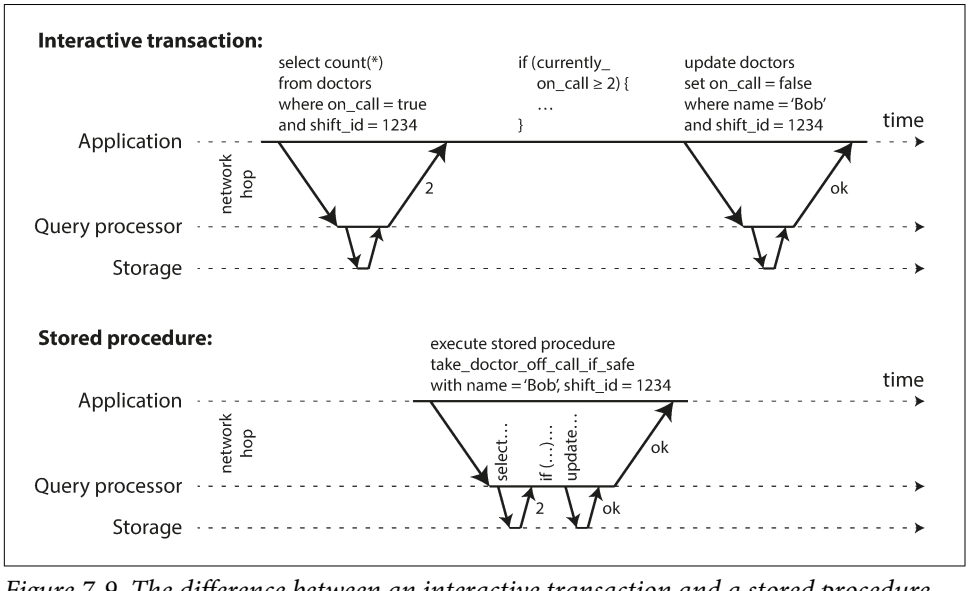



*Figure 7-9. The difference between an interactive transaction and a stored procedure (using the example transaction of Figure 7-8).*

### Pros and cons of stored procedures

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:

- Each database vendor has 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.

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, and Redis uses Lua.

With stored procedures and in-memory data, executing all transactions on a single thread becomes feasible. As they don't need to wait for I/O and they 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.

### Partitioning

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 potentially partition your data (see Chapter 6), which is supported in VoltDB. If you can find a way of partitioning your dataset so that each transaction only needs to read and write data within a single partition, then each partition can have its own transaction processing thread running independently from the others. In this case, you can give each CPU core its own partition, which allows your transaction throughput to scale linearly with the number of CPU cores [47].

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

Since cross-partition transactions have additional coordination overhead, they are vastly slower than single-partition transactions. VoltDB reports a throughput of about 1,000 cross-partition writes per second, which is orders of magnitude below its single-partition throughput and cannot be increased by adding more machines [49].

Whether transactions can be single-partition depends very much on the structure of the data used by the application. Simple key-value data can often be partitioned very easily, but data with multiple secondary indexes is likely to require a lot of cross-partition coordination (see "Partitioning and Secondary Indexes" on page 206).

### 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 limited to use cases 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.[x]
- Write throughput must be low enough to be handled on a single CPU core, or else transactions need to be partitioned without requiring cross-partition coordination.
- Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.

x. If a transaction needs to access data that's not in memory, the best solution may be to abort the transaction, asynchronously fetch the data into memory while continuing to process other transactions, and then restart the transaction when the data has been loaded. This approach is known as *anti-caching*, as previously mentioned in "Keeping everything in memory" on page 88.

## Two-Phase Locking (2PL)

For around 30 years, there was only one widely used algorithm for serializability in databases: *two-phase locking* (2PL).[xi]

### 2PL is not 2PC

Note that while two-phase *locking* (2PL) sounds very similar to two-phase *commit* (2PC), they are completely different things. We will discuss 2PC in Chapter 9.

We saw previously that locks are often used to prevent dirty writes (see "No dirty writes" on page 235): 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 7-4, 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 "Implementing snapshot isolation" on page 239), 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 [23, 36].



xi. Sometimes called *strong strict two-phase locking* (SS2PL) to distinguish it from other variants of 2PL.

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

- If a transaction 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 (while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released.

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 used by everybody 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.

Traditional relational databases don't limit the duration of a transaction, because they are designed for interactive applications that wait for human input. Consequently, when one transaction has to wait on another, there is no limit on how long it may have to wait. Even if you make sure that you keep all your transactions short, a

queue may form if several transactions want to access the same object, so a transaction may have to wait for several others to complete before it can do anything.

For this reason, databases running 2PL can have quite unstable latencies, and they can be very slow at high percentiles (see "Describing Performance" on page 13) 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. This instability is problematic when robust operation is required.

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" on page 250 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 7-2), 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* [3]. It works similarly to the shared/exclusive lock described earlier, but rather than belonging to a particular object (e.g., one row in a table), it belongs to all objects that match some search condition, such as:

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

A predicate lock restricts access as follows:

- If transaction A wants to read objects matching some condition, like in that `SELECT` query, it must acquire a shared-mode predicate lock on the conditions of the query. If another transaction B currently has an exclusive lock on any object matching those conditions, A must wait until B releases its lock before it is allowed to make its query.

- If transaction A wants to insert, update, or delete any object, it must first check whether either the old or the new value matches any existing predicate lock. If there is a matching predicate lock held by transaction B, then A must wait until B has committed or aborted before it can continue.

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

### Index-range locks

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 [41, 50].

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, 2018.

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?

Perhaps not: an algorithm called *serializable snapshot isolation* (SSI) is very promising. It provides full serializability, but has only a small performance penalty compared to snapshot isolation. SSI is fairly new: it was first described in 2008 [40] and is the subject of Michael Cahill's PhD thesis [51].

Today SSI is used both in single-node databases (the serializable isolation level in PostgreSQL since version 9.1 [41]) and distributed databases (FoundationDB uses a similar algorithm). As SSI is so young compared to other concurrency control mechanisms, it is still proving its performance in practice, but it has the possibility of being fast enough to become the new default in the future.

### 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 partition 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 trans-

action is aborted and has to be retried. Only transactions that executed serializably are allowed to commit.

Optimistic concurrency control is an old idea [52], and its advantages and disadvantages have been debated for a long time [53]. 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 page 237). This is the main difference compared to earlier optimistic concurrency control techniques. On top of snapshot isolation, SSI adds an algorithm for detecting serialization conflicts among 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" on page 246), 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 "Implementing snapshot isolation" on page 239). 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 7-10, transaction 43 sees Alice as having `on_call = true`, because transaction 42 (which modified Alice'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.

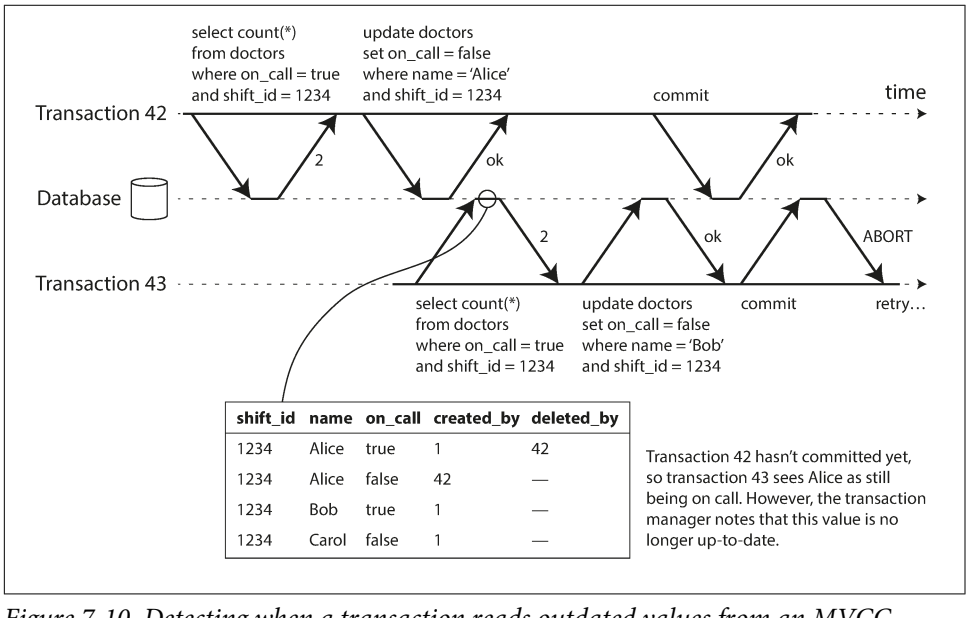



*Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.*

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 7-11.

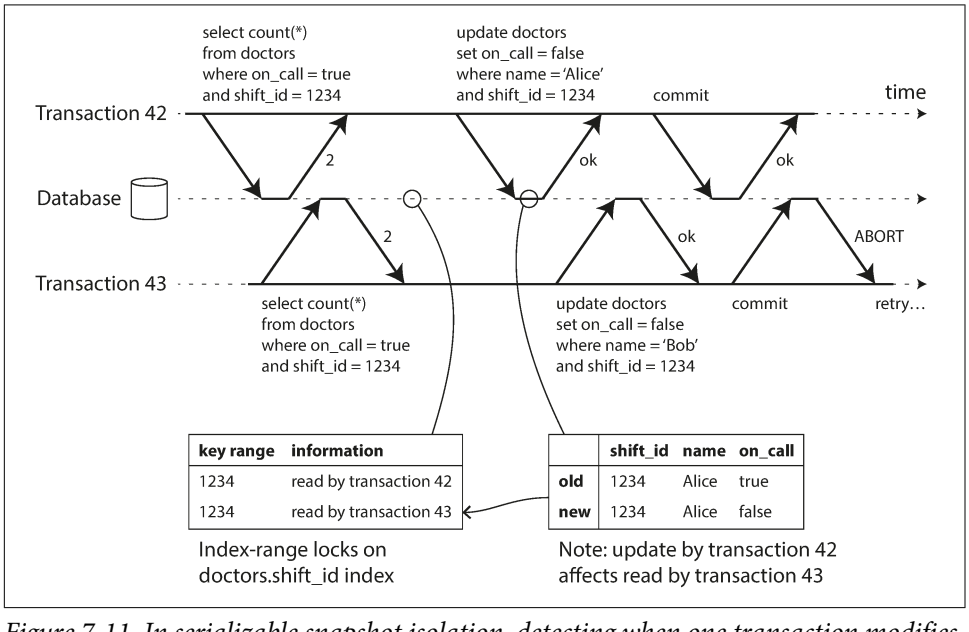



*Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.*

In the context of two-phase locking we discussed index-range locks (see "Index-range locks" on page 260), 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 7-11, 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 7-11, 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 [11, 41].

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: FoundationDB distributes the detection of serialization conflicts across multiple machines, allowing it to scale to very high throughput. Even though data may be partitioned across multiple machines, transactions can read and write data in multiple partitions while ensuring serializable isolation [54].

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 may be okay). However, SSI is probably less sensitive to slow transactions than two-phase locking or serial execution.

## 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:

*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, and the transaction throughput is low enough to process on a single CPU core, 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 performance characteristics.

*Serializable snapshot isolation (SSI)*
: A fairly 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.

The examples in this chapter used a relational data model. However, as discussed in "The need for multi-object transactions" on page 231, transactions are a valuable database feature, no matter which data model is used.

In this chapter, we explored ideas and algorithms mostly in the context of a database running on a single machine. Transactions in distributed databases open a new set of difficult challenges, which we'll discuss in the next two chapters.

## References


[1] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, et al.: "A History and Evaluation of System R," *Communications of the ACM*, volume 24, number 10, pages 632–646, October 1981. doi:10.1145/358769.358784

[2] 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

[3] 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, number 11, pages 624–633, November 1976.

[4] "ACID Transactions Are Incredibly Helpful," FoundationDB, LLC, 2013.

[5] John D. Cook: "ACID Versus BASE for Database Transactions," *johndcook.com*, July 6, 2009.

[6] Gavin Clarke: "NoSQL's CAP Theorem Busters: We Don't Drop ACID," *theregister.co.uk*, November 22, 2012.

[7] Theo Härder and Andreas Reuter: "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, volume 15, number 4, pages 287–317, December 1983. doi:10.1145/289.291

[8] Peter Bailis, Alan Fekete, Ali Ghodsi, et al.: "HAT, not CAP: Towards Highly Available Transactions," at *14th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2013.

[9] Armando Fox, Steven D. Gribble, Yatin Chawathe, et al.: "Cluster-Based Scalable Network Services," at *16th ACM Symposium on Operating Systems Principles* (SOSP), October 1997.

[10] 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 *research.microsoft.com*.

[11] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, et al.: "Making Snapshot Isolation Serializable," *ACM Transactions on Database Systems*, volume 30, number 2, pages 492–528, June 2005. doi:10.1145/1071610.1071615

[12] 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.

[13] Laurie Denness: "SSDs: A Gift and a Curse," *laur.ie*, June 2, 2015.

[14] Adam Surak: "When Solid State Drives Are Not That Solid," *blog.algolia.com*, June 15, 2015.

[15] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, et al.: "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.

[16] Chris Siebenmann: "Unix's File Durability Problem," *utcc.utoronto.ca*, April 14, 2016.

[17] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, et al.: "An Analysis of Data Corruption in the Storage Stack," at *6th USENIX Conference on File and Storage Technologies* (FAST), February 2008.

[18] 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.

[19] Don Allison: "SSD Storage – Ignorance of Technology Is No Excuse," *blog.korelogic.com*, March 24, 2015.

[20] Dave Scherer: "Those Are Not Transactions (Cassandra 2.0)," *blog.foundationdb.com*, September 6, 2013.

[21] Kyle Kingsbury: "Call Me Maybe: Cassandra," *aphyr.com*, September 24, 2013.

[22] "ACID Support in Aerospike," Aerospike, Inc., June 2014.

[23] Martin Kleppmann: "Hermitage: Testing the 'I' in ACID," *martin.kleppmann.com*, November 25, 2014.

[24] Tristan D'Agosta: "BTC Stolen from Poloniex," *bitcointalk.org*, March 4, 2014.

[25] bitcointhief2: "How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More!," *reddit.com*, February 2, 2014.

[26] 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.

[27] Michael Melanson: "Transactions: The Limits of Isolation," *michaelmelanson.net*, November 30, 2014.

[28] Hal Berenson, Philip A. Bernstein, Jim N. Gray, et al.: "A Critique of ANSI SQL Isolation Levels," at *ACM International Conference on Management of Data* (SIGMOD), May 1995.

[29] Atul Adya: "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," PhD Thesis, Massachusetts Institute of Technology, March 1999.

[30] Peter Bailis, Aaron Davidson, Alan Fekete, et al.: "Highly Available Transactions: Virtues and Limitations (Extended Version)," at *40th International Conference on Very Large Data Bases* (VLDB), September 2014.

[31] Bruce Momjian: "MVCC Unmasked," *momjian.us*, July 2014.

[32] Annamalai Gurusami: "Repeatable Read Isolation Level in InnoDB – How Consistent Read View Works," *blogs.oracle.com*, January 15, 2013.

[33] Nikita Prokopov: "Unofficial Guide to Datomic Internals," *tonsky.me*, May 6, 2014.

[34] Baron Schwartz: "Immutability, MVCC, and Garbage Collection," *xaprb.com*, December 28, 2013.

[35] J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6

[36] Rikdeb Mukherjee: "Isolation in DB2 (Repeatable Read, Read Stability, Cursor Stability, Uncommitted Read) with Examples," *mframes.blogspot.co.uk*, July 4, 2013.

[37] Steve Hilker: "Cursor Stability (CS) – IBM DB2 Community," *toadworld.com*, March 14, 2013.

[38] Nate Wiger: "An Atomic Rant," *nateware.com*, February 18, 2010.

[39] Joel Jacobson: "Riak 2.0: Data Types," *blog.joeljacobson.com*, March 23, 2014.

[40] 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

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

[42] Tony Andrews: "Enforcing Complex Constraints in Oracle," *tonyandrews.blogspot.co.uk*, October 15, 2004.

[43] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, et al.: "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

[44] Gary Fredericks: "Postgres Serializability Bug," *github.com*, September 2015.

[45] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al.: "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.

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

[47] Robert Kallman, Hideaki Kimura, Jonathan Natkins, et al.: "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System," *Proceedings of the VLDB Endowment*, volume 1, number 2, pages 1496–1499, August 2008.

[48] Rich Hickey: "The Architecture of Datomic," *infoq.com*, November 2, 2012.

[49] John Hugg: "Debunking Myths About the VoltDB In-Memory Database," *dzone.com*, May 28, 2014.

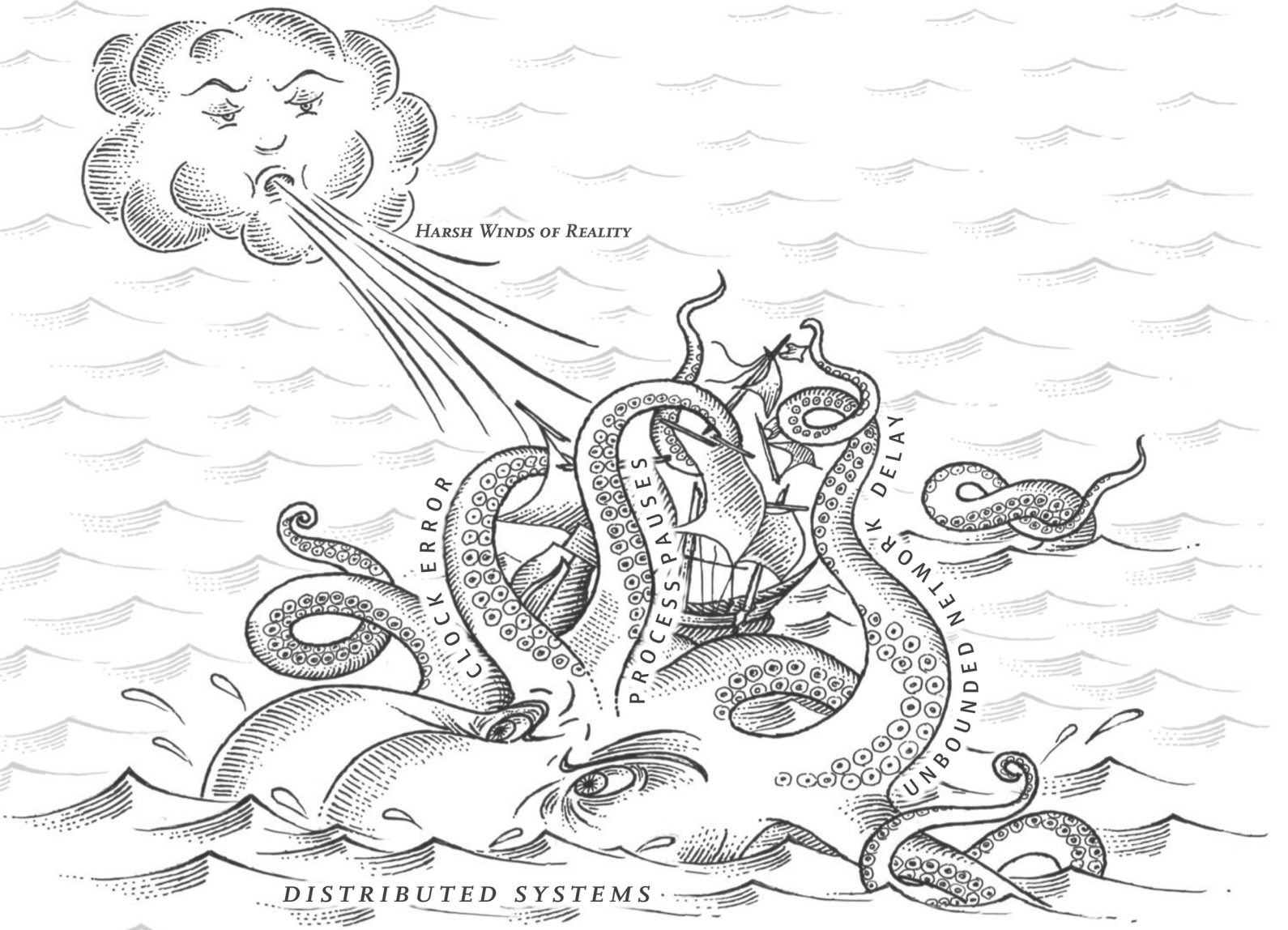
[50] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton: "Architecture of a Database System," *Foundations and Trends in Databases*, volume 1, number 2, pages 141–259, November 2007. doi:10.1561/1900000002

[51] Michael J. Cahill: "Serializable Isolation for Snapshot Databases," PhD Thesis, University of Sydney, July 2009.

[52] D. Z. Badal: "Correctness of Concurrency Control and Implications in Distributed Databases," at *3rd International IEEE Computer Software and Applications Conference* (COMPSAC), November 1979.

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

[54] Dave Rosenthal: "Databases at 14.4MHz," *blog.foundationdb.com*, December 10, 2014.



# CHAPTER 8
# The Trouble with Distributed Systems

*Hey I just met you*
*The network's laggy*
*But here's my data*
*So store it maybe*
—Kyle Kingsbury, *Carly Rae Jepsen and the Perils of Network Partitions* (2013)

A recurring theme in the last few chapters has been how systems handle things going wrong. For example, we discussed replica failover ("Handling Node Outages" on page 156), replication lag ("Problems with Replication Lag" on page 161), and concurrency control for transactions ("Weak Isolation Levels" on page 233). As we come to understand various edge cases that can occur in real systems, we get better at handling them.

However, even though we have talked a lot about faults, the last few chapters have still been too optimistic. The reality is even darker. We will now turn our pessimism to the maximum and assume that anything that *can* go wrong *will* go wrong.[i] (Experienced systems operators will tell you that is a reasonable assumption. If you ask nicely, they might tell you some frightening stories while nursing their scars of past battles.)

Working with distributed systems is fundamentally different from writing software on a single computer—and the main difference is that there are lots of new and exciting ways for things to go wrong [1, 2]. In this chapter, we will get a taste of the problems that arise in practice, and an understanding of the things we can and cannot rely on.

i. With one exception: we will assume that faults are *non-Byzantine* (see "Byzantine Faults" on page 304).

In the end, our task as engineers is to build systems that do their job (i.e., meet the guarantees that users are expecting), in spite of everything going wrong. In Chapter 9, we will look at some examples of algorithms that can provide such guarantees in a distributed system. But first, in this chapter, we must understand what challenges we are up against.

This chapter is a thoroughly pessimistic and depressing overview of things that may go wrong in a distributed system. We will look into problems with networks ("Unreliable Networks" on page 277); clocks and timing issues ("Unreliable Clocks" on page 287); and we'll discuss to what degree they are avoidable. The consequences of all these issues are disorienting, so we'll explore how to think about the state of a distributed system and how to reason about things that have happened ("Knowledge, Truth, and Lies" on page 300).

## Faults and Partial Failures

When you are writing a program on a single computer, it normally behaves in a fairly predictable way: either it works or it doesn't. Buggy software may give the appearance that the computer is sometimes "having a bad day" (a problem that is often fixed by a reboot), but that is mostly just a consequence of badly written software.

There is no fundamental reason why software on a single computer should be flaky: when the hardware is working correctly, the same operation always produces the same result (it is *deterministic*). If there is a hardware problem (e.g., memory corruption or a loose connector), the consequence is usually a total system failure (e.g., kernel panic, "blue screen of death," failure to start up). An individual computer with good software is usually either fully functional or entirely broken, but not something in between.

This is a deliberate choice in the design of computers: if an internal fault occurs, we prefer a computer to crash completely rather than returning a wrong result, because wrong results are difficult and confusing to deal with. Thus, computers hide the fuzzy physical reality on which they are implemented and present an idealized system model that operates with mathematical perfection. A CPU instruction always does the same thing; if you write some data to memory or disk, that data remains intact and doesn't get randomly corrupted. This design goal of always-correct computation goes all the way back to the very first digital computer [3].

When you are writing software that runs on several computers, connected by a network, the situation is fundamentally different. In distributed systems, we are no longer operating in an idealized system model—we have no choice but to confront the messy reality of the physical world. And in the physical world, a remarkably wide range of things can go wrong, as illustrated by this anecdote [4]:

> In my limited experience I've dealt with long-lived network partitions in a single data center (DC), PDU [power distribution unit] failures, switch failures, accidental power cycles of whole racks, whole-DC backbone failures, whole-DC power failures, and a hypoglycemic driver smashing his Ford pickup truck into a DC's HVAC [heating, ventilation, and air conditioning] system. And I'm not even an ops guy.
>
> —Coda Hale

In a distributed system, there may well be some parts of the system that are broken in some unpredictable way, even though other parts of the system are working fine. This is known as a *partial failure*. The difficulty is that partial failures are *nondeterministic*: if you try to do anything involving multiple nodes and the network, it may sometimes work and sometimes unpredictably fail. As we shall see, you may not even *know* whether something succeeded or not, as the time it takes for a message to travel across a network is also nondeterministic!

This nondeterminism and possibility of partial failures is what makes distributed systems hard to work with [5].

## Cloud Computing and Supercomputing

There is a spectrum of philosophies on how to build large-scale computing systems:

- At one end of the scale is the field of *high-performance computing* (HPC). Supercomputers with thousands of CPUs are typically used for computationally intensive scientific computing tasks, such as weather forecasting or molecular dynamics (simulating the movement of atoms and molecules).
- At the other extreme is *cloud computing*, which is not very well defined [6] but is often associated with multi-tenant datacenters, commodity computers connected with an IP network (often Ethernet), elastic/on-demand resource allocation, and metered billing.
- Traditional enterprise datacenters lie somewhere between these extremes.

With these philosophies come very different approaches to handling faults. In a supercomputer, a job typically checkpoints the state of its computation to durable storage from time to time. If one node fails, a common solution is to simply stop the entire cluster workload. After the faulty node is repaired, the computation is restarted from the last checkpoint [7, 8]. Thus, a supercomputer is more like a single-node computer than a distributed system: it deals with partial failure by letting it escalate into total failure—if any part of the system fails, just let everything crash (like a kernel panic on a single machine).

In this book we focus on systems for implementing internet services, which usually look very different from supercomputers:

- Many internet-related applications are *online*, in the sense that they need to be able to serve users with low latency at any time. Making the service unavailable—for example, stopping the cluster for repair—is not acceptable. In contrast, offline (batch) jobs like weather simulations can be stopped and restarted with fairly low impact.
- Supercomputers are typically built from specialized hardware, where each node is quite reliable, and nodes communicate through shared memory and remote direct memory access (RDMA). On the other hand, nodes in cloud services are built from commodity machines, which can provide equivalent performance at lower cost due to economies of scale, but also have higher failure rates.
- Large datacenter networks are often based on IP and Ethernet, arranged in Clos topologies to provide high bisection bandwidth [9]. Supercomputers often use specialized network topologies, such as multi-dimensional meshes and toruses [10], which yield better performance for HPC workloads with known communication patterns.
- The bigger a system gets, the more likely it is that one of its components is broken. Over time, broken things get fixed and new things break, but in a system with thousands of nodes, it is reasonable to assume that *something* is always broken [7]. When the error handling strategy consists of simply giving up, a large system can end up spending a lot of its time recovering from faults rather than doing useful work [8].
- If the system can tolerate failed nodes and still keep working as a whole, that is a very useful feature for operations and maintenance: for example, you can perform a rolling upgrade (see Chapter 4), restarting one node at a time, while the service continues serving users without interruption. In cloud environments, if one virtual machine is not performing well, you can just kill it and request a new one (hoping that the new one will be faster).
- In a geographically distributed deployment (keeping data geographically close to your users to reduce access latency), communication most likely goes over the internet, which is slow and unreliable compared to local networks. Supercomputers generally assume that all of their nodes are close together.

If we want to make distributed systems work, we must accept the possibility of partial failure and build fault-tolerance mechanisms into the software. In other words, we need to build a reliable system from unreliable components. (As discussed in "Reliability" on page 6, there is no such thing as perfect reliability, so we'll need to understand the limits of what we can realistically promise.)

Even in smaller systems consisting of only a few nodes, it's important to think about partial failure. In a small system, it's quite likely that most of the components are working correctly most of the time. However, sooner or later, some part of the system

*will* become faulty, and the software will have to somehow handle it. The fault handling must be part of the software design, and you (as operator of the software) need to know what behavior to expect from the software in the case of a fault.

It would be unwise to assume that faults are rare and simply hope for the best. It is important to consider a wide range of possible faults—even fairly unlikely ones—and to artificially create such situations in your testing environment to see what happens. In distributed systems, suspicion, pessimism, and paranoia pay off.

### Building a Reliable System from Unreliable Components

You may wonder whether this makes any sense—intuitively it may seem like a system can only be as reliable as its least reliable component (its *weakest link*). This is not the case: in fact, it is an old idea in computing to construct a more reliable system from a less reliable underlying base [11]. For example:

- Error-correcting codes allow digital data to be transmitted accurately across a communication channel that occasionally gets some bits wrong, for example due to radio interference on a wireless network [12].
- IP (the Internet Protocol) is unreliable: it may drop, delay, duplicate, or reorder packets. TCP (the Transmission Control Protocol) provides a more reliable transport layer on top of IP: it ensures that missing packets are retransmitted, duplicates are eliminated, and packets are reassembled into the order in which they were sent.

Although the system can be more reliable than its underlying parts, there is always a limit to how much more reliable it can be. For example, error-correcting codes can deal with a small number of single-bit errors, but if your signal is swamped by interference, there is a fundamental limit to how much data you can get through your communication channel [13]. TCP can hide packet loss, duplication, and reordering from you, but it cannot magically remove delays in the network.

Although the more reliable higher-level system is not perfect, it's still useful because it takes care of some of the tricky low-level faults, and so the remaining faults are usually easier to reason about and deal with. We will explore this matter further in "The end-to-end argument" on page 519.

## Unreliable Networks

As discussed in the introduction to Part II, the distributed systems we focus on in this book are *shared-nothing systems*: i.e., a bunch of machines connected by a network. The network is the only way those machines can communicate—we assume that each

machine has its own memory and disk, and one machine cannot access another machine's memory or disk (except by making requests to a service over the network).

Shared-nothing is not the only way of building systems, but it has become the dominant approach for building internet services, for several reasons: it's comparatively cheap because it requires no special hardware, it can make use of commoditized cloud computing services, and it can achieve high reliability through redundancy across multiple geographically distributed datacenters.

The internet and most internal networks in datacenters (often Ethernet) are *asynchronous packet networks*. In this kind of network, one node can send a message (a packet) to another node, but the network gives no guarantees as to when it will arrive, or whether it will arrive at all. If you send a request and expect a response, many things could go wrong (some of which are illustrated in Figure 8-1):

1. Your request may have been lost (perhaps someone unplugged a network cable).
2. Your request may be waiting in a queue and will be delivered later (perhaps the network or the recipient is overloaded).
3. The remote node may have failed (perhaps it crashed or it was powered down).
4. The remote node may have temporarily stopped responding (perhaps it is experiencing a long garbage collection pause; see "Process Pauses" on page 295), but it will start responding again later.
5. The remote node may have processed your request, but the response has been lost on the network (perhaps a network switch has been misconfigured).
6. The remote node may have processed your request, but the response has been delayed and will be delivered later (perhaps the network or your own machine is overloaded).

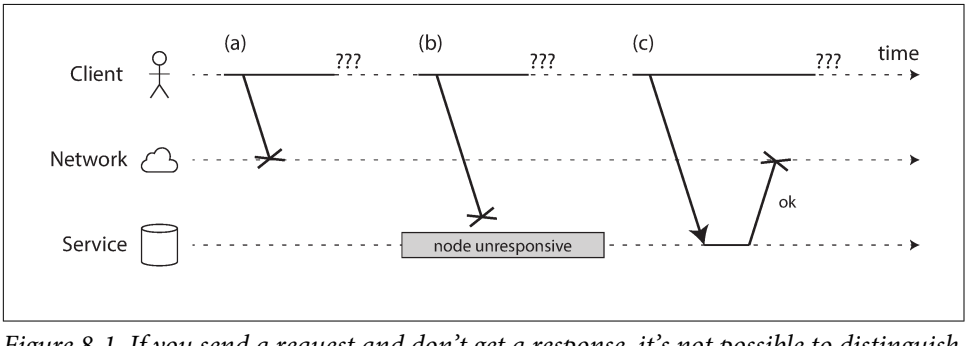



*Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.*

The sender can't even tell whether the packet was delivered: the only option is for the recipient to send a response message, which may in turn be lost or delayed. These issues are indistinguishable in an asynchronous network: the only information you have is that you haven't received a response yet. If you send a request to another node and don't receive a response, it is *impossible* to tell why.

The usual way of handling this issue is a *timeout*: after some time you give up waiting and assume that the response is not going to arrive. However, when a timeout occurs, you still don't know whether the remote node got your request or not (and if the request is still queued somewhere, it may still be delivered to the recipient, even if the sender has given up on it).

## Network Faults in Practice

We have been building computer networks for decades—one might hope that by now we would have figured out how to make them reliable. However, it seems that we have not yet succeeded.

There are some systematic studies, and plenty of anecdotal evidence, showing that network problems can be surprisingly common, even in controlled environments like a datacenter operated by one company [14]. One study in a medium-sized datacenter found about 12 network faults per month, of which half disconnected a single machine, and half disconnected an entire rack [15]. Another study measured the failure rates of components like top-of-rack switches, aggregation switches, and load balancers [16]. It found that adding redundant networking gear doesn't reduce faults as much as you might hope, since it doesn't guard against human error (e.g., misconfigured switches), which is a major cause of outages.

Public cloud services such as EC2 are notorious for having frequent transient network glitches [14], and well-managed private datacenter networks can be stabler environments. Nevertheless, nobody is immune from network problems: for example, a problem during a software upgrade for a switch could trigger a network topology reconfiguration, during which network packets could be delayed for more than a minute [17]. Sharks might bite undersea cables and damage them [18]. Other surprising faults include a network interface that sometimes drops all inbound packets but sends outbound packets successfully [19]: just because a network link works in one direction doesn't guarantee it's also working in the opposite direction.

### Network partitions

When one part of the network is cut off from the rest due to a network fault, that is sometimes called a *network partition* or *netsplit*. In this book we'll generally stick with the more general term *network fault*, to avoid confusion with partitions (shards) of a storage system, as discussed in Chapter 6.

Even if network faults are rare in your environment, the fact that faults *can* occur means that your software needs to be able to handle them. Whenever any communication happens over a network, it may fail—there is no way around it.

If the error handling of network faults is not defined and tested, arbitrarily bad things could happen: for example, the cluster could become deadlocked and permanently unable to serve requests, even when the network recovers [20], or it could even delete all of your data [21]. If software is put in an unanticipated situation, it may do arbitrary unexpected things.

Handling network faults doesn't necessarily mean *tolerating* them: if your network is normally fairly reliable, a valid approach may be to simply show an error message to users while your network is experiencing problems. However, you do need to know how your software reacts to network problems and ensure that the system can recover from them. It may make sense to deliberately trigger network problems and test the system's response (this is the idea behind Chaos Monkey; see "Reliability" on page 6).

## Detecting Faults

Many systems need to automatically detect faulty nodes. For example:

- A load balancer needs to stop sending requests to a node that is dead (i.e., take it *out of rotation*).
- In a distributed database with single-leader replication, if the leader fails, one of the followers needs to be promoted to be the new leader (see "Handling Node Outages" on page 156).

Unfortunately, the uncertainty about the network makes it difficult to tell whether a node is working or not. In some specific circumstances you might get some feedback to explicitly tell you that something is not working:

- If you can reach the machine on which the node should be running, but no process is listening on the destination port (e.g., because the process crashed), the operating system will helpfully close or refuse TCP connections by sending a `RST` or `FIN` packet in reply. However, if the node crashed while it was handling your request, you have no way of knowing how much data was actually processed by the remote node [22].
- If a node process crashed (or was killed by an administrator) but the node's operating system is still running, a script can notify other nodes about the crash so that another node can take over quickly without having to wait for a timeout to expire. For example, HBase does this [23].

- If you have access to the management interface of the network switches in your datacenter, you can query them to detect link failures at a hardware level (e.g., if the remote machine is powered down). This option is ruled out if you're connecting via the internet, or if you're in a shared datacenter with no access to the switches themselves, or if you can't reach the management interface due to a network problem.
- If a router is sure that the IP address you're trying to connect to is unreachable, it may reply to you with an ICMP Destination Unreachable packet. However, the router doesn't have a magic failure detection capability either—it is subject to the same limitations as other participants of the network.

Rapid feedback about a remote node being down is useful, but you can't count on it. Even if TCP acknowledges that a packet was delivered, the application may have crashed before handling it. If you want to be sure that a request was successful, you need a positive response from the application itself [24].

Conversely, if something has gone wrong, you may get an error response at some level of the stack, but in general you have to assume that you will get no response at all. You can retry a few times (TCP retries transparently, but you may also retry at the application level), wait for a timeout to elapse, and eventually declare the node dead if you don't hear back within the timeout.

## Timeouts and Unbounded Delays

If a timeout is the only sure way of detecting a fault, then how long should the timeout be? There is unfortunately no simple answer.

A long timeout means a long wait until a node is declared dead (and during this time, users may have to wait or see error messages). A short timeout detects faults faster, but carries a higher risk of incorrectly declaring a node dead when in fact it has only suffered a temporary slowdown (e.g., due to a load spike on the node or the network).

Prematurely declaring a node dead is problematic: if the node is actually alive and in the middle of performing some action (for example, sending an email), and another node takes over, the action may end up being performed twice. We will discuss this issue in more detail in "Knowledge, Truth, and Lies" on page 300, and in Chapters 9 and 11.

When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network. If the system is already struggling with high load, declaring nodes dead prematurely can make the problem worse. In particular, it could happen that the node actually wasn't dead but only slow to respond due to overload; transferring its load to other nodes can cause a cascading failure (in the extreme case, all nodes declare each other dead, and everything stops working).

Imagine a fictitious system with a network that guaranteed a maximum delay for packets—every packet is either delivered within some time $d$, or it is lost, but delivery never takes longer than $d$. Furthermore, assume that you can guarantee that a non-failed node always handles a request within some time $r$. In this case, you could guarantee that every successful request receives a response within time $2d + r$—and if you don't receive a response within that time, you know that either the network or the remote node is not working. If this was true, $2d + r$ would be a reasonable timeout to use.

Unfortunately, most systems we work with have neither of those guarantees: asynchronous networks have *unbounded delays* (that is, they try to deliver packets as quickly as possible, but there is no upper limit on the time it may take for a packet to arrive), and most server implementations cannot guarantee that they can handle requests within some maximum time (see "Response time guarantees" on page 298). For failure detection, it's not sufficient for the system to be fast most of the time: if your timeout is low, it only takes a transient spike in round-trip times to throw the system off-balance.

### Network congestion and queueing

When driving a car, travel times on road networks often vary most due to traffic congestion. Similarly, the variability of packet delays on computer networks is most often due to queueing [25]:

- If several different nodes simultaneously try to send packets to the same destination, the network switch must queue them up and feed them into the destination network link one by one (as illustrated in Figure 8-2). On a busy network link, a packet may have to wait a while until it can get a slot (this is called *network congestion*). If there is so much incoming data that the switch queue fills up, the packet is dropped, so it needs to be resent—even though the network is functioning fine.
- When a packet reaches the destination machine, if all CPU cores are currently busy, the incoming request from the network is queued by the operating system until the application is ready to handle it. Depending on the load on the machine, this may take an arbitrary length of time.
- In virtualized environments, a running operating system is often paused for tens of milliseconds while another virtual machine uses a CPU core. During this time, the VM cannot consume any data from the network, so the incoming data is queued (buffered) by the virtual machine monitor [26], further increasing the variability of network delays.
- TCP performs *flow control* (also known as *congestion avoidance* or *backpressure*), in which a node limits its own rate of sending in order to avoid overloading a

network link or the receiving node [27]. This means additional queueing at the sender before the data even enters the network.

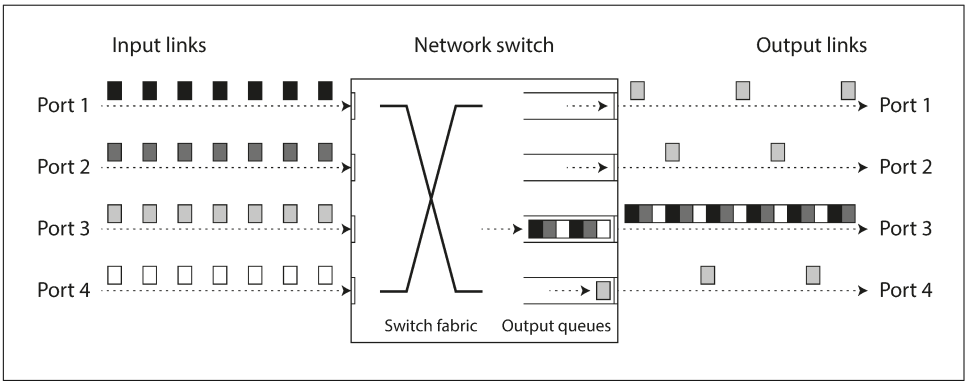



*Figure 8-2. If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2, and 4 are all trying to send packets to port 3.*

Moreover, TCP considers a packet to be lost if it is not acknowledged within some timeout (which is calculated from observed round-trip times), and lost packets are automatically retransmitted. Although the application does not see the packet loss and retransmission, it does see the resulting delay (waiting for the timeout to expire, and then waiting for the retransmitted packet to be acknowledged).

### TCP Versus UDP

Some latency-sensitive applications, such as videoconferencing and Voice over IP (VoIP), use UDP rather than TCP. It's a trade-off between reliability and variability of delays: as UDP does not perform flow control and does not retransmit lost packets, it avoids some of the reasons for variable network delays (although it is still susceptible to switch queues and scheduling delays).

UDP is a good choice in situations where delayed data is worthless. For example, in a VoIP phone call, there probably isn't enough time to retransmit a lost packet before its data is due to be played over the loudspeakers. In this case, there's no point in retransmitting the packet—the application must instead fill the missing packet's time slot with silence (causing a brief interruption in the sound) and move on in the stream. The retry happens at the human layer instead. ("Could you repeat that please? The sound just cut out for a moment.")

All of these factors contribute to the variability of network delays. Queueing delays have an especially wide range when a system is close to its maximum capacity: a sys-

tem with plenty of spare capacity can easily drain queues, whereas in a highly utilized system, long queues can build up very quickly.

In public clouds and multi-tenant datacenters, resources are shared among many customers: the network links and switches, and even each machine's network interface and CPUs (when running on virtual machines), are shared. Batch workloads such as MapReduce (see Chapter 10) can easily saturate network links. As you have no control over or insight into other customers' usage of the shared resources, network delays can be highly variable if someone near you (a *noisy neighbor*) is using a lot of resources [28, 29].

In such environments, you can only choose timeouts experimentally: measure the distribution of network round-trip times over an extended period, and over many machines, to determine the expected variability of delays. Then, taking into account your application's characteristics, you can determine an appropriate trade-off between failure detection delay and risk of premature timeouts.

Even better, rather than using configured constant timeouts, systems can continually measure response times and their variability (*jitter*), and automatically adjust timeouts according to the observed response time distribution. This can be done with a Phi Accrual failure detector [30], which is used for example in Akka and Cassandra [31]. TCP retransmission timeouts also work similarly [27].

## Synchronous Versus Asynchronous Networks

Distributed systems would be a lot simpler if we could rely on the network to deliver packets with some fixed maximum delay, and not to drop packets. Why can't we solve this at the hardware level and make the network reliable so that the software doesn't need to worry about it?

To answer this question, it's interesting to compare datacenter networks to the traditional fixed-line telephone network (non-cellular, non-VoIP), which is extremely reliable: delayed audio frames and dropped calls are very rare. A phone call requires a constantly low end-to-end latency and enough bandwidth to transfer the audio samples of your voice. Wouldn't it be nice to have similar reliability and predictability in computer networks?

When you make a call over the telephone network, it establishes a *circuit*: a fixed, guaranteed amount of bandwidth is allocated for the call, along the entire route between the two callers. This circuit remains in place until the call ends [32]. For example, an ISDN network runs at a fixed rate of 4,000 frames per second. When a call is established, it is allocated 16 bits of space within each frame (in each direction). Thus, for the duration of the call, each side is guaranteed to be able to send exactly 16 bits of audio data every 250 microseconds [33, 34].

This kind of network is *synchronous*: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been reserved in the next hop of the network. And because there is no queueing, the maximum end-to-end latency of the network is fixed. We call this a *bounded delay*.

### Can we not simply make network delays predictable?

Note that a circuit in a telephone network is very different from a TCP connection: a circuit is a fixed amount of reserved bandwidth which nobody else can use while the circuit is established, whereas the packets of a TCP connection opportunistically use whatever network bandwidth is available. You can give TCP a variable-sized block of data (e.g., an email or a web page), and it will try to transfer it in the shortest time possible. While a TCP connection is idle, it doesn't use any bandwidth.[ii]

If datacenter networks and the internet were circuit-switched networks, it would be possible to establish a guaranteed maximum round-trip time when a circuit was set up. However, they are not: Ethernet and IP are packet-switched protocols, which suffer from queueing and thus unbounded delays in the network. These protocols do not have the concept of a circuit.

Why do datacenter networks and the internet use packet switching? The answer is that they are optimized for *bursty traffic*. A circuit is good for an audio or video call, which needs to transfer a fairly constant number of bits per second for the duration of the call. On the other hand, requesting a web page, sending an email, or transferring a file doesn't have any particular bandwidth requirement—we just want it to complete as quickly as possible.

If you wanted to transfer a file over a circuit, you would have to guess a bandwidth allocation. If you guess too low, the transfer is unnecessarily slow, leaving network capacity unused. If you guess too high, the circuit cannot be set up (because the network cannot allow a circuit to be created if its bandwidth allocation cannot be guaranteed). Thus, using circuits for bursty data transfers wastes network capacity and makes transfers unnecessarily slow. By contrast, TCP dynamically adapts the rate of data transfer to the available network capacity.

There have been some attempts to build hybrid networks that support both circuit switching and packet switching, such as ATM.[iii] InfiniBand has some similarities [35]: it implements end-to-end flow control at the link layer, which reduces the need for

ii. Except perhaps for an occasional keepalive packet, if TCP keepalive is enabled.

iii. *Asynchronous Transfer Mode* (ATM) was a competitor to Ethernet in the 1980s [32], but it didn't gain much adoption outside of telephone network core switches. It has nothing to do with automatic teller machines (also known as cash machines), despite sharing an acronym. Perhaps, in some parallel universe, the internet is based on something like ATM—in that universe, internet video calls are probably a lot more reliable than they are in ours, because they don't suffer from dropped and delayed packets.

queueing in the network, although it can still suffer from delays due to link congestion [36]. With careful use of *quality of service* (QoS, prioritization and scheduling of packets) and *admission control* (rate-limiting senders), it is possible to emulate circuit switching on packet networks, or provide statistically bounded delay [25, 32].

## Latency and Resource Utilization

More generally, you can think of variable delays as a consequence of dynamic resource partitioning.

Say you have a wire between two telephone switches that can carry up to 10,000 simultaneous calls. Each circuit that is switched over this wire occupies one of those call slots. Thus, you can think of the wire as a resource that can be shared by up to 10,000 simultaneous users. The resource is divided up in a *static* way: even if you're the only call on the wire right now, and all other 9,999 slots are unused, your circuit is still allocated the same fixed amount of bandwidth as when the wire is fully utilized.

By contrast, the internet shares network bandwidth *dynamically*. Senders push and jostle with each other to get their packets over the wire as quickly as possible, and the network switches decide which packet to send (i.e., the bandwidth allocation) from one moment to the next. This approach has the downside of queueing, but the advantage is that it maximizes utilization of the wire. The wire has a fixed cost, so if you utilize it better, each byte you send over the wire is cheaper.

A similar situation arises with CPUs: if you share each CPU core dynamically between several threads, one thread sometimes has to wait in the operating system's run queue while another thread is running, so a thread can be paused for varying lengths of time. However, this utilizes the hardware better than if you allocated a static number of CPU cycles to each thread (see "Response time guarantees" on page 298). Better hardware utilization is also a significant motivation for using virtual machines.

Latency guarantees are achievable in certain environments, if resources are statically partitioned (e.g., dedicated hardware and exclusive bandwidth allocations). However, it comes at the cost of reduced utilization—in other words, it is more expensive. On the other hand, multi-tenancy with dynamic resource partitioning provides better utilization, so it is cheaper, but it has the downside of variable delays.

Variable delays in networks are not a law of nature, but simply the result of a cost/benefit trade-off.

However, such quality of service is currently not enabled in multi-tenant datacenters and public clouds, or when communicating via the internet.[iv] Currently deployed technology does not allow us to make any guarantees about delays or reliability of the network: we have to assume that network congestion, queueing, and unbounded delays will happen. Consequently, there's no "correct" value for timeouts—they need to be determined experimentally.

## Unreliable Clocks

Clocks and time are important. Applications depend on clocks in various ways to answer questions like the following:

1. Has this request timed out yet?
2. What's the 99th percentile response time of this service?
3. How many queries per second did this service handle on average in the last five minutes?
4. How long did the user spend on our site?
5. When was this article published?
6. At what date and time should the reminder email be sent?
7. When does this cache entry expire?
8. What is the timestamp on this error message in the log file?

Examples 1–4 measure *durations* (e.g., the time interval between a request being sent and a response being received), whereas examples 5–8 describe *points in time* (events that occur on a particular date, at a particular time).

In a distributed system, time is a tricky business, because communication is not instantaneous: it takes time for a message to travel across the network from one machine to another. The time when a message is received is always later than the time when it is sent, but due to variable delays in the network, we don't know how much later. This fact sometimes makes it difficult to determine the order in which things happened when multiple machines are involved.

Moreover, each machine on the network has its own clock, which is an actual hardware device: usually a quartz crystal oscillator. These devices are not perfectly accurate, so each machine has its own notion of time, which may be slightly faster or

iv. Peering agreements between internet service providers and the establishment of routes through the Border Gateway Protocol (BGP), bear closer resemblance to circuit switching than IP itself. At this level, it is possible to buy dedicated bandwidth. However, internet routing operates at the level of networks, not individual connections between hosts, and at a much longer timescale.

slower than on other machines. It is possible to synchronize clocks to some degree: the most commonly used mechanism is the Network Time Protocol (NTP), which allows the computer clock to be adjusted according to the time reported by a group of servers [37]. The servers in turn get their time from a more accurate time source, such as a GPS receiver.

## Monotonic Versus Time-of-Day Clocks

Modern computers have at least two different kinds of clocks: a *time-of-day clock* and a *monotonic clock*. Although they both measure time, it is important to distinguish the two, since they serve different purposes.

### Time-of-day clocks

A time-of-day clock does what you intuitively expect of a clock: it returns the current date and time according to some calendar (also known as *wall-clock time*). For example, `clock_gettime(CLOCK_REALTIME)` on Linux[v] and `System.currentTimeMillis()` in Java return the number of seconds (or milliseconds) since the *epoch*: midnight UTC on January 1, 1970, according to the Gregorian calendar, not counting leap seconds. Some systems use other dates as their reference point.

Time-of-day clocks are usually synchronized with NTP, which means that a timestamp from one machine (ideally) means the same as a timestamp on another machine. However, time-of-day clocks also have various oddities, as described in the next section. In particular, if the local clock is too far ahead of the NTP server, it may be forcibly reset and appear to jump back to a previous point in time. These jumps, as well as similar jumps caused by leap seconds, make time-of-day clocks unsuitable for measuring elapsed time [38].

Time-of-day clocks have also historically had quite a coarse-grained resolution, e.g., moving forward in steps of 10 ms on older Windows systems [39]. On recent systems, this is less of a problem.

### Monotonic clocks

A monotonic clock is suitable for measuring a duration (time interval), such as a timeout or a service's response time: `clock_gettime(CLOCK_MONOTONIC)` on Linux and `System.nanoTime()` in Java are monotonic clocks, for example. The name comes from the fact that they are guaranteed to always move forward (whereas a time-of-day clock may jump back in time).



v. Although the clock is called *real-time*, it has nothing to do with real-time operating systems, as discussed in "Response time guarantees" on page 298.

You can check the value of the monotonic clock at one point in time, do something, and then check the clock again at a later time. The *difference* between the two values tells you how much time elapsed between the two checks. However, the *absolute* value of the clock is meaningless: it might be the number of nanoseconds since the computer was started, or something similarly arbitrary. In particular, it makes no sense to compare monotonic clock values from two different computers, because they don't mean the same thing.

On a server with multiple CPU sockets, there may be a separate timer per CPU, which is not necessarily synchronized with other CPUs. Operating systems compensate for any discrepancy and try to present a monotonic view of the clock to application threads, even as they are scheduled across different CPUs. However, it is wise to take this guarantee of monotonicity with a pinch of salt [40].

NTP may adjust the frequency at which the monotonic clock moves forward (this is known as *slewing* the clock) if it detects that the computer's local quartz is moving faster or slower than the NTP server. By default, NTP allows the clock rate to be speeded up or slowed down by up to 0.05%, but NTP cannot cause the monotonic clock to jump forward or backward. The resolution of monotonic clocks is usually quite good: on most systems they can measure time intervals in microseconds or less.

In a distributed system, using a monotonic clock for measuring elapsed time (e.g., timeouts) is usually fine, because it doesn't assume any synchronization between different nodes' clocks and is not sensitive to slight inaccuracies of measurement.

## Clock Synchronization and Accuracy

Monotonic clocks don't need synchronization, but time-of-day clocks need to be set according to an NTP server or other external time source in order to be useful. Unfortunately, our methods for getting a clock to tell the correct time aren't nearly as reliable or accurate as you might hope—hardware clocks and NTP can be fickle beasts. To give just a few examples:

- The quartz clock in a computer is not very accurate: it *drifts* (runs faster or slower than it should). Clock drift varies depending on the temperature of the machine. Google assumes a clock drift of 200 ppm (parts per million) for its servers [41], which is equivalent to 6 ms drift for a clock that is resynchronized with a server every 30 seconds, or 17 seconds drift for a clock that is resynchronized once a day. This drift limits the best possible accuracy you can achieve, even if everything is working correctly.
- If a computer's clock differs too much from an NTP server, it may refuse to synchronize, or the local clock will be forcibly reset [37]. Any applications observing the time before and after this reset may see time go backward or suddenly jump forward.

- If a node is accidentally firewalled off from NTP servers, the misconfiguration may go unnoticed for some time. Anecdotal evidence suggests that this does happen in practice.
- NTP synchronization can only be as good as the network delay, so there is a limit to its accuracy when you're on a congested network with variable packet delays. One experiment showed that a minimum error of 35 ms is achievable when synchronizing over the internet [42], though occasional spikes in network delay lead to errors of around a second. Depending on the configuration, large network delays can cause the NTP client to give up entirely.
- Some NTP servers are wrong or misconfigured, reporting time that is off by hours [43, 44]. NTP clients are quite robust, because they query several servers and ignore outliers. Nevertheless, it's somewhat worrying to bet the correctness of your systems on the time that you were told by a stranger on the internet.
- Leap seconds result in a minute that is 59 seconds or 61 seconds long, which messes up timing assumptions in systems that are not designed with leap seconds in mind [45]. The fact that leap seconds have crashed many large systems [38, 46] shows how easy it is for incorrect assumptions about clocks to sneak into a system. The best way of handling leap seconds may be to make NTP servers "lie," by performing the leap second adjustment gradually over the course of a day (this is known as *smearing*) [47, 48], although actual NTP server behavior varies in practice [49].
- In virtual machines, the hardware clock is virtualized, which raises additional challenges for applications that need accurate timekeeping [50]. When a CPU core is shared between virtual machines, each VM is paused for tens of milliseconds while another VM is running. From an application's point of view, this pause manifests itself as the clock suddenly jumping forward [26].
- If you run software on devices that you don't fully control (e.g., mobile or embedded devices), you probably cannot trust the device's hardware clock at all. Some users deliberately set their hardware clock to an incorrect date and time, for example to circumvent timing limitations in games. As a result, the clock might be set to a time wildly in the past or the future.

It is possible to achieve very good clock accuracy if you care about it sufficiently to invest significant resources. For example, the MiFID II draft European regulation for financial institutions requires all high-frequency trading funds to synchronize their clocks to within 100 microseconds of UTC, in order to help debug market anomalies such as "flash crashes" and to help detect market manipulation [51].

Such accuracy can be achieved using GPS receivers, the Precision Time Protocol (PTP) [52], and careful deployment and monitoring. However, it requires significant effort and expertise, and there are plenty of ways clock synchronization can go

wrong. If your NTP daemon is misconfigured, or a firewall is blocking NTP traffic, the clock error due to drift can quickly become large.

## Relying on Synchronized Clocks

The problem with clocks is that while they seem simple and easy to use, they have a surprising number of pitfalls: a day may not have exactly 86,400 seconds, time-of-day clocks may move backward in time, and the time on one node may be quite different from the time on another node.

Earlier in this chapter we discussed networks dropping and arbitrarily delaying packets. Even though networks are well behaved most of the time, software must be designed on the assumption that the network will occasionally be faulty, and the software must handle such faults gracefully. The same is true with clocks: although they work quite well most of the time, robust software needs to be prepared to deal with incorrect clocks.

Part of the problem is that incorrect clocks easily go unnoticed. If a machine's CPU is defective or its network is misconfigured, it most likely won't work at all, so it will quickly be noticed and fixed. On the other hand, if its quartz clock is defective or its NTP client is misconfigured, most things will seem to work fine, even though its clock gradually drifts further and further away from reality. If some piece of software is relying on an accurately synchronized clock, the result is more likely to be silent and subtle data loss than a dramatic crash [53, 54].

Thus, if you use software that requires synchronized clocks, it is essential that you also carefully monitor the clock offsets between all the machines. Any node whose clock drifts too far from the others should be declared dead and removed from the cluster. Such monitoring ensures that you notice the broken clocks before they can cause too much damage.

### Timestamps for ordering events

Let's consider one particular situation in which it is tempting, but dangerous, to rely on clocks: ordering of events across multiple nodes. For example, if two clients write to a distributed database, who got there first? Which write is the more recent one?

Figure 8-3 illustrates a dangerous use of time-of-day clocks in a database with multi-leader replication (the example is similar to Figure 5-9). Client A writes $x = 1$ on node 1; the write is replicated to node 3; client B increments $x$ on node 3 (we now have $x = 2$); and finally, both writes are replicated to node 2.

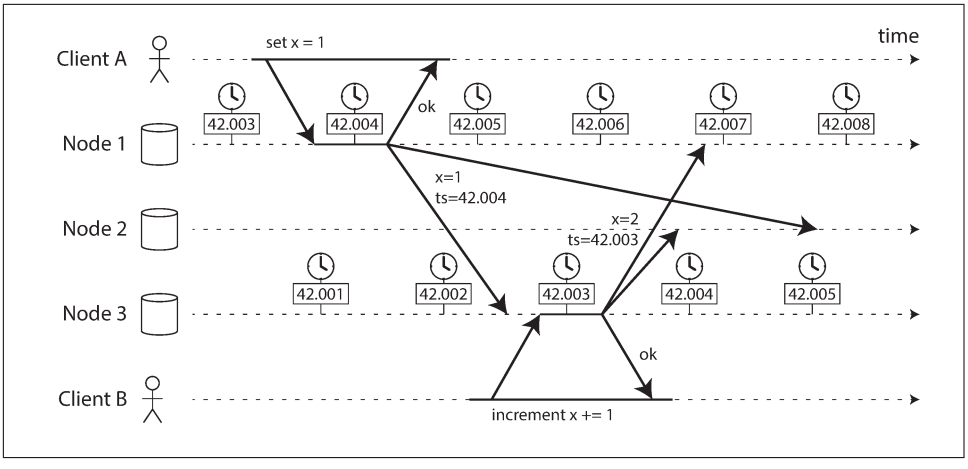



*Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.*

In Figure 8-3, when a write is replicated to other nodes, it is tagged with a timestamp according to the time-of-day clock on the node where the write originated. The clock synchronization is very good in this example: the skew between node 1 and node 3 is less than 3 ms, which is probably better than you can expect in practice.

Nevertheless, the timestamps in Figure 8-3 fail to order the events correctly: the write $x = 1$ has a timestamp of 42.004 seconds, but the write $x = 2$ has a timestamp of 42.003 seconds, even though $x = 2$ occurred unambiguously later. When node 2 receives these two events, it will incorrectly conclude that $x = 1$ is the more recent value and drop the write $x = 2$. In effect, client B's increment operation will be lost.

This conflict resolution strategy is called *last write wins* (LWW), and it is widely used in both multi-leader replication and leaderless databases such as Cassandra [53] and Riak [54] (see "Last write wins (discarding concurrent writes)" on page 186). Some implementations generate timestamps on the client rather than the server, but this doesn't change the fundamental problems with LWW:

- Database writes can mysteriously disappear: a node with a lagging clock is unable to overwrite values previously written by a node with a fast clock until the clock skew between the nodes has elapsed [54, 55]. This scenario can cause arbitrary amounts of data to be silently dropped without any error being reported to the application.
- LWW cannot distinguish between writes that occurred sequentially in quick succession (in Figure 8-3, client B's increment definitely occurs *after* client A's write) and writes that were truly concurrent (neither writer was aware of the other). Additional causality tracking mechanisms, such as version vectors, are

needed in order to prevent violations of causality (see "Detecting Concurrent Writes" on page 184).

- It is possible for two nodes to independently generate writes with the same timestamp, especially when the clock only has millisecond resolution. An additional tiebreaker value (which can simply be a large random number) is required to resolve such conflicts, but this approach can also lead to violations of causality [53].

Thus, even though it is tempting to resolve conflicts by keeping the most "recent" value and discarding others, it's important to be aware that the definition of "recent" depends on a local time-of-day clock, which may well be incorrect. Even with tightly NTP-synchronized clocks, you could send a packet at timestamp 100 ms (according to the sender's clock) and have it arrive at timestamp 99 ms (according to the recipient's clock)—so it appears as though the packet arrived before it was sent, which is impossible.

Could NTP synchronization be made accurate enough that such incorrect orderings cannot occur? Probably not, because NTP's synchronization accuracy is itself limited by the network round-trip time, in addition to other sources of error such as quartz drift. For correct ordering, you would need the clock source to be significantly more accurate than the thing you are measuring (namely network delay).

So-called *logical clocks* [56, 57], which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events (see "Detecting Concurrent Writes" on page 184). Logical clocks do not measure the time of day or the number of seconds elapsed, only the relative ordering of events (whether one event happened before or after another). In contrast, time-of-day and monotonic clocks, which measure actual elapsed time, are also known as *physical clocks*. We'll look at ordering a bit more in "Ordering Guarantees" on page 339.

### Clock readings have a confidence interval

You may be able to read a machine's time-of-day clock with microsecond or even nanosecond resolution. But even if you can get such a fine-grained measurement, that doesn't mean the value is actually accurate to such precision. In fact, it most likely is not—as mentioned previously, the drift in an imprecise quartz clock can easily be several milliseconds, even if you synchronize with an NTP server on the local network every minute. With an NTP server on the public internet, the best possible accuracy is probably to the tens of milliseconds, and the error may easily spike to over 100 ms when there is network congestion [57].

Thus, it doesn't make sense to think of a clock reading as a point in time—it is more like a range of times, within a confidence interval: for example, a system may be 95% confident that the time now is between 10.3 and 10.5 seconds past the minute, but it

doesn't know any more precisely than that [58]. If we only know the time +/– 100 ms, the microsecond digits in the timestamp are essentially meaningless.

The uncertainty bound can be calculated based on your time source. If you have a GPS receiver or atomic (caesium) clock directly attached to your computer, the expected error range is reported by the manufacturer. If you're getting the time from a server, the uncertainty is based on the expected quartz drift since your last sync with the server, plus the NTP server's uncertainty, plus the network round-trip time to the server (to a first approximation, and assuming you trust the server).

Unfortunately, most systems don't expose this uncertainty: for example, when you call `clock_gettime()`, the return value doesn't tell you the expected error of the timestamp, so you don't know if its confidence interval is five milliseconds or five years.

An interesting exception is Google's *TrueTime* API in Spanner [41], which explicitly reports the confidence interval on the local clock. When you ask it for the current time, you get back two values: [*`earliest, latest`*], which are the *earliest possible* and the *latest possible* timestamp. Based on its uncertainty calculations, the clock knows that the actual current time is somewhere within that interval. The width of the interval depends, among other things, on how long it has been since the local quartz clock was last synchronized with a more accurate clock source.

### Synchronized clocks for global snapshots

In "Snapshot Isolation and Repeatable Read" on page 237 we discussed *snapshot isolation*, which is a very useful feature in databases that need to support both small, fast read-write transactions and large, long-running read-only transactions (e.g., for backups or analytics). It allows read-only transactions to see the database in a consistent state at a particular point in time, without locking and interfering with read-write transactions.

The most common implementation of snapshot isolation requires a monotonically increasing transaction ID. If a write happened later than the snapshot (i.e., the write has a greater transaction ID than the snapshot), that write is invisible to the snapshot transaction. On a single-node database, a simple counter is sufficient for generating transaction IDs.

However, when a database is distributed across many machines, potentially in multiple datacenters, a global, monotonically increasing transaction ID (across all partitions) is difficult to generate, because it requires coordination. The transaction ID must reflect causality: if transaction B reads a value that was written by transaction A, then B must have a higher transaction ID than A—otherwise, the snapshot would not

be consistent. With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck.[vi]

Can we use the timestamps from synchronized time-of-day clocks as transaction IDs? If we could get the synchronization good enough, they would have the right properties: later transactions have a higher timestamp. The problem, of course, is the uncertainty about clock accuracy.

Spanner implements snapshot isolation across datacenters in this way [59, 60]. It uses the clock's confidence interval as reported by the TrueTime API, and is based on the following observation: if you have two confidence intervals, each consisting of an earliest and latest possible timestamp ($A = [A_{earliest}, A_{latest}]$ and $B = [B_{earliest}, B_{latest}]$), and those two intervals do not overlap (i.e., $A_{earliest} < A_{latest} < B_{earliest} < B_{latest}$), then B definitely happened after A—there can be no doubt. Only if the intervals overlap are we unsure in which order A and B happened.

In order to ensure that transaction timestamps reflect causality, Spanner deliberately waits for the length of the confidence interval before committing a read-write transaction. By doing so, it ensures that any transaction that may read the data is at a sufficiently later time, so their confidence intervals do not overlap. In order to keep the wait time as short as possible, Spanner needs to keep the clock uncertainty as small as possible; for this purpose, Google deploys a GPS receiver or atomic clock in each datacenter, allowing clocks to be synchronized to within about 7 ms [41].

Using clock synchronization for distributed transaction semantics is an area of active research [57, 61, 62]. These ideas are interesting, but they have not yet been implemented in mainstream databases outside of Google.

## Process Pauses

Let's consider another example of dangerous clock use in a distributed system. Say you have a database with a single leader per partition. Only the leader is allowed to accept writes. How does a node know that it is still leader (that it hasn't been declared dead by the others), and that it may safely accept writes?

One option is for the leader to obtain a *lease* from the other nodes, which is similar to a lock with a timeout [63]. Only one node can hold the lease at any one time—thus, when a node obtains a lease, it knows that it is the leader for some amount of time, until the lease expires. In order to remain leader, the node must periodically renew

vi. There are distributed sequence number generators, such as Twitter's Snowflake, that generate *approximately* monotonically increasing unique IDs in a scalable way (e.g., by allocating blocks of the ID space to different nodes). However, they typically cannot guarantee an ordering that is consistent with causality, because the timescale at which blocks of IDs are assigned is longer than the timescale of database reads and writes. See also "Ordering Guarantees" on page 339.

the lease before it expires. If the node fails, it stops renewing the lease, so another node can take over when it expires.

You can imagine the request-handling loop looking something like this:

```
while (true) {
    request = getIncomingRequest();

    // Ensure that the lease always has at least 10 seconds remaining
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

What's wrong with this code? Firstly, it's relying on synchronized clocks: the expiry time on the lease is set by a different machine (where the expiry may be calculated as the current time plus 30 seconds, for example), and it's being compared to the local system clock. If the clocks are out of sync by more than a few seconds, this code will start doing strange things.

Secondly, even if we change the protocol to only use the local monotonic clock, there is another problem: the code assumes that very little time passes between the point that it checks the time (`System.currentTimeMillis()`) and the time when the request is processed (`process(request)`). Normally this code runs very quickly, so the 10 second buffer is more than enough to ensure that the lease doesn't expire in the middle of processing a request.

However, what if there is an unexpected pause in the execution of the program? For example, imagine the thread stops for 15 seconds around the line `lease.isValid()` before finally continuing. In that case, it's likely that the lease will have expired by the time the request is processed, and another node has already taken over as leader. However, there is nothing to tell this thread that it was paused for so long, so this code won't notice that the lease has expired until the next iteration of the loop—by which time it may have already done something unsafe by processing the request.

Is it crazy to assume that a thread might be paused for so long? Unfortunately not. There are various reasons why this could happen:

- Many programming language runtimes (such as the Java Virtual Machine) have a *garbage collector* (GC) that occasionally needs to stop all running threads. These *"stop-the-world" GC pauses* have sometimes been known to last for several minutes [64]! Even so-called "concurrent" garbage collectors like the HotSpot JVM's CMS cannot fully run in parallel with the application code—even they need to stop the world from time to time [65]. Although the pauses can often be

reduced by changing allocation patterns or tuning GC settings [66], we must assume the worst if we want to offer robust guarantees.

- In virtualized environments, a virtual machine can be *suspended* (pausing the execution of all processes and saving the contents of memory to disk) and *resumed* (restoring the contents of memory and continuing execution). This pause can occur at any time in a process's execution and can last for an arbitrary length of time. This feature is sometimes used for *live migration* of virtual machines from one host to another without a reboot, in which case the length of the pause depends on the rate at which processes are writing to memory [67].
- On end-user devices such as laptops, execution may also be suspended and resumed arbitrarily, e.g., when the user closes the lid of their laptop.
- When the operating system context-switches to another thread, or when the hypervisor switches to a different virtual machine (when running in a virtual machine), the currently running thread can be paused at any arbitrary point in the code. In the case of a virtual machine, the CPU time spent in other virtual machines is known as *steal time*. If the machine is under heavy load—i.e., if there is a long queue of threads waiting to run—it may take some time before the paused thread gets to run again.
- If the application performs synchronous disk access, a thread may be paused waiting for a slow disk I/O operation to complete [68]. In many languages, disk access can happen surprisingly, even if the code doesn't explicitly mention file access—for example, the Java classloader lazily loads class files when they are first used, which could happen at any time in the program execution. I/O pauses and GC pauses may even conspire to combine their delays [69]. If the disk is actually a network filesystem or network block device (such as Amazon's EBS), the I/O latency is further subject to the variability of network delays [29].
- If the operating system is configured to allow *swapping to disk* (*paging*), a simple memory access may result in a page fault that requires a page from disk to be loaded into memory. The thread is paused while this slow I/O operation takes place. If memory pressure is high, this may in turn require a different page to be swapped out to disk. In extreme circumstances, the operating system may spend most of its time swapping pages in and out of memory and getting little actual work done (this is known as *thrashing*). To avoid this problem, paging is often disabled on server machines (if you would rather kill a process to free up memory than risk thrashing).
- A Unix process can be paused by sending it the `SIGSTOP` signal, for example by pressing Ctrl-Z in a shell. This signal immediately stops the process from getting any more CPU cycles until it is resumed with `SIGCONT`, at which point it continues running where it left off. Even if your environment does not normally use `SIGSTOP`, it might be sent accidentally by an operations engineer.

All of these occurrences can *preempt* the running thread at any point and resume it at some later time, without the thread even noticing. The problem is similar to making multi-threaded code on a single machine thread-safe: you can't assume anything about timing, because arbitrary context switches and parallelism may occur.

When writing multi-threaded code on a single machine, we have fairly good tools for making it thread-safe: mutexes, semaphores, atomic counters, lock-free data structures, blocking queues, and so on. Unfortunately, these tools don't directly translate to distributed systems, because a distributed system has no shared memory—only messages sent over an unreliable network.

A node in a distributed system must assume that its execution can be paused for a significant length of time at any point, even in the middle of a function. During the pause, the rest of the world keeps moving and may even declare the paused node dead because it's not responding. Eventually, the paused node may continue running, without even noticing that it was asleep until it checks its clock sometime later.

### Response time guarantees

In many programming languages and operating systems, threads and processes may pause for an unbounded amount of time, as discussed. Those reasons for pausing *can* be eliminated if you try hard enough.

Some software runs in environments where a failure to respond within a specified time can cause serious damage: computers that control aircraft, rockets, robots, cars, and other physical objects must respond quickly and predictably to their sensor inputs. In these systems, there is a specified *deadline* by which the software must respond; if it doesn't meet the deadline, that may cause a failure of the entire system. These are so-called *hard real-time* systems.

#### Is real-time really real?

In embedded systems, *real-time* means that a system is carefully designed and tested to meet specified timing guarantees in all circumstances. This meaning is in contrast to the more vague use of the term *real-time* on the web, where it describes servers pushing data to clients and stream processing without hard response time constraints (see Chapter 11).

For example, if your car's onboard sensors detect that you are currently experiencing a crash, you wouldn't want the release of the airbag to be delayed due to an inopportune GC pause in the airbag release system.

Providing real-time guarantees in a system requires support from all levels of the software stack: a *real-time operating system* (RTOS) that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals is needed; library

functions must document their worst-case execution times; dynamic memory allocation may be restricted or disallowed entirely (real-time garbage collectors exist, but the application must still ensure that it doesn't give the GC too much work to do); and an enormous amount of testing and measurement must be done to ensure that guarantees are being met.

All of this requires a large amount of additional work and severely restricts the range of programming languages, libraries, and tools that can be used (since most languages and tools do not provide real-time guarantees). For these reasons, developing real-time systems is very expensive, and they are most commonly used in safety-critical embedded devices. Moreover, "real-time" is not the same as "high-performance"—in fact, real-time systems may have lower throughput, since they have to prioritize timely responses above all else (see also "Latency and Resource Utilization" on page 286).

For most server-side data processing systems, real-time guarantees are simply not economical or appropriate. Consequently, these systems must suffer the pauses and clock instability that come from operating in a non-real-time environment.

### Limiting the impact of garbage collection

The negative effects of process pauses can be mitigated without resorting to expensive real-time scheduling guarantees. Language runtimes have some flexibility around when they schedule garbage collections, because they can track the rate of object allocation and the remaining free memory over time.

An emerging idea is to treat GC pauses like brief planned outages of a node, and to let other nodes handle requests from clients while one node is collecting its garbage. If the runtime can warn the application that a node soon requires a GC pause, the application can stop sending new requests to that node, wait for it to finish processing outstanding requests, and then perform the GC while no requests are in progress. This trick hides GC pauses from clients and reduces the high percentiles of response time [70, 71]. Some latency-sensitive financial trading systems [72] use this approach.

A variant of this idea is to use the garbage collector only for short-lived objects (which are fast to collect) and to restart processes periodically, before they accumulate enough long-lived objects to require a full GC of long-lived objects [65, 73]. One node can be restarted at a time, and traffic can be shifted away from the node before the planned restart, like in a rolling upgrade (see Chapter 4).

These measures cannot fully prevent garbage collection pauses, but they can usefully reduce their impact on the application.

# Knowledge, Truth, and Lies

So far in this chapter we have explored the ways in which distributed systems are different from programs running on a single computer: there is no shared memory, only message passing via an unreliable network with variable delays, and the systems may suffer from partial failures, unreliable clocks, and processing pauses.

The consequences of these issues are profoundly disorienting if you're not used to distributed systems. A node in the network cannot *know* anything for sure—it can only make guesses based on the messages it receives (or doesn't receive) via the network. A node can only find out what state another node is in (what data it has stored, whether it is correctly functioning, etc.) by exchanging messages with it. If a remote node doesn't respond, there is no way of knowing what state it is in, because problems in the network cannot reliably be distinguished from problems at a node.

Discussions of these systems border on the philosophical: What do we know to be true or false in our system? How sure can we be of that knowledge, if the mechanisms for perception and measurement are unreliable? Should software systems obey the laws that we expect of the physical world, such as cause and effect?

Fortunately, we don't need to go as far as figuring out the meaning of life. In a distributed system, we can state the assumptions we are making about the behavior (the *system model*) and design the actual system in such a way that it meets those assumptions. Algorithms can be proved to function correctly within a certain system model. This means that reliable behavior is achievable, even if the underlying system model provides very few guarantees.

However, although it is possible to make software well behaved in an unreliable system model, it is not straightforward to do so. In the rest of this chapter we will further explore the notions of knowledge and truth in distributed systems, which will help us think about the kinds of assumptions we can make and the guarantees we may want to provide. In Chapter 9 we will proceed to look at some examples of distributed algorithms that provide particular guarantees under particular assumptions.

## The Truth Is Defined by the Majority

Imagine a network with an asymmetric fault: a node is able to receive all messages sent to it, but any outgoing messages from that node are dropped or delayed [19]. Even though that node is working perfectly well, and is receiving requests from other nodes, the other nodes cannot hear its responses. After some timeout, the other nodes declare it dead, because they haven't heard from the node. The situation unfolds like a nightmare: the semi-disconnected node is dragged to the graveyard, kicking and screaming "I'm not dead!"—but since nobody can hear its screaming, the funeral procession continues with stoic determination.

In a slightly less nightmarish scenario, the semi-disconnected node may notice that the messages it is sending are not being acknowledged by other nodes, and so realize that there must be a fault in the network. Nevertheless, the node is wrongly declared dead by the other nodes, and the semi-disconnected node cannot do anything about it.

As a third scenario, imagine a node that experiences a long stop-the-world garbage collection pause. All of the node's threads are preempted by the GC and paused for one minute, and consequently, no requests are processed and no responses are sent. The other nodes wait, retry, grow impatient, and eventually declare the node dead and load it onto the hearse. Finally, the GC finishes and the node's threads continue as if nothing had happened. The other nodes are surprised as the supposedly dead node suddenly raises its head out of the coffin, in full health, and starts cheerfully chatting with bystanders. At first, the GCing node doesn't even realize that an entire minute has passed and that it was declared dead—from its perspective, hardly any time has passed since it was last talking to the other nodes.

The moral of these stories is that a node cannot necessarily trust its own judgment of a situation. A distributed system cannot exclusively rely on a single node, because a node may fail at any time, potentially leaving the system stuck and unable to recover. Instead, many distributed algorithms rely on a *quorum*, that is, voting among the nodes (see "Quorums for reading and writing" on page 179): decisions require some minimum number of votes from several nodes in order to reduce the dependence on any one particular node.

That includes decisions about declaring nodes dead. If a quorum of nodes declares another node dead, then it must be considered dead, even if that node still very much feels alive. The individual node must abide by the quorum decision and step down.

Most commonly, the quorum is an absolute majority of more than half the nodes (although other kinds of quorums are possible). A majority quorum allows the system to continue working if individual nodes have failed (with three nodes, one failure can be tolerated; with five nodes, two failures can be tolerated). However, it is still safe, because there can only be only one majority in the system—there cannot be two majorities with conflicting decisions at the same time. We will discuss the use of quorums in more detail when we get to *consensus algorithms* in Chapter 9.

### The leader and the lock

Frequently, a system requires there to be only one of some thing. For example:

- Only one node is allowed to be the leader for a database partition, to avoid split brain (see "Handling Node Outages" on page 156).
- Only one transaction or client is allowed to hold the lock for a particular resource or object, to prevent concurrently writing to it and corrupting it.

- Only one user is allowed to register a particular username, because a username must uniquely identify a user.

Implementing this in a distributed system requires care: even if a node believes that it is "the chosen one" (the leader of the partition, the holder of the lock, the request handler of the user who successfully grabbed the username), that doesn't necessarily mean a quorum of nodes agrees! A node may have formerly been the leader, but if the other nodes declared it dead in the meantime (e.g., due to a network interruption or GC pause), it may have been demoted and another leader may have already been elected.

If a node continues acting as the chosen one, even though the majority of nodes have declared it dead, it could cause problems in a system that is not carefully designed. Such a node could send messages to other nodes in its self-appointed capacity, and if other nodes believe it, the system as a whole may do something incorrect.

For example, Figure 8-4 shows a data corruption bug due to an incorrect implementation of locking. (The bug is not theoretical: HBase used to have this problem [74, 75].) Say you want to ensure that a file in a storage service can only be accessed by one client at a time, because if multiple clients tried to write to it, the file would become corrupted. You try to implement this by requiring a client to obtain a lease from a lock service before accessing the file.

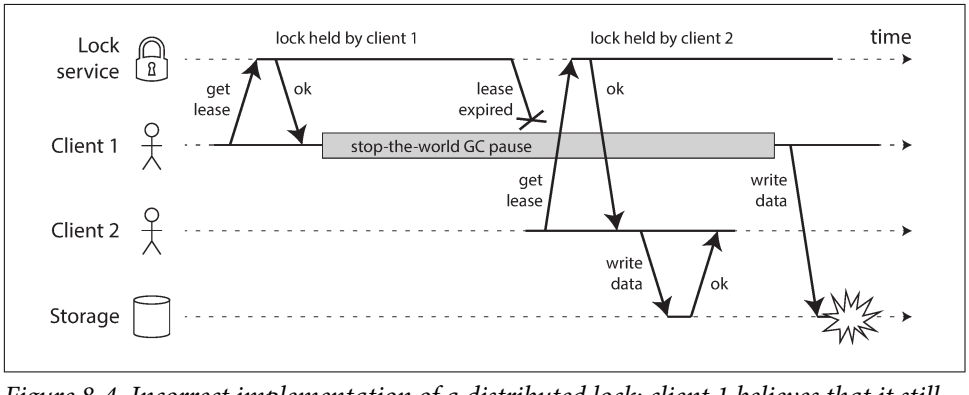



*Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.*

The problem is an example of what we discussed in "Process Pauses" on page 295: if the client holding the lease is paused for too long, its lease expires. Another client can obtain a lease for the same file, and start writing to the file. When the paused client comes back, it believes (incorrectly) that it still has a valid lease and proceeds to also write to the file. As a result, the clients' writes clash and corrupt the file.

### Fencing tokens

When using a lock or lease to protect access to some resource, such as the file storage in Figure 8-4, we need to ensure that a node that is under a false belief of being "the chosen one" cannot disrupt the rest of the system. A fairly simple technique that achieves this goal is called *fencing*, and is illustrated in Figure 8-5.

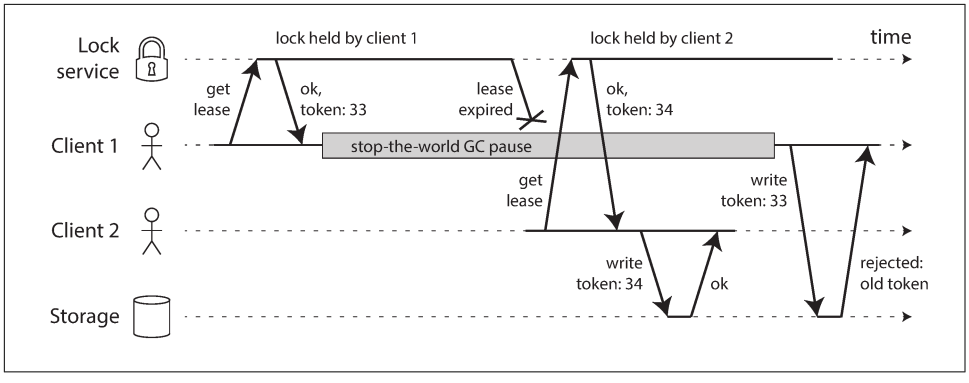



*Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.*

Let's assume that every time the lock server grants a lock or lease, it also returns a *fencing token*, which is a number that increases every time a lock is granted (e.g., incremented by the lock service). We can then require that every time a client sends a write request to the storage service, it must include its current fencing token.

In Figure 8-5, client 1 acquires the lease with a token of 33, but then it goes into a long pause and the lease expires. Client 2 acquires the lease with a token of 34 (the number always increases) and then sends its write request to the storage service, including the token of 34. Later, client 1 comes back to life and sends its write to the storage service, including its token value 33. However, the storage server remembers that it has already processed a write with a higher token number (34), and so it rejects the request with token 33.

If ZooKeeper is used as lock service, the transaction ID `zxid` or the node version `cversion` can be used as fencing token. Since they are guaranteed to be monotonically increasing, they have the required properties [74].

Note that this mechanism requires the resource itself to take an active role in checking tokens by rejecting any writes with an older token than one that has already been processed—it is not sufficient to rely on clients checking their lock status themselves. For resources that do not explicitly support fencing tokens, you might still be able work around the limitation (for example, in the case of a file storage service you could include the fencing token in the filename). However, some kind of check is necessary to avoid processing requests outside of the lock's protection.

Checking a token on the server side may seem like a downside, but it is arguably a good thing: it is unwise for a service to assume that its clients will always be well behaved, because the clients are often run by people whose priorities are very different from the priorities of the people running the service [76]. Thus, it is a good idea for any service to protect itself from accidentally abusive clients.

## Byzantine Faults

Fencing tokens can detect and block a node that is *inadvertently* acting in error (e.g., because it hasn't yet found out that its lease has expired). However, if the node deliberately wanted to subvert the system's guarantees, it could easily do so by sending messages with a fake fencing token.

In this book we assume that nodes are unreliable but honest: they may be slow or never respond (due to a fault), and their state may be outdated (due to a GC pause or network delays), but we assume that if a node *does* respond, it is telling the "truth": to the best of its knowledge, it is playing by the rules of the protocol.

Distributed systems problems become much harder if there is a risk that nodes may "lie" (send arbitrary faulty or corrupted responses)—for example, if a node may claim to have received a particular message when in fact it didn't. Such behavior is known as a *Byzantine fault*, and the problem of reaching consensus in this untrusting environment is known as the *Byzantine Generals Problem* [77].

### The Byzantine Generals Problem

The Byzantine Generals Problem is a generalization of the so-called *Two Generals Problem* [78], which imagines a situation in which two army generals need to agree on a battle plan. As they have set up camp on two different sites, they can only communicate by messenger, and the messengers sometimes get delayed or lost (like packets in a network). We will discuss this problem of *consensus* in Chapter 9.

In the Byzantine version of the problem, there are $n$ generals who need to agree, and their endeavor is hampered by the fact that there are some traitors in their midst. Most of the generals are loyal, and thus send truthful messages, but the traitors may try to deceive and confuse the others by sending fake or untrue messages (while trying to remain undiscovered). It is not known in advance who the traitors are.

Byzantium was an ancient Greek city that later became Constantinople, in the place which is now Istanbul in Turkey. There isn't any historic evidence that the generals of Byzantium were any more prone to intrigue and conspiracy than those elsewhere. Rather, the name is derived from *Byzantine* in the sense of *excessively complicated, bureaucratic, devious*, which was used in politics long before computers [79]. Lamport wanted to choose a nationality that would not offend any readers, and he was advised that calling it *The Albanian Generals Problem* was not such a good idea [80].

A system is *Byzantine fault-tolerant* if it continues to operate correctly even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network. This concern is relevant in certain specific circumstances. For example:

- In aerospace environments, the data in a computer's memory or CPU register could become corrupted by radiation, leading it to respond to other nodes in arbitrarily unpredictable ways. Since a system failure would be very expensive (e.g., an aircraft crashing and killing everyone on board, or a rocket colliding with the International Space Station), flight control systems must tolerate Byzantine faults [81, 82].
- In a system with multiple participating organizations, some participants may attempt to cheat or defraud others. In such circumstances, it is not safe for a node to simply trust another node's messages, since they may be sent with malicious intent. For example, peer-to-peer networks like Bitcoin and other blockchains can be considered to be a way of getting mutually untrusting parties to agree whether a transaction happened or not, without relying on a central authority [83].

However, in the kinds of systems we discuss in this book, we can usually safely assume that there are no Byzantine faults. In your datacenter, all the nodes are controlled by your organization (so they can hopefully be trusted) and radiation levels are low enough that memory corruption is not a major problem. Protocols for making systems Byzantine fault-tolerant are quite complicated [84], and fault-tolerant embedded systems rely on support from the hardware level [81]. In most server-side data systems, the cost of deploying Byzantine fault-tolerant solutions makes them impracticable.

Web applications do need to expect arbitrary and malicious behavior of clients that are under end-user control, such as web browsers. This is why input validation, sanitization, and output escaping are so important: to prevent SQL injection and cross-site scripting, for example. However, we typically don't use Byzantine fault-tolerant protocols here, but simply make the server the authority on deciding what client behavior is and isn't allowed. In peer-to-peer networks, where there is no such central authority, Byzantine fault tolerance is more relevant.

A bug in the software could be regarded as a Byzantine fault, but if you deploy the same software to all nodes, then a Byzantine fault-tolerant algorithm cannot save you. Most Byzantine fault-tolerant algorithms require a supermajority of more than two-thirds of the nodes to be functioning correctly (i.e., if you have four nodes, at most one may malfunction). To use this approach against bugs, you would have to have four independent implementations of the same software and hope that a bug only appears in one of the four implementations.

Similarly, it would be appealing if a protocol could protect us from vulnerabilities, security compromises, and malicious attacks. Unfortunately, this is not realistic either: in most systems, if an attacker can compromise one node, they can probably compromise all of them, because they are probably running the same software. Thus, traditional mechanisms (authentication, access control, encryption, firewalls, and so on) continue to be the main protection against attackers.

### Weak forms of lying

Although we assume that nodes are generally honest, it can be worth adding mechanisms to software that guard against weak forms of "lying"—for example, invalid messages due to hardware issues, software bugs, and misconfiguration. Such protection mechanisms are not full-blown Byzantine fault tolerance, as they would not withstand a determined adversary, but they are nevertheless simple and pragmatic steps toward better reliability. For example:

- Network packets do sometimes get corrupted due to hardware issues or bugs in operating systems, drivers, routers, etc. Usually, corrupted packets are caught by the checksums built into TCP and UDP, but sometimes they evade detection [85, 86, 87]. Simple measures are usually sufficient protection against such corruption, such as checksums in the application-level protocol.
- A publicly accessible application must carefully sanitize any inputs from users, for example checking that a value is within a reasonable range and limiting the size of strings to prevent denial of service through large memory allocations. An internal service behind a firewall may be able to get away with less strict checks on inputs, but some basic sanity-checking of values (e.g., in protocol parsing [85]) is a good idea.
- NTP clients can be configured with multiple server addresses. When synchronizing, the client contacts all of them, estimates their errors, and checks that a majority of servers agree on some time range. As long as most of the servers are okay, a misconfigured NTP server that is reporting an incorrect time is detected as an outlier and is excluded from synchronization [37]. The use of multiple servers makes NTP more robust than if it only uses a single server.

## System Model and Reality

Many algorithms have been designed to solve distributed systems problems—for example, we will examine solutions for the consensus problem in Chapter 9. In order to be useful, these algorithms need to tolerate the various faults of distributed systems that we discussed in this chapter.

Algorithms need to be written in a way that does not depend too heavily on the details of the hardware and software configuration on which they are run. This in

turn requires that we somehow formalize the kinds of faults that we expect to happen in a system. We do this by defining a *system model*, which is an abstraction that describes what things an algorithm may assume.

With regard to timing assumptions, three system models are in common use:

*Synchronous model*
: The synchronous model assumes bounded network delay, bounded process pauses, and bounded clock error. This does not imply exactly synchronized clocks or zero network delay; it just means you know that network delay, pauses, and clock drift will never exceed some fixed upper bound [88]. The synchronous model is not a realistic model of most practical systems, because (as discussed in this chapter) unbounded delays and pauses do occur.

*Partially synchronous model*
: Partial synchrony means that a system behaves like a synchronous system *most of the time*, but it sometimes exceeds the bounds for network delay, process pauses, and clock drift [88]. This is a realistic model of many systems: most of the time, networks and processes are quite well behaved—otherwise we would never be able to get anything done—but we have to reckon with the fact that any timing assumptions may be shattered occasionally. When this happens, network delay, pauses, and clock error may become arbitrarily large.

*Asynchronous model*
: In this model, an algorithm is not allowed to make any timing assumptions—in fact, it does not even have a clock (so it cannot use timeouts). Some algorithms can be designed for the asynchronous model, but it is very restrictive.

Moreover, besides timing issues, we have to consider node failures. The three most common system models for nodes are:

*Crash-stop faults*
: In the crash-stop model, an algorithm may assume that a node can fail in only one way, namely by crashing. This means that the node may suddenly stop responding at any moment, and thereafter that node is gone forever—it never comes back.

*Crash-recovery faults*
: We assume that nodes may crash at any moment, and perhaps start responding again after some unknown time. In the crash-recovery model, nodes are assumed to have stable storage (i.e., nonvolatile disk storage) that is preserved across crashes, while the in-memory state is assumed to be lost.

*Byzantine (arbitrary) faults*
: Nodes may do absolutely anything, including trying to trick and deceive other nodes, as described in the last section.

For modeling real systems, the partially synchronous model with crash-recovery faults is generally the most useful model. But how do distributed algorithms cope with that model?

### Correctness of an algorithm

To define what it means for an algorithm to be *correct*, we can describe its *properties*. For example, the output of a sorting algorithm has the property that for any two distinct elements of the output list, the element further to the left is smaller than the element further to the right. That is simply a formal way of defining what it means for a list to be sorted.

Similarly, we can write down the properties we want of a distributed algorithm to define what it means to be correct. For example, if we are generating fencing tokens for a lock (see "Fencing tokens" on page 303), we may require the algorithm to have the following properties:

*Uniqueness*
: No two requests for a fencing token return the same value.

*Monotonic sequence*
: If request $x$ returned token $t_x$, and request $y$ returned token $t_y$, and $x$ completed before $y$ began, then $t_x < t_y$.

*Availability*
: A node that requests a fencing token and does not crash eventually receives a response.

An algorithm is correct in some system model if it always satisfies its properties in all situations that we assume may occur in that system model. But how does this make sense? If all nodes crash, or all network delays suddenly become infinitely long, then no algorithm will be able to get anything done.

### Safety and liveness

To clarify the situation, it is worth distinguishing between two different kinds of properties: *safety* and *liveness* properties. In the example just given, *uniqueness* and *monotonic sequence* are safety properties, but *availability* is a liveness property.

What distinguishes the two kinds of properties? A giveaway is that liveness properties often include the word "eventually" in their definition. (And yes, you guessed it—*eventual consistency* is a liveness property [89].)

Safety is often informally defined as *nothing bad happens*, and liveness as *something good eventually happens*. However, it's best to not read too much into those informal definitions, because the meaning of good and bad is subjective. The actual definitions of safety and liveness are precise and mathematical [90]:

- If a safety property is violated, we can point at a particular point in time at which it was broken (for example, if the uniqueness property was violated, we can identify the particular operation in which a duplicate fencing token was returned). After a safety property has been violated, the violation cannot be undone—the damage is already done.
- A liveness property works the other way round: it may not hold at some point in time (for example, a node may have sent a request but not yet received a response), but there is always hope that it may be satisfied in the future (namely by receiving a response).

An advantage of distinguishing between safety and liveness properties is that it helps us deal with difficult system models. For distributed algorithms, it is common to require that safety properties *always* hold, in all possible situations of a system model [88]. That is, even if all nodes crash, or the entire network fails, the algorithm must nevertheless ensure that it does not return a wrong result (i.e., that the safety properties remain satisfied).

However, with liveness properties we are allowed to make caveats: for example, we could say that a request needs to receive a response only if a majority of nodes have not crashed, and only if the network eventually recovers from an outage. The definition of the partially synchronous model requires that eventually the system returns to a synchronous state—that is, any period of network interruption lasts only for a finite duration and is then repaired.

### Mapping system models to the real world

Safety and liveness properties and system models are very useful for reasoning about the correctness of a distributed algorithm. However, when implementing an algorithm in practice, the messy facts of reality come back to bite you again, and it becomes clear that the system model is a simplified abstraction of reality.

For example, algorithms in the crash-recovery model generally assume that data in stable storage survives crashes. However, what happens if the data on disk is corrupted, or the data is wiped out due to hardware error or misconfiguration [91]? What happens if a server has a firmware bug and fails to recognize its hard drives on reboot, even though the drives are correctly attached to the server [92]?

Quorum algorithms (see "Quorums for reading and writing" on page 179) rely on a node remembering the data that it claims to have stored. If a node may suffer from amnesia and forget previously stored data, that breaks the quorum condition, and thus breaks the correctness of the algorithm. Perhaps a new system model is needed, in which we assume that stable storage mostly survives crashes, but may sometimes be lost. But that model then becomes harder to reason about.

The theoretical description of an algorithm can declare that certain things are simply assumed not to happen—and in non-Byzantine systems, we do have to make some assumptions about faults that can and cannot happen. However, a real implementation may still have to include code to handle the case where something happens that was assumed to be impossible, even if that handling boils down to `printf("Sucks to be you")` and `exit(666)`—i.e., letting a human operator clean up the mess [93]. (This is arguably the difference between computer science and software engineering.)

That is not to say that theoretical, abstract system models are worthless—quite the opposite. They are incredibly helpful for distilling down the complexity of real systems to a manageable set of faults that we can reason about, so that we can understand the problem and try to solve it systematically. We can prove algorithms correct by showing that their properties always hold in some system model.

Proving an algorithm correct does not mean its *implementation* on a real system will necessarily always behave correctly. But it's a very good first step, because the theoretical analysis can uncover problems in an algorithm that might remain hidden for a long time in a real system, and that only come to bite you when your assumptions (e.g., about timing) are defeated due to unusual circumstances. Theoretical analysis and empirical testing are equally important.

# Summary

In this chapter we have discussed a wide range of problems that can occur in distributed systems, including:

- Whenever you try to send a packet over the network, it may be lost or arbitrarily delayed. Likewise, the reply may be lost or delayed, so if you don't get a reply, you have no idea whether the message got through.
- A node's clock may be significantly out of sync with other nodes (despite your best efforts to set up NTP), it may suddenly jump forward or back in time, and relying on it is dangerous because you most likely don't have a good measure of your clock's confidence interval.
- A process may pause for a substantial amount of time at any point in its execution (perhaps due to a stop-the-world garbage collector), be declared dead by other nodes, and then come back to life again without realizing that it was paused.

The fact that such *partial failures* can occur is the defining characteristic of distributed systems. Whenever software tries to do anything involving other nodes, there is the possibility that it may occasionally fail, or randomly go slow, or not respond at all (and eventually time out). In distributed systems, we try to build toler-

ance of partial failures into software, so that the system as a whole may continue functioning even when some of its constituent parts are broken.

To tolerate faults, the first step is to *detect* them, but even that is hard. Most systems don't have an accurate mechanism of detecting whether a node has failed, so most distributed algorithms rely on timeouts to determine whether a remote node is still available. However, timeouts can't distinguish between network and node failures, and variable network delay sometimes causes a node to be falsely suspected of crashing. Moreover, sometimes a node can be in a degraded state: for example, a Gigabit network interface could suddenly drop to 1 Kb/s throughput due to a driver bug [94]. Such a node that is "limping" but not dead can be even more difficult to deal with than a cleanly failed node.

Once a fault is detected, making a system tolerate it is not easy either: there is no global variable, no shared memory, no common knowledge or any other kind of shared state between the machines. Nodes can't even agree on what time it is, let alone on anything more profound. The only way information can flow from one node to another is by sending it over the unreliable network. Major decisions cannot be safely made by a single node, so we require protocols that enlist help from other nodes and try to get a quorum to agree.

If you're used to writing software in the idealized mathematical perfection of a single computer, where the same operation always deterministically returns the same result, then moving to the messy physical reality of distributed systems can be a bit of a shock. Conversely, distributed systems engineers will often regard a problem as trivial if it can be solved on a single computer [5], and indeed a single computer can do a lot nowadays [95]. If you can avoid opening Pandora's box and simply keep things on a single machine, it is generally worth doing so.

However, as discussed in the introduction to Part II, scalability is not the only reason for wanting to use a distributed system. Fault tolerance and low latency (by placing data geographically close to users) are equally important goals, and those things cannot be achieved with a single node.

In this chapter we also went on some tangents to explore whether the unreliability of networks, clocks, and processes is an inevitable law of nature. We saw that it isn't: it is possible to give hard real-time response guarantees and bounded delays in networks, but doing so is very expensive and results in lower utilization of hardware resources. Most non-safety-critical systems choose cheap and unreliable over expensive and reliable.

We also touched on supercomputers, which assume reliable components and thus have to be stopped and restarted entirely when a component does fail. By contrast, distributed systems can run forever without being interrupted at the service level, because all faults and maintenance can be handled at the node level—at least in

theory. (In practice, if a bad configuration change is rolled out to all nodes, that will still bring a distributed system to its knees.)

This chapter has been all about problems, and has given us a bleak outlook. In the next chapter we will move on to solutions, and discuss some algorithms that have been designed to cope with the problems in distributed systems.

## References


[1] Mark Cavage: "There's Just No Getting Around It: You're Building a Distributed System," *ACM Queue*, volume 11, number 4, pages 80-89, April 2013. doi:10.1145/2466486.2482856

[2] Jay Kreps: "Getting Real About Distributed System Reliability," *blog.empathybox.com*, March 19, 2012.

[3] Sydney Padua: *The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer*. Particular Books, April 2015. ISBN: 978-0-141-98151-2

[4] Coda Hale: "You Can't Sacrifice Partition Tolerance," *codahale.com*, October 7, 2010.

[5] Jeff Hodges: "Notes on Distributed Systems for Young Bloods," *somethingsimilar.com*, January 14, 2013.

[6] Antonio Regalado: "Who Coined 'Cloud Computing'?," *technologyreview.com*, October 31, 2011.

[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle: "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition," *Synthesis Lectures on Computer Architecture*, volume 8, number 3, Morgan & Claypool Publishers, July 2013. doi:10.2200/S00516ED2V01Y201306CAC024, ISBN: 978-1-627-05010-4

[8] David Fiala, Frank Mueller, Christian Engelmann, et al.: "Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing," at *International Conference for High Performance Computing, Networking, Storage and Analysis* (SC12), November 2012.

[9] Arjun Singh, Joon Ong, Amit Agarwal, et al.: "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," at *Annual Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM), August 2015. doi:10.1145/2785956.2787508

[10] Glenn K. Lockwood: "Hadoop's Uncomfortable Fit in HPC," *glennklockwood.blogspot.co.uk*, May 16, 2014.

[11] John von Neumann: "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies (AM-34)*, edited by Claude E. Shannon and John McCarthy, Princeton University Press, 1956. ISBN: 978-0-691-07916-5

[12] Richard W. Hamming: *The Art of Doing Science and Engineering*. Taylor & Francis, 1997. ISBN: 978-9-056-99500-3

[13] Claude E. Shannon: "A Mathematical Theory of Communication," *The Bell System Technical Journal*, volume 27, number 3, pages 379–423 and 623–656, July 1948.

[14] Peter Bailis and Kyle Kingsbury: "The Network Is Reliable," *ACM Queue*, volume 12, number 7, pages 48-55, July 2014. doi:10.1145/2639988.2639988

[15] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish: "Taming Uncertainty in Distributed Systems with Help from the Network," at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741976

[16] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan: "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," at *ACM SIGCOMM Conference*, August 2011. doi:10.1145/2018436.2018477

[17] Mark Imbriaco: "Downtime Last Saturday," *github.com*, December 26, 2012.

[18] Will Oremus: "The Global Internet Is Being Attacked by Sharks, Google Confirms," *slate.com*, August 15, 2014.

[19] Marc A. Donges: "Re: bnx2 cards Intermittantly Going Offline," Message to Linux *netdev* mailing list, *spinics.net*, September 13, 2012.

[20] Kyle Kingsbury: "Call Me Maybe: Elasticsearch," *aphyr.com*, June 15, 2014.

[21] Salvatore Sanfilippo: "A Few Arguments About Redis Sentinel Properties and Fail Scenarios," *antirez.com*, October 21, 2014.

[22] Bert Hubert: "The Ultimate SO_LINGER Page, or: Why Is My TCP Not Reliable," *blog.netherlabs.nl*, January 18, 2009.

[23] Nicolas Liochon: "CAP: If All You Have Is a Timeout, Everything Looks Like a Partition," *blog.thislongrun.com*, May 25, 2015.

[24] Jerome H. Saltzer, David P. Reed, and David D. Clark: "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, volume 2, number 4, pages 277–288, November 1984. doi:10.1145/357401.357402

[25] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, et al.: "Queues Don't Matter When You Can JUMP Them!," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

[26] Guohui Wang and T. S. Eugene Ng: "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," at *29th IEEE International Conference on Computer Communications* (INFOCOM), March 2010. doi:10.1109/INFCOM.2010.5461931

[27] Van Jacobson: "Congestion Avoidance and Control," at *ACM Symposium on Communications Architectures and Protocols* (SIGCOMM), August 1988. doi:10.1145/52324.52356

[28] Brandon Philips: "etcd: Distributed Locking and Service Discovery," at *Strange Loop*, September 2014.

[29] Steve Newman: "A Systematic Look at EC2 I/O," *blog.scalyr.com*, October 16, 2012.

[30] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama: "The ϕ Accrual Failure Detector," Japan Advanced Institute of Science and Technology, School of Information Science, Technical Report IS-RR-2004-010, May 2004.

[31] Jeffrey Wang: "Phi Accrual Failure Detector," *ternarysearch.blogspot.co.uk*, August 11, 2013.

[32] Srinivasan Keshav: *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley Professional, May 1997. ISBN: 978-0-201-63442-6

[33] Cisco, "Integrated Services Digital Network," *docwiki.cisco.com*.

[34] Othmar Kyas: *ATM Networks*. International Thomson Publishing, 1995. ISBN: 978-1-850-32128-6

[35] "InfiniBand FAQ," Mellanox Technologies, December 22, 2014.

[36] Jose Renato Santos, Yoshio Turner, and G. (John) Janakiraman: "End-to-End Congestion Control for InfiniBand," at *22nd Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM), April 2003. Also published by HP Laboratories Palo Alto, Tech Report HPL-2002-359. doi:10.1109/INFCOM.2003.1208949

[37] Ulrich Windl, David Dalton, Marc Martinec, and Dale R. Worley: "The NTP FAQ and HOWTO," *ntp.org*, November 2006.

[38] John Graham-Cumming: "How and why the leap second affected Cloudflare DNS," *blog.cloudflare.com*, January 1, 2017.

[39] David Holmes: "Inside the Hotspot VM: Clocks, Timers and Scheduling Events – Part I – Windows," *blogs.oracle.com*, October 2, 2006.

[40] Steve Loughran: "Time on Multi-Core, Multi-Socket Servers," *stevelough-ran.blogspot.co.uk*, September 17, 2015.

[41] James C. Corbett, Jeffrey Dean, Michael Epstein, et al.: "Spanner: Google's Globally-Distributed Database," at *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.

[42] M. Caporaloni and R. Ambrosini: "How Closely Can a Personal Computer Clock Track the UTC Timescale Via the Internet?," *European Journal of Physics*, volume 23, number 4, pages L17–L21, June 2012. doi:10.1088/0143-0807/23/4/103

[43] Nelson Minar: "A Survey of the NTP Network," *alumni.media.mit.edu*, December 1999.

[44] Viliam Holub: "Synchronizing Clocks in a Cassandra Cluster Pt. 1 – The Problem," *blog.rapid7.com*, March 14, 2014.

[45] Poul-Henning Kamp: "The One-Second War (What Time Will You Die?)," *ACM Queue*, volume 9, number 4, pages 44–48, April 2011. doi:10.1145/1966989.1967009

[46] Nelson Minar: "Leap Second Crashes Half the Internet," *somebits.com*, July 3, 2012.

[47] Christopher Pascoe: "Time, Technology and Leaping Seconds," *googleblog.blogspot.co.uk*, September 15, 2011.

[48] Mingxue Zhao and Jeff Barr: "Look Before You Leap – The Coming Leap Second and AWS," *aws.amazon.com*, May 18, 2015.

[49] Darryl Veitch and Kanthaiah Vijayalayan: "Network Timing and the 2015 Leap Second," at *17th International Conference on Passive and Active Measurement* (PAM), April 2016. doi:10.1007/978-3-319-30505-9_29

[50] "Timekeeping in VMware Virtual Machines," Information Guide, VMware, Inc., December 2011.

[51] "MiFID II / MiFIR: Regulatory Technical and Implementing Standards – Annex I (Draft)," European Securities and Markets Authority, Report ESMA/2015/1464, September 2015.

[52] Luke Bigum: "Solving MiFID II Clock Synchronisation With Minimum Spend (Part 1)," *lmax.com*, November 27, 2015.

[53] Kyle Kingsbury: "Call Me Maybe: Cassandra," *aphyr.com*, September 24, 2013.

[54] John Daily: "Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems," *riak.com*, November 12, 2013.

[55] Kyle Kingsbury: "The Trouble with Timestamps," *aphyr.com*, October 12, 2013.

[56] Leslie Lamport: "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563

[57] Sandeep Kulkarni, Murat Demirbas, Deepak Madeppa, et al.: "Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases," State University of New York at Buffalo, Computer Science and Engineering Technical Report 2014-04, May 2014.

[58] Justin Sheehy: "There Is No Now: Problems With Simultaneity in Distributed Systems," *ACM Queue*, volume 13, number 3, pages 36–41, March 2015. doi:10.1145/2733108

[59] Murat Demirbas: "Spanner: Google's Globally-Distributed Database," *muratbuffalo.blogspot.co.uk*, July 4, 2013.

[60] Dahlia Malkhi and Jean-Philippe Martin: "Spanner's Concurrency Control," *ACM SIGACT News*, volume 44, number 3, pages 73–77, September 2013. doi:10.1145/2527748.2527767

[61] Manuel Bravo, Nuno Diegues, Jingna Zeng, et al.: "On the Use of Clocks to Enforce Consistency in the Cloud," *IEEE Data Engineering Bulletin*, volume 38, number 1, pages 18–31, March 2015.

[62] Spencer Kimball: "Living Without Atomic Clocks," *cockroachlabs.com*, February 17, 2016.

[63] Cary G. Gray and David R. Cheriton: "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," at *12th ACM Symposium on Operating Systems Principles* (SOSP), December 1989. doi:10.1145/74850.74870

[64] Todd Lipcon: "Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1," *blog.cloudera.com*, February 24, 2011.

[65] Martin Thompson: "Java Garbage Collection Distilled," *mechanical-sympathy.blogspot.co.uk*, July 16, 2013.

[66] Alexey Ragozin: "How to Tame Java GC Pauses? Surviving 16GiB Heap and Greater," *dzone.com*, June 28, 2011.

[67] Christopher Clark, Keir Fraser, Steven Hand, et al.: "Live Migration of Virtual Machines," at *2nd USENIX Symposium on Symposium on Networked Systems Design & Implementation* (NSDI), May 2005.

[68] Mike Shaver: "fsyncers and Curveballs," *shaver.off.net*, May 25, 2008.

[69] Zhenyun Zhuang and Cuong Tran: "Eliminating Large JVM GC Pauses Caused by Background IO Traffic," *engineering.linkedin.com*, February 10, 2016.

[70] David Terei and Amit Levy: "Blade: A Data Center Garbage Collector," arXiv: 1504.02578, April 13, 2015.

[71] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz: "Trash Day: Coordinating Garbage Collection in Distributed Systems," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

[72] "Predictable Low Latency," Cinnober Financial Technology AB, *cinnober.com*, November 24, 2013.

[73] Martin Fowler: "The LMAX Architecture," *martinfowler.com*, July 12, 2011.

[74] Flavio P. Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013. ISBN: 978-1-449-36130-3

[75] Enis Söztutar: "HBase and HDFS: Understanding Filesystem Usage in HBase," at *HBaseCon*, June 2013.

[76] Caitie McCaffrey: "Clients Are Jerks: AKA How Halo 4 DoSed the Services at Launch & How We Survived," *caitiem.com*, June 23, 2015.

[77] Leslie Lamport, Robert Shostak, and Marshall Pease: "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 4, number 3, pages 382–401, July 1982. doi:10.1145/357172.357176

[78] Jim N. Gray: "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, volume 60, edited by R. Bayer, R. M. Graham, and G. Seegmüller, pages 393–481, Springer-Verlag, 1978. ISBN: 978-3-540-08755-7

[79] Brian Palmer: "How Complicated Was the Byzantine Empire?," *slate.com*, October 20, 2011.

[80] Leslie Lamport: "My Writings," *lamport.azurewebsites.net*, December 16, 2014. This page can be found by searching the web for the 23-character string obtained by removing the hyphens from the string `allla-mport-spubso-ntheweb`.

[81] John Rushby: "Bus Architectures for Safety-Critical Embedded Systems," at *1st International Workshop on Embedded Software* (EMSOFT), October 2001.

[82] Jake Edge: "ELC: SpaceX Lessons Learned," *lwn.net*, March 6, 2013.

[83] Andrew Miller and Joseph J. LaViola, Jr.: "Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin," University of Central Florida, Technical Report CS-TR-14-01, April 2014.

[84] James Mickens: "The Saddest Moment," *USENIX ;login: logout*, May 2013.

[85] Evan Gilman: "The Discovery of Apache ZooKeeper's Poison Packet," *pagerduty.com*, May 7, 2015.

[86] Jonathan Stone and Craig Partridge: "When the CRC and TCP Checksum Disagree," at *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (SIGCOMM), August 2000. doi:10.1145/347059.347561

[87] Evan Jones: "How Both TCP and Ethernet Checksums Fail," *evanjones.ca*, October 5, 2015.

[88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:10.1145/42282.42283

[89] Peter Bailis and Ali Ghodsi: "Eventual Consistency Today: Limitations, Extensions, and Beyond," *ACM Queue*, volume 11, number 3, pages 55-63, March 2013. doi:10.1145/2460276.2462076

[90] Bowen Alpern and Fred B. Schneider: "Defining Liveness," *Information Processing Letters*, volume 21, number 4, pages 181–185, October 1985. doi:10.1016/0020-0190(85)90056-0

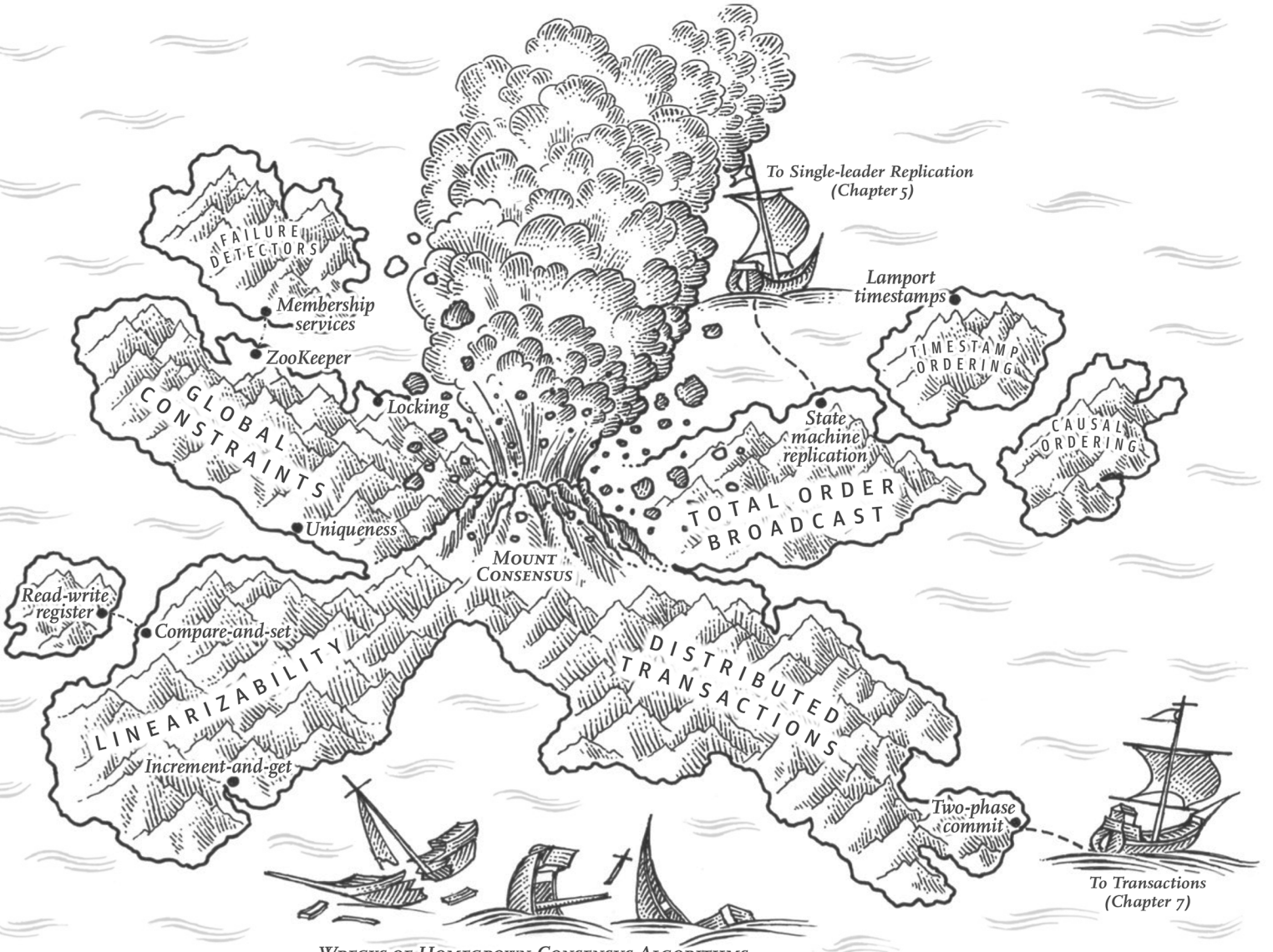
[91] Flavio P. Junqueira: "Dude, Where's My Metadata?," *fpj.me*, May 28, 2015.

[92] Scott Sanders: "January 28th Incident Report," *github.com*, February 3, 2016.

[93] Jay Kreps: "A Few Notes on Kafka and Jepsen," *blog.empathybox.com*, September 25, 2013.

[94] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, et al.: "Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems," at *4th ACM Symposium on Cloud Computing* (SoCC), October 2013. doi:10.1145/2523616.2523627

[95] Frank McSherry, Michael Isard, and Derek G. Murray: "Scalability! But at What COST?," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.




*Wrecks of Homegrown Consensus Algorithms*

# CHAPTER 9
# Consistency and Consensus

> *Is it better to be alive and wrong or right and dead?*
> —Jay Kreps, *A Few Notes on Kafka and Jepsen* (2013)

Lots of things can go wrong in distributed systems, as discussed in Chapter 8. The simplest way of handling such faults is to simply let the entire service fail, and show the user an error message. If that solution is unacceptable, we need to find ways of *tolerating* faults—that is, of keeping the service functioning correctly, even if some internal component is faulty.

In this chapter, we will talk about some examples of algorithms and protocols for building fault-tolerant distributed systems. We will assume that all the problems from Chapter 8 can occur: packets can be lost, reordered, duplicated, or arbitrarily delayed in the network; clocks are approximate at best; and nodes can pause (e.g., due to garbage collection) or crash at any time.

The best way of building fault-tolerant systems is to find some general-purpose abstractions with useful guarantees, implement them once, and then let applications rely on those guarantees. This is the same approach as we used with transactions in Chapter 7: by using a transaction, the application can pretend that there are no crashes (atomicity), that nobody else is concurrently accessing the database (isolation), and that storage devices are perfectly reliable (durability). Even though crashes, race conditions, and disk failures do occur, the transaction abstraction hides those problems so that the application doesn't need to worry about them.

We will now continue along the same lines, and seek abstractions that can allow an application to ignore some of the problems with distributed systems. For example, one of the most important abstractions for distributed systems is *consensus*: that is, getting all of the nodes to agree on something. As we shall see in this chapter, reliably

reaching consensus in spite of network faults and process failures is a surprisingly tricky problem.

Once you have an implementation of consensus, applications can use it for various purposes. For example, say you have a database with single-leader replication. If the leader dies and you need to fail over to another node, the remaining database nodes can use consensus to elect a new leader. As discussed in "Handling Node Outages" on page 156, it's important that there is only one leader, and that all nodes agree who the leader is. If two nodes both believe that they are the leader, that situation is called *split brain*, and it often leads to data loss. Correct implementations of consensus help avoid such problems.

Later in this chapter, in "Distributed Transactions and Consensus" on page 352, we will look into algorithms to solve consensus and related problems. But we first need to explore the range of guarantees and abstractions that can be provided in a distributed system.

We need to understand the scope of what can and cannot be done: in some situations, it's possible for the system to tolerate faults and continue working; in other situations, that is not possible. The limits of what is and isn't possible have been explored in depth, both in theoretical proofs and in practical implementations. We will get an overview of those fundamental limits in this chapter.

Researchers in the field of distributed systems have been studying these topics for decades, so there is a lot of material—we'll only be able to scratch the surface. In this book we don't have space to go into details of the formal models and proofs, so we will stick with informal intuitions. The literature references offer plenty of additional depth if you're interested.

## Consistency Guarantees

In "Problems with Replication Lag" on page 161 we looked at some timing issues that occur in a replicated database. If you look at two database nodes at the same moment in time, you're likely to see different data on the two nodes, because write requests arrive on different nodes at different times. These inconsistencies occur no matter what replication method the database uses (single-leader, multi-leader, or leaderless replication).

Most replicated databases provide at least *eventual consistency*, which means that if you stop writing to the database and wait for some unspecified length of time, then eventually all read requests will return the same value [1]. In other words, the inconsistency is temporary, and it eventually resolves itself (assuming that any faults in the network are also eventually repaired). A better name for eventual consistency may be *convergence*, as we expect all replicas to eventually converge to the same value [2].

However, this is a very weak guarantee—it doesn't say anything about *when* the replicas will converge. Until the time of convergence, reads could return anything or nothing [1]. For example, if you write a value and then immediately read it again, there is no guarantee that you will see the value you just wrote, because the read may be routed to a different replica (see "Reading Your Own Writes" on page 162).

Eventual consistency is hard for application developers because it is so different from the behavior of variables in a normal single-threaded program. If you assign a value to a variable and then read it shortly afterward, you don't expect to read back the old value, or for the read to fail. A database looks superficially like a variable that you can read and write, but in fact it has much more complicated semantics [3].

When working with a database that provides only weak guarantees, you need to be constantly aware of its limitations and not accidentally assume too much. Bugs are often subtle and hard to find by testing, because the application may work well most of the time. The edge cases of eventual consistency only become apparent when there is a fault in the system (e.g., a network interruption) or at high concurrency.

In this chapter we will explore stronger consistency models that data systems may choose to provide. They don't come for free: systems with stronger guarantees may have worse performance or be less fault-tolerant than systems with weaker guarantees. Nevertheless, stronger guarantees can be appealing because they are easier to use correctly. Once you have seen a few different consistency models, you'll be in a better position to decide which one best fits your needs.

There is some similarity between distributed consistency models and the hierarchy of transaction isolation levels we discussed previously [4, 5] (see "Weak Isolation Levels" on page 233). But while there is some overlap, they are mostly independent concerns: transaction isolation is primarily about avoiding race conditions due to concurrently executing transactions, whereas distributed consistency is mostly about coordinating the state of replicas in the face of delays and faults.

This chapter covers a broad range of topics, but as we shall see, these areas are in fact deeply linked:

- We will start by looking at one of the strongest consistency models in common use, *linearizability*, and examine its pros and cons.
- We'll then examine the issue of ordering events in a distributed system ("Ordering Guarantees" on page 339), particularly around causality and total ordering.
- In the third section ("Distributed Transactions and Consensus" on page 352) we will explore how to atomically commit a distributed transaction, which will finally lead us toward solutions for the consensus problem.

# Linearizability

In an eventually consistent database, if you ask two different replicas the same question at the same time, you may get two different answers. That's confusing. Wouldn't it be a lot simpler if the database could give the illusion that there is only one replica (i.e., only one copy of the data)? Then every client would have the same view of the data, and you wouldn't have to worry about replication lag.

This is the idea behind *linearizability* [6] (also known as *atomic consistency* [7], *strong consistency*, *immediate consistency*, or *external consistency* [8]). The exact definition of linearizability is quite subtle, and we will explore it in the rest of this section. But the basic idea is to make a system appear as if there were only one copy of the data, and all operations on it are atomic. With this guarantee, even though there may be multiple replicas in reality, the application does not need to worry about them.

In a linearizable system, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written. Maintaining the illusion of a single copy of the data means guaranteeing that the value read is the most recent, up-to-date value, and doesn't come from a stale cache or replica. In other words, linearizability is a *recency guarantee*. To clarify this idea, let's look at an example of a system that is not linearizable.

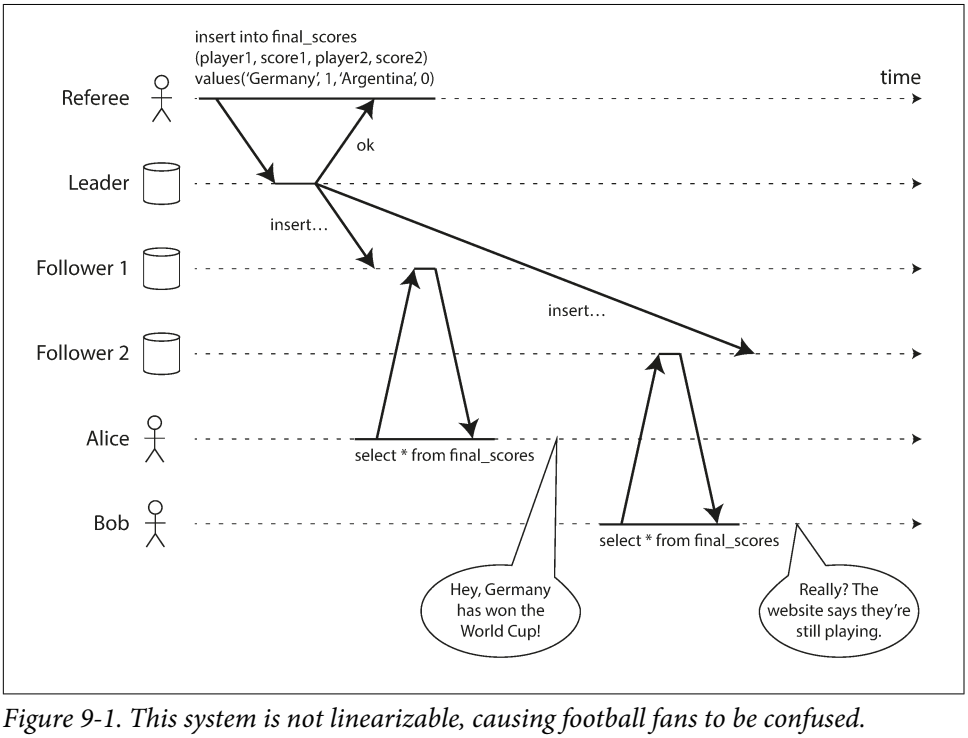



*Figure 9-1. This system is not linearizable, causing football fans to be confused.*

Figure 9-1 shows an example of a nonlinearizable sports website [9]. Alice and Bob are sitting in the same room, both checking their phones to see the outcome of the 2014 FIFA World Cup final. Just after the final score is announced, Alice refreshes the page, sees the winner announced, and excitedly tells Bob about it. Bob incredulously hits *reload* on his own phone, but his request goes to a database replica that is lagging, and so his phone shows that the game is still ongoing.

If Alice and Bob had hit reload at the same time, it would have been less surprising if they had gotten two different query results, because they wouldn't know at exactly what time their respective requests were processed by the server. However, Bob knows that he hit the reload button (initiated his query) *after* he heard Alice exclaim the final score, and therefore he expects his query result to be at least as recent as Alice's. The fact that his query returned a stale result is a violation of linearizability.

## What Makes a System Linearizable?

The basic idea behind linearizability is simple: to make a system appear as if there is only a single copy of the data. However, nailing down precisely what that means actually requires some care. In order to understand linearizability better, let's look at some more examples.

Figure 9-2 shows three clients concurrently reading and writing the same object $x$ in a linearizable database. In the distributed systems literature, $x$ is called a *register*—in practice, it could be one key in a key-value store, one row in a relational database, or one document in a document database, for example.

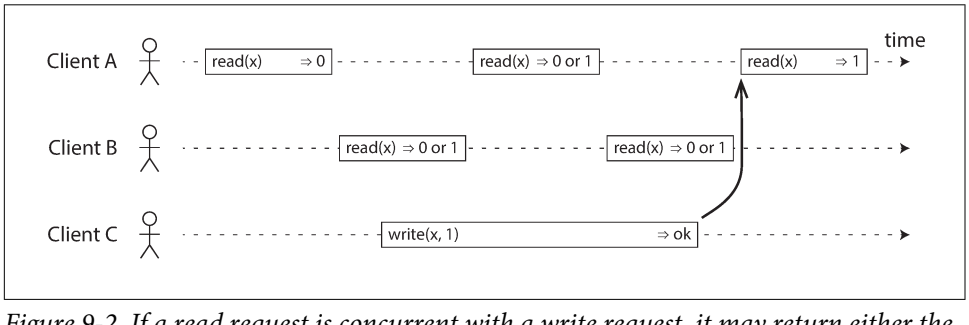



*Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.*

For simplicity, Figure 9-2 shows only the requests from the clients' point of view, not the internals of the database. Each bar is a request made by a client, where the start of a bar is the time when the request was sent, and the end of a bar is when the response was received by the client. Due to variable network delays, a client doesn't know

exactly when the database processed its request—it only knows that it must have happened sometime between the client sending the request and receiving the response.[i]

In this example, the register has two types of operations:

- *read*($x$) ⇒ $v$ means the client requested to read the value of register $x$, and the database returned the value $v$.
- *write*($x$, $v$) ⇒ $r$ means the client requested to set the register $x$ to value $v$, and the database returned response $r$ (which could be *ok* or *error*).

In Figure 9-2, the value of $x$ is initially 0, and client C performs a write request to set it to 1. While this is happening, clients A and B are repeatedly polling the database to read the latest value. What are the possible responses that A and B might get for their read requests?

- The first read operation by client A completes before the write begins, so it must definitely return the old value 0.
- The last read by client A begins after the write has completed, so it must definitely return the new value 1 if the database is linearizable: we know that the write must have been processed sometime between the start and end of the write operation, and the read must have been processed sometime between the start and end of the read operation. If the read started after the write ended, then the read must have been processed after the write, and therefore it must see the new value that was written.
- Any read operations that overlap in time with the write operation might return either 0 or 1, because we don't know whether or not the write has taken effect at the time when the read operation is processed. These operations are *concurrent* with the write.

However, that is not yet sufficient to fully describe linearizability: if reads that are concurrent with a write can return either the old or the new value, then readers could see a value flip back and forth between the old and the new value several times while a write is going on. That is not what we expect of a system that emulates a "single copy of the data."[ii]

i. A subtle detail of this diagram is that it assumes the existence of a global clock, represented by the horizontal axis. Even though real systems typically don't have accurate clocks (see "Unreliable Clocks" on page 287), this assumption is okay: for the purposes of analyzing a distributed algorithm, we may pretend that an accurate global clock exists, as long as the algorithm doesn't have access to it [47]. Instead, the algorithm can only see a mangled approximation of real time, as produced by a quartz oscillator and NTP.

ii. A register in which reads may return either the old or the new value if they are concurrent with a write is known as a *regular register* [7, 25].

To make the system linearizable, we need to add another constraint, illustrated in Figure 9-3.

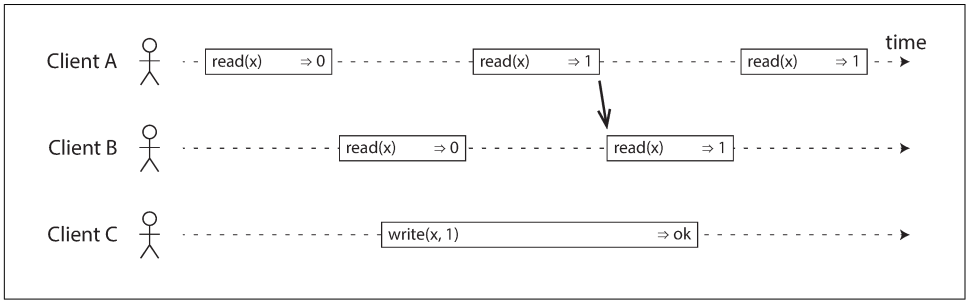



*Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.*

In a linearizable system we imagine that there must be some point in time (between the start and end of the write operation) at which the value of $x$ atomically flips from 0 to 1. Thus, if one client's read returns the new value 1, all subsequent reads must also return the new value, even if the write operation has not yet completed.

This timing dependency is illustrated with an arrow in Figure 9-3. Client A is the first to read the new value, 1. Just after A's read returns, B begins a new read. Since B's read occurs strictly after A's read, it must also return 1, even though the write by C is still ongoing. (It's the same situation as with Alice and Bob in Figure 9-1: after Alice has read the new value, Bob also expects to read the new value.)

We can further refine this timing diagram to visualize each operation taking effect atomically at some point in time. A more complex example is shown in Figure 9-4 [10].

In Figure 9-4 we add a third type of operation besides *read* and *write*:

- $cas(x, v_{old}, v_{new}) \Rightarrow r$ means the client requested an atomic *compare-and-set* operation (see "Compare-and-set" on page 245). If the current value of the register $x$ equals $v_{old}$, it should be atomically set to $v_{new}$. If $x \neq v_{old}$ then the operation should leave the register unchanged and return an error. $r$ is the database's response (*ok* or *error*).

Each operation in Figure 9-4 is marked with a vertical line (inside the bar for each operation) at the time when we think the operation was executed. Those markers are joined up in a sequential order, and the result must be a valid sequence of reads and writes for a register (every read must return the value set by the most recent write).

The requirement of linearizability is that the lines joining up the operation markers always move forward in time (from left to right), never backward. This requirement

ensures the recency guarantee we discussed earlier: once a new value has been written or read, all subsequent reads see the value that was written, until it is overwritten again.

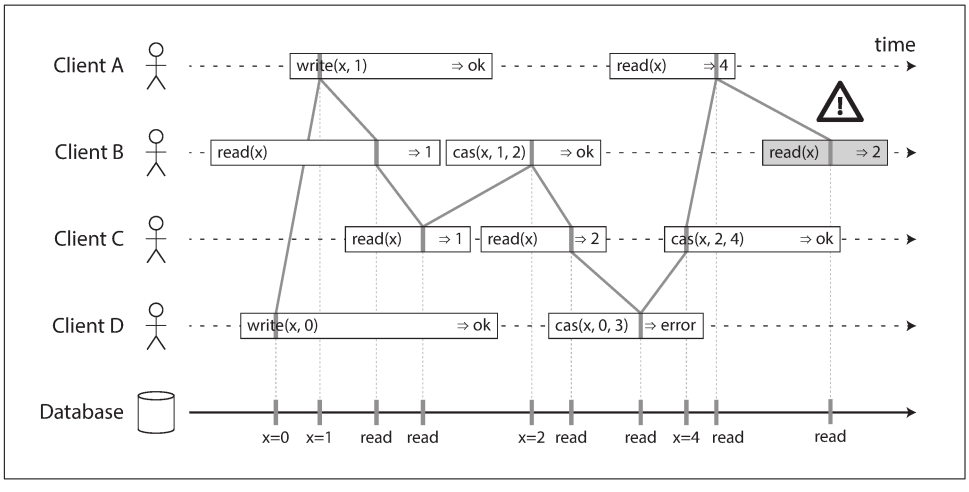



*Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.*

There are a few interesting details to point out in Figure 9-4:

- First client B sent a request to read *x*, then client D sent a request to set *x* to 0, and then client A sent a request to set *x* to 1. Nevertheless, the value returned to B's read is 1 (the value written by A). This is okay: it means that the database first processed D's write, then A's write, and finally B's read. Although this is not the order in which the requests were sent, it's an acceptable order, because the three requests are concurrent. Perhaps B's read request was slightly delayed in the network, so it only reached the database after the two writes.
- Client B's read returned 1 before client A received its response from the database, saying that the write of the value 1 was successful. This is also okay: it doesn't mean the value was read before it was written, it just means the *ok* response from the database to client A was slightly delayed in the network.
- This model doesn't assume any transaction isolation: another client may change a value at any time. For example, C first reads 1 and then reads 2, because the value was changed by B between the two reads. An atomic compare-and-set (*cas*) operation can be used to check the value hasn't been concurrently changed by another client: B and C's *cas* requests succeed, but D's *cas* request fails (by the time the database processes it, the value of *x* is no longer 0).
- The final read by client B (in a shaded bar) is not linearizable. The operation is concurrent with C's *cas* write, which updates *x* from 2 to 4. In the absence of

other requests, it would be okay for B's read to return 2. However, client A has already read the new value 4 before B's read started, so B is not allowed to read an older value than A. Again, it's the same situation as with Alice and Bob in Figure 9-1.

That is the intuition behind linearizability; the formal definition [6] describes it more precisely. It is possible (though computationally expensive) to test whether a system's behavior is linearizable by recording the timings of all requests and responses, and checking whether they can be arranged into a valid sequential order [11].

## Linearizability Versus Serializability

Linearizability is easily confused with serializability (see "Serializability" on page 251), as both words seem to mean something like "can be arranged in a sequential order." However, they are two quite different guarantees, and it is important to distinguish between them:

*Serializability*
: Serializability is an isolation property of *transactions*, where every transaction may read and write multiple objects (rows, documents, records)—see "Single-Object and Multi-Object Operations" on page 228. It guarantees that transactions behave the same as if they had executed in *some* serial order (each transaction running to completion before the next transaction starts). It is okay for that serial order to be different from the order in which transactions were actually run [12].

*Linearizability*
: Linearizability is a recency guarantee on reads and writes of a register (an *individual object*). It doesn't group operations together into transactions, so it does not prevent problems such as write skew (see "Write Skew and Phantoms" on page 246), unless you take additional measures such as materializing conflicts (see "Materializing conflicts" on page 251).

A database may provide both serializability and linearizability, and this combination is known as *strict serializability* or *strong one-copy serializability* (*strong-1SR*) [4, 13]. Implementations of serializability based on two-phase locking (see "Two-Phase Locking (2PL)" on page 257) or actual serial execution (see "Actual Serial Execution" on page 252) are typically linearizable.

However, serializable snapshot isolation (see "Serializable Snapshot Isolation (SSI)" on page 261) is not linearizable: by design, it makes reads from a consistent snapshot, to avoid lock contention between readers and writers. The whole point of a consistent snapshot is that it does not include writes that are more recent than the snapshot, and thus reads from the snapshot are not linearizable.

## Relying on Linearizability

In what circumstances is linearizability useful? Viewing the final score of a sporting match is perhaps a frivolous example: a result that is outdated by a few seconds is unlikely to cause any real harm in this situation. However, there a few areas in which linearizability is an important requirement for making a system work correctly.

### Locking and leader election

A system that uses single-leader replication needs to ensure that there is indeed only one leader, not several (split brain). One way of electing a leader is to use a lock: every node that starts up tries to acquire the lock, and the one that succeeds becomes the leader [14]. No matter how this lock is implemented, it must be linearizable: all nodes must agree which node owns the lock; otherwise it is useless.

Coordination services like Apache ZooKeeper [15] and etcd [16] are often used to implement distributed locks and leader election. They use consensus algorithms to implement linearizable operations in a fault-tolerant way (we discuss such algorithms later in this chapter, in "Fault-Tolerant Consensus" on page 364).[iii] There are still many subtle details to implementing locks and leader election correctly (see for example the fencing issue in "The leader and the lock" on page 301), and libraries like Apache Curator [17] help by providing higher-level recipes on top of ZooKeeper. However, a linearizable storage service is the basic foundation for these coordination tasks.

Distributed locking is also used at a much more granular level in some distributed databases, such as Oracle Real Application Clusters (RAC) [18]. RAC uses a lock per disk page, with multiple nodes sharing access to the same disk storage system. Since these linearizable locks are on the critical path of transaction execution, RAC deployments usually have a dedicated cluster interconnect network for communication between database nodes.

### Constraints and uniqueness guarantees

Uniqueness constraints are common in databases: for example, a username or email address must uniquely identify one user, and in a file storage service there cannot be two files with the same path and filename. If you want to enforce this constraint as the data is written (such that if two people try to concurrently create a user or a file with the same name, one of them will be returned an error), you need linearizability.

iii. Strictly speaking, ZooKeeper and etcd provide linearizable writes, but reads may be stale, since by default they can be served by any one of the replicas. You can optionally request a linearizable read: etcd calls this a *quorum read* [16], and in ZooKeeper you need to call `sync()` before the read [15]; see "Implementing linearizable storage using total order broadcast" on page 350.

This situation is actually similar to a lock: when a user registers for your service, you can think of them acquiring a "lock" on their chosen username. The operation is also very similar to an atomic compare-and-set, setting the username to the ID of the user who claimed it, provided that the username is not already taken.

Similar issues arise if you want to ensure that a bank account balance never goes negative, or that you don't sell more items than you have in stock in the warehouse, or that two people don't concurrently book the same seat on a flight or in a theater. These constraints all require there to be a single up-to-date value (the account balance, the stock level, the seat occupancy) that all nodes agree on.

In real applications, it is sometimes acceptable to treat such constraints loosely (for example, if a flight is overbooked, you can move customers to a different flight and offer them compensation for the inconvenience). In such cases, linearizability may not be needed, and we will discuss such loosely interpreted constraints in "Timeliness and Integrity" on page 524.

However, a hard uniqueness constraint, such as the one you typically find in relational databases, requires linearizability. Other kinds of constraints, such as foreign key or attribute constraints, can be implemented without requiring linearizability [19].

### Cross-channel timing dependencies

Notice a detail in Figure 9-1: if Alice hadn't exclaimed the score, Bob wouldn't have known that the result of his query was stale. He would have just refreshed the page again a few seconds later, and eventually seen the final score. The linearizability violation was only noticed because there was an additional communication channel in the system (Alice's voice to Bob's ears).

Similar situations can arise in computer systems. For example, say you have a website where users can upload a photo, and a background process resizes the photos to lower resolution for faster download (thumbnails). The architecture and dataflow of this system is illustrated in Figure 9-5.

The image resizer needs to be explicitly instructed to perform a resizing job, and this instruction is sent from the web server to the resizer via a message queue (see Chapter 11). The web server doesn't place the entire photo on the queue, since most message brokers are designed for small messages, and a photo may be several megabytes in size. Instead, the photo is first written to a file storage service, and once the write is complete, the instruction to the resizer is placed on the queue.

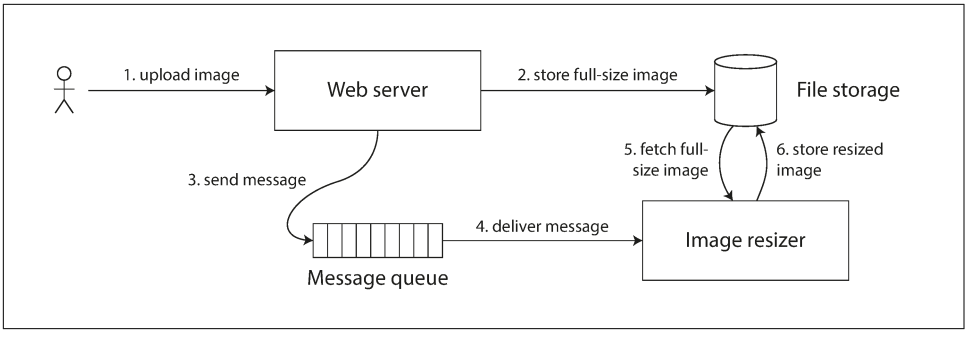



*Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.*

If the file storage service is linearizable, then this system should work fine. If it is not linearizable, there is the risk of a race condition: the message queue (steps 3 and 4 in Figure 9-5) might be faster than the internal replication inside the storage service. In this case, when the resizer fetches the image (step 5), it might see an old version of the image, or nothing at all. If it processes an old version of the image, the full-size and resized images in the file storage become permanently inconsistent.

This problem arises because there are two different communication channels between the web server and the resizer: the file storage and the message queue. Without the recency guarantee of linearizability, race conditions between these two channels are possible. This situation is analogous to Figure 9-1, where there was also a race condition between two communication channels: the database replication and the real-life audio channel between Alice's mouth and Bob's ears.

Linearizability is not the only way of avoiding this race condition, but it's the simplest to understand. If you control the additional communication channel (like in the case of the message queue, but not in the case of Alice and Bob), you can use alternative approaches similar to what we discussed in "Reading Your Own Writes" on page 162, at the cost of additional complexity.

## Implementing Linearizable Systems

Now that we've looked at a few examples in which linearizability is useful, let's think about how we might implement a system that offers linearizable semantics.

Since linearizability essentially means "behave as though there is only a single copy of the data, and all operations on it are atomic," the simplest answer would be to really only use a single copy of the data. However, that approach would not be able to tolerate faults: if the node holding that one copy failed, the data would be lost, or at least inaccessible until the node was brought up again.

The most common approach to making a system fault-tolerant is to use replication. Let's revisit the replication methods from Chapter 5, and compare whether they can be made linearizable:

*Single-leader replication (potentially linearizable)*

In a system with single-leader replication (see "Leaders and Followers" on page 152), the leader has the primary copy of the data that is used for writes, and the followers maintain backup copies of the data on other nodes. If you make reads from the leader, or from synchronously updated followers, they have the *potential* to be linearizable.[iv] However, not every single-leader database is actually linearizable, either by design (e.g., because it uses snapshot isolation) or due to concurrency bugs [10].

Using the leader for reads relies on the assumption that you know for sure who the leader is. As discussed in "The Truth Is Defined by the Majority" on page 300, it is quite possible for a node to think that it is the leader, when in fact it is not—and if the delusional leader continues to serve requests, it is likely to violate linearizability [20]. With asynchronous replication, failover may even lose committed writes (see "Handling Node Outages" on page 156), which violates both durability and linearizability.

*Consensus algorithms (linearizable)*

Some consensus algorithms, which we will discuss later in this chapter, bear a resemblance to single-leader replication. However, consensus protocols contain measures to prevent split brain and stale replicas. Thanks to these details, consensus algorithms can implement linearizable storage safely. This is how ZooKeeper [21] and etcd [22] work, for example.

*Multi-leader replication (not linearizable)*

Systems with multi-leader replication are generally not linearizable, because they concurrently process writes on multiple nodes and asynchronously replicate them to other nodes. For this reason, they can produce conflicting writes that require resolution (see "Handling Write Conflicts" on page 171). Such conflicts are an artifact of the lack of a single copy of the data.

*Leaderless replication (probably not linearizable)*

For systems with leaderless replication (Dynamo-style; see "Leaderless Replication" on page 177), people sometimes claim that you can obtain "strong consistency" by requiring quorum reads and writes ($w + r > n$). Depending on the exact

iv. Partitioning (sharding) a single-leader database, so that there is a separate leader per partition, does not affect linearizability, since it is only a single-object guarantee. Cross-partition transactions are a different matter (see "Distributed Transactions and Consensus" on page 352).

configuration of the quorums, and depending on how you define strong consistency, this is not quite true.

"Last write wins" conflict resolution methods based on time-of-day clocks (e.g., in Cassandra; see "Relying on Synchronized Clocks" on page 291) are almost certainly nonlinearizable, because clock timestamps cannot be guaranteed to be consistent with actual event ordering due to clock skew. Sloppy quorums ("Sloppy Quorums and Hinted Handoff" on page 183) also ruin any chance of linearizability. Even with strict quorums, nonlinearizable behavior is possible, as demonstrated in the next section.

### Linearizability and quorums

Intuitively, it seems as though strict quorum reads and writes should be linearizable in a Dynamo-style model. However, when we have variable network delays, it is possible to have race conditions, as demonstrated in Figure 9-6.

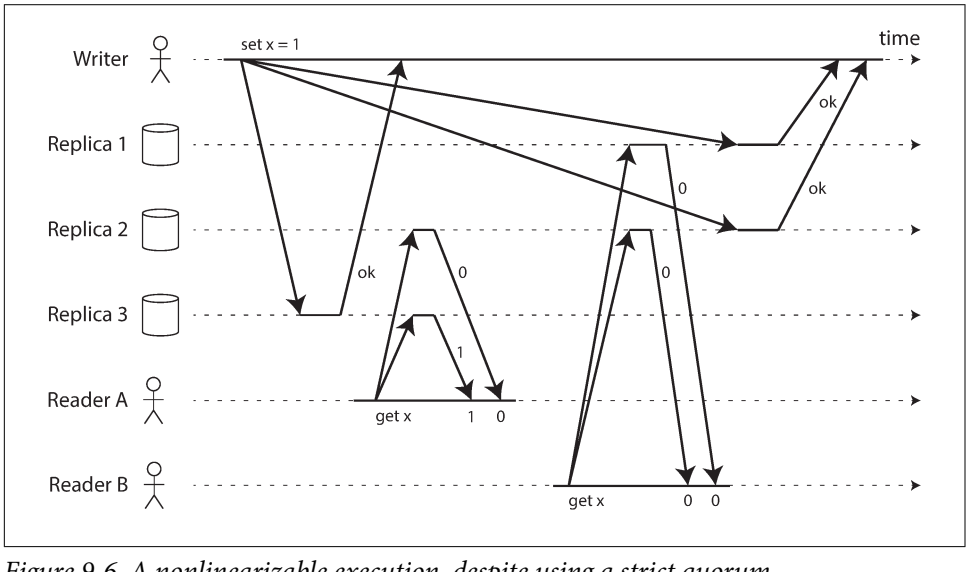



*Figure 9-6. A nonlinearizable execution, despite using a strict quorum.*

In Figure 9-6, the initial value of $x$ is 0, and a writer client is updating $x$ to 1 by sending the write to all three replicas ($n = 3$, $w = 3$). Concurrently, client A reads from a quorum of two nodes ($r = 2$) and sees the new value 1 on one of the nodes. Also concurrently with the write, client B reads from a different quorum of two nodes, and gets back the old value 0 from both.

The quorum condition is met ($w + r > n$), but this execution is nevertheless not linearizable: B's request begins after A's request completes, but B returns the old value

while A returns the new value. (It's once again the Alice and Bob situation from Figure 9-1.)

Interestingly, it *is* possible to make Dynamo-style quorums linearizable at the cost of reduced performance: a reader must perform read repair (see "Read repair and anti-entropy" on page 178) synchronously, before returning results to the application [23], and a writer must read the latest state of a quorum of nodes before sending its writes [24, 25]. However, Riak does not perform synchronous read repair due to the performance penalty [26]. Cassandra *does* wait for read repair to complete on quorum reads [27], but it loses linearizability if there are multiple concurrent writes to the same key, due to its use of last-write-wins conflict resolution.

Moreover, only linearizable read and write operations can be implemented in this way; a linearizable compare-and-set operation cannot, because it requires a consensus algorithm [28].

In summary, it is safest to assume that a leaderless system with Dynamo-style replication does not provide linearizability.

## The Cost of Linearizability

As some replication methods can provide linearizability and others cannot, it is interesting to explore the pros and cons of linearizability in more depth.

We already discussed some use cases for different replication methods in Chapter 5; for example, we saw that multi-leader replication is often a good choice for multi-datacenter replication (see "Multi-datacenter operation" on page 168). An example of such a deployment is illustrated in Figure 9-7.

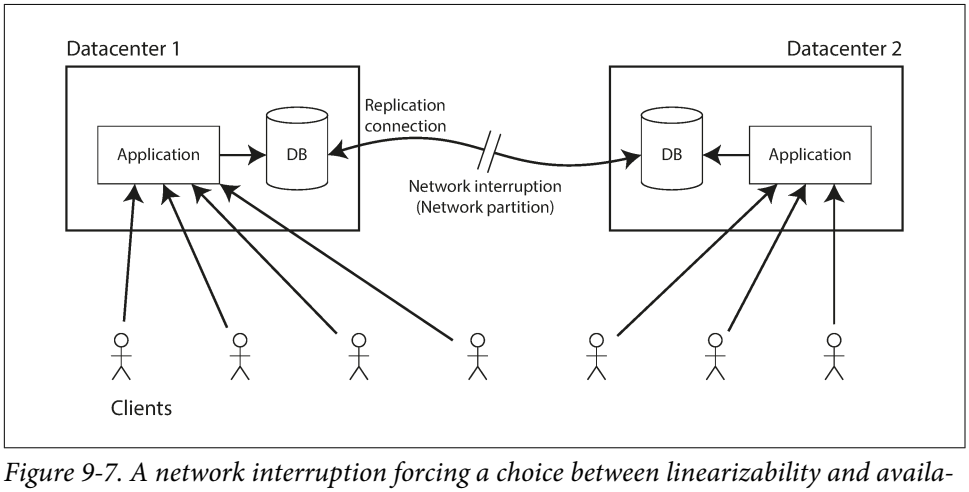



*Figure 9-7. A network interruption forcing a choice between linearizability and availability.*

Consider what happens if there is a network interruption between the two datacenters. Let's assume that the network within each datacenter is working, and clients can reach the datacenters, but the datacenters cannot connect to each other.

With a multi-leader database, each datacenter can continue operating normally: since writes from one datacenter are asynchronously replicated to the other, the writes are simply queued up and exchanged when network connectivity is restored.

On the other hand, if single-leader replication is used, then the leader must be in one of the datacenters. Any writes and any linearizable reads must be sent to the leader—thus, for any clients connected to a follower datacenter, those read and write requests must be sent synchronously over the network to the leader datacenter.

If the network between datacenters is interrupted in a single-leader setup, clients connected to follower datacenters cannot contact the leader, so they cannot make any writes to the database, nor any linearizable reads. They can still make reads from the follower, but they might be stale (nonlinearizable). If the application requires linearizable reads and writes, the network interruption causes the application to become unavailable in the datacenters that cannot contact the leader.

If clients can connect directly to the leader datacenter, this is not a problem, since the application continues to work normally there. But clients that can only reach a follower datacenter will experience an outage until the network link is repaired.

### The CAP theorem

This issue is not just a consequence of single-leader and multi-leader replication: any linearizable database has this problem, no matter how it is implemented. The issue also isn't specific to multi-datacenter deployments, but can occur on any unreliable network, even within one datacenter. The trade-off is as follows:[v]

- If your application *requires* linearizability, and some replicas are disconnected from the other replicas due to a network problem, then some replicas cannot process requests while they are disconnected: they must either wait until the network problem is fixed, or return an error (either way, they become *unavailable*).
- If your application *does not require* linearizability, then it can be written in a way that each replica can process requests independently, even if it is disconnected from other replicas (e.g., multi-leader). In this case, the application can remain *available* in the face of a network problem, but its behavior is not linearizable.

v. These two choices are sometimes known as CP (consistent but not available under network partitions) and AP (available but not consistent under network partitions), respectively. However, this classification scheme has several flaws [9], so it is best avoided.

Thus, applications that don't require linearizability can be more tolerant of network problems. This insight is popularly known as the *CAP theorem* [29, 30, 31, 32], named by Eric Brewer in 2000, although the trade-off has been known to designers of distributed databases since the 1970s [33, 34, 35, 36].

CAP was originally proposed as a rule of thumb, without precise definitions, with the goal of starting a discussion about trade-offs in databases. At the time, many distributed databases focused on providing linearizable semantics on a cluster of machines with shared storage [18], and CAP encouraged database engineers to explore a wider design space of distributed shared-nothing systems, which were more suitable for implementing large-scale web services [37]. CAP deserves credit for this culture shift—witness the explosion of new database technologies since the mid-2000s (known as NoSQL).

### The Unhelpful CAP Theorem

CAP is sometimes presented as *Consistency, Availability, Partition tolerance: pick 2 out of 3*. Unfortunately, putting it this way is misleading [32] because network partitions are a kind of fault, so they aren't something about which you have a choice: they will happen whether you like it or not [38].

At times when the network is working correctly, a system can provide both consistency (linearizability) and total availability. When a network fault occurs, you have to choose between either linearizability or total availability. Thus, a better way of phrasing CAP would be *either Consistent or Available when Partitioned* [39]. A more reliable network needs to make this choice less often, but at some point the choice is inevitable.

In discussions of CAP there are several contradictory definitions of the term *availability*, and the formalization as a theorem [30] does not match its usual meaning [40]. Many so-called "highly available" (fault-tolerant) systems actually do not meet CAP's idiosyncratic definition of availability. All in all, there is a lot of misunderstanding and confusion around CAP, and it does not help us understand systems better, so CAP is best avoided.

The CAP theorem as formally defined [30] is of very narrow scope: it only considers one consistency model (namely linearizability) and one kind of fault (*network partitions*,[vi] or nodes that are alive but disconnected from each other). It doesn't say any-

vi. As discussed in "Network Faults in Practice" on page 279, this book uses *partitioning* to refer to deliberately breaking down a large dataset into smaller ones (*sharding*; see Chapter 6). By contrast, a network partition is a particular type of network fault, which we normally don't consider separately from other kinds of faults. However, since it's the P in CAP, we can't avoid the confusion in this case.

thing about network delays, dead nodes, or other trade-offs. Thus, although CAP has been historically influential, it has little practical value for designing systems [9, 40].

There are many more interesting impossibility results in distributed systems [41], and CAP has now been superseded by more precise results [2, 42], so it is of mostly historical interest today.

### Linearizability and network delays

Although linearizability is a useful guarantee, surprisingly few systems are actually linearizable in practice. For example, even RAM on a modern multi-core CPU is not linearizable [43]: if a thread running on one CPU core writes to a memory address, and a thread on another CPU core reads the same address shortly afterward, it is not guaranteed to read the value written by the first thread (unless a *memory barrier* or *fence* [44] is used).

The reason for this behavior is that every CPU core has its own memory cache and store buffer. Memory access first goes to the cache by default, and any changes are asynchronously written out to main memory. Since accessing data in the cache is much faster than going to main memory [45], this feature is essential for good performance on modern CPUs. However, there are now several copies of the data (one in main memory, and perhaps several more in various caches), and these copies are asynchronously updated, so linearizability is lost.

Why make this trade-off? It makes no sense to use the CAP theorem to justify the multi-core memory consistency model: within one computer we usually assume reliable communication, and we don't expect one CPU core to be able to continue operating normally if it is disconnected from the rest of the computer. The reason for dropping linearizability is *performance*, not fault tolerance.

The same is true of many distributed databases that choose not to provide linearizable guarantees: they do so primarily to increase performance, not so much for fault tolerance [46]. Linearizability is slow—and this is true all the time, not only during a network fault.

Can't we maybe find a more efficient implementation of linearizable storage? It seems the answer is no: Attiya and Welch [47] prove that if you want linearizability, the response time of read and write requests is at least proportional to the uncertainty of delays in the network. In a network with highly variable delays, like most computer networks (see "Timeouts and Unbounded Delays" on page 281), the response time of linearizable reads and writes is inevitably going to be high. A faster algorithm for linearizability does not exist, but weaker consistency models can be much faster, so this trade-off is important for latency-sensitive systems. In Chapter 12 we will discuss some approaches for avoiding linearizability without sacrificing correctness.

# Ordering Guarantees

We said previously that a linearizable register behaves as if there is only a single copy of the data, and that every operation appears to take effect atomically at one point in time. This definition implies that operations are executed in some well-defined order. We illustrated the ordering in Figure 9-4 by joining up the operations in the order in which they seem to have executed.

Ordering has been a recurring theme in this book, which suggests that it might be an important fundamental idea. Let's briefly recap some of the other contexts in which we have discussed ordering:

- In Chapter 5 we saw that the main purpose of the leader in single-leader replication is to determine the *order of writes* in the replication log—that is, the order in which followers apply those writes. If there is no single leader, conflicts can occur due to concurrent operations (see "Handling Write Conflicts" on page 171).
- Serializability, which we discussed in Chapter 7, is about ensuring that transactions behave as if they were executed in *some sequential order*. It can be achieved by literally executing transactions in that serial order, or by allowing concurrent execution while preventing serialization conflicts (by locking or aborting).
- The use of timestamps and clocks in distributed systems that we discussed in Chapter 8 (see "Relying on Synchronized Clocks" on page 291) is another attempt to introduce order into a disorderly world, for example to determine which one of two writes happened later.

It turns out that there are deep connections between ordering, linearizability, and consensus. Although this notion is a bit more theoretical and abstract than the rest of this book, it is very helpful for clarifying our understanding of what systems can and cannot do. We will explore this topic in the next few sections.

## Ordering and Causality

There are several reasons why ordering keeps coming up, and one of the reasons is that it helps preserve *causality*. We have already seen several examples over the course of this book where causality has been important:

- In "Consistent Prefix Reads" on page 165 (Figure 5-5) we saw an example where the observer of a conversation saw first the answer to a question, and then the question being answered. This is confusing because it violates our intuition of cause and effect: if a question is answered, then clearly the question had to be there first, because the person giving the answer must have seen the question (assuming they are not psychic and cannot see into the future). We say that there is a *causal dependency* between the question and the answer.

- A similar pattern appeared in Figure 5-9, where we looked at the replication between three leaders and noticed that some writes could "overtake" others due to network delays. From the perspective of one of the replicas it would look as though there was an update to a row that did not exist. Causality here means that a row must first be created before it can be updated.
- In "Detecting Concurrent Writes" on page 184 we observed that if you have two operations A and B, there are three possibilities: either A happened before B, or B happened before A, or A and B are concurrent. This *happened before* relationship is another expression of causality: if A happened before B, that means B might have known about A, or built upon A, or depended on A. If A and B are concurrent, there is no causal link between them; in other words, we are sure that neither knew about the other.
- In the context of snapshot isolation for transactions ("Snapshot Isolation and Repeatable Read" on page 237), we said that a transaction reads from a consistent snapshot. But what does "consistent" mean in this context? It means *consistent with causality*: if the snapshot contains an answer, it must also contain the question being answered [48]. Observing the entire database at a single point in time makes it consistent with causality: the effects of all operations that happened causally before that point in time are visible, but no operations that happened causally afterward can be seen. Read skew (non-repeatable reads, as illustrated in Figure 7-6) means reading data in a state that violates causality.
- Our examples of write skew between transactions (see "Write Skew and Phantoms" on page 246) also demonstrated causal dependencies: in Figure 7-8, Alice was allowed to go off call because the transaction thought that Bob was still on call, and vice versa. In this case, the action of going off call is causally dependent on the observation of who is currently on call. Serializable snapshot isolation (see "Serializable Snapshot Isolation (SSI)" on page 261) detects write skew by tracking the causal dependencies between transactions.
- In the example of Alice and Bob watching football (Figure 9-1), the fact that Bob got a stale result from the server after hearing Alice exclaim the result is a causality violation: Alice's exclamation is causally dependent on the announcement of the score, so Bob should also be able to see the score after hearing Alice. The same pattern appeared again in "Cross-channel timing dependencies" on page 331 in the guise of an image resizing service.

Causality imposes an ordering on events: cause comes before effect; a message is sent before that message is received; the question comes before the answer. And, like in real life, one thing leads to another: one node reads some data and then writes something as a result, another node reads the thing that was written and writes something else in turn, and so on. These chains of causally dependent operations define the causal order in the system—i.e., what happened before what.

If a system obeys the ordering imposed by causality, we say that it is *causally consistent*. For example, snapshot isolation provides causal consistency: when you read from the database, and you see some piece of data, then you must also be able to see any data that causally precedes it (assuming it has not been deleted in the meantime).

### The causal order is not a total order

A *total order* allows any two elements to be compared, so if you have two elements, you can always say which one is greater and which one is smaller. For example, natural numbers are totally ordered: if I give you any two numbers, say 5 and 13, you can tell me that 13 is greater than 5.

However, mathematical sets are not totally ordered: is $\{a, b\}$ greater than $\{b, c\}$? Well, you can't really compare them, because neither is a subset of the other. We say they are *incomparable*, and therefore mathematical sets are *partially ordered*: in some cases one set is greater than another (if one set contains all the elements of another), but in other cases they are incomparable.

The difference between a total order and a partial order is reflected in different database consistency models:

*Linearizability*
: In a linearizable system, we have a *total order* of operations: if the system behaves as if there is only a single copy of the data, and every operation is atomic, this means that for any two operations we can always say which one happened first. This total ordering is illustrated as a timeline in Figure 9-4.

*Causality*
: We said that two operations are concurrent if neither happened before the other (see "The "happens-before" relationship and concurrency" on page 186). Put another way, two events are ordered if they are causally related (one happened before the other), but they are incomparable if they are concurrent. This means that causality defines a *partial order*, not a total order: some operations are ordered with respect to each other, but some are incomparable.

Therefore, according to this definition, there are no concurrent operations in a linearizable datastore: there must be a single timeline along which all operations are totally ordered. There might be several requests waiting to be handled, but the datastore ensures that every request is handled atomically at a single point in time, acting on a single copy of the data, along a single timeline, without any concurrency.

Concurrency would mean that the timeline branches and merges again—and in this case, operations on different branches are incomparable (i.e., concurrent). We saw this phenomenon in Chapter 5: for example, Figure 5-14 is not a straight-line total order, but rather a jumble of different operations going on concurrently. The arrows in the diagram indicate causal dependencies—the partial ordering of operations.

If you are familiar with distributed version control systems such as Git, their version histories are very much like the graph of causal dependencies. Often one commit happens after another, in a straight line, but sometimes you get branches (when several people concurrently work on a project), and merges are created when those concurrently created commits are combined.

### Linearizability is stronger than causal consistency

So what is the relationship between the causal order and linearizability? The answer is that linearizability *implies* causality: any system that is linearizable will preserve causality correctly [7]. In particular, if there are multiple communication channels in a system (such as the message queue and the file storage service in Figure 9-5), linearizability ensures that causality is automatically preserved without the system having to do anything special (such as passing around timestamps between different components).

The fact that linearizability ensures causality is what makes linearizable systems simple to understand and appealing. However, as discussed in "The Cost of Linearizability" on page 335, making a system linearizable can harm its performance and availability, especially if the system has significant network delays (for example, if it's geographically distributed). For this reason, some distributed data systems have abandoned linearizability, which allows them to achieve better performance but can make them difficult to work with.

The good news is that a middle ground is possible. Linearizability is not the only way of preserving causality—there are other ways too. A system can be causally consistent without incurring the performance hit of making it linearizable (in particular, the CAP theorem does not apply). In fact, causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remains available in the face of network failures [2, 42].

In many cases, systems that appear to require linearizability in fact only really require causal consistency, which can be implemented more efficiently. Based on this observation, researchers are exploring new kinds of databases that preserve causality, with performance and availability characteristics that are similar to those of eventually consistent systems [49, 50, 51].

As this research is quite recent, not much of it has yet made its way into production systems, and there are still challenges to be overcome [52, 53]. However, it is a promising direction for future systems.

### Capturing causal dependencies

We won't go into all the nitty-gritty details of how nonlinearizable systems can maintain causal consistency here, but just briefly explore some of the key ideas.

In order to maintain causality, you need to know which operation *happened before* which other operation. This is a partial order: concurrent operations may be processed in any order, but if one operation happened before another, then they must be processed in that order on every replica. Thus, when a replica processes an operation, it must ensure that all causally preceding operations (all operations that happened before) have already been processed; if some preceding operation is missing, the later operation must wait until the preceding operation has been processed.

In order to determine causal dependencies, we need some way of describing the "knowledge" of a node in the system. If a node had already seen the value X when it issued the write Y, then X and Y may be causally related. The analysis uses the kinds of questions you would expect in a criminal investigation of fraud charges: did the CEO *know* about X at the time when they made decision Y?

The techniques for determining which operation happened before which other operation are similar to what we discussed in "Detecting Concurrent Writes" on page 184. That section discussed causality in a leaderless datastore, where we need to detect concurrent writes to the same key in order to prevent lost updates. Causal consistency goes further: it needs to track causal dependencies across the entire database, not just for a single key. Version vectors can be generalized to do this [54].

In order to determine the causal ordering, the database needs to know which version of the data was read by the application. This is why, in Figure 5-13, the version number from the prior operation is passed back to the database on a write. A similar idea appears in the conflict detection of SSI, as discussed in "Serializable Snapshot Isolation (SSI)" on page 261: when a transaction wants to commit, the database checks whether the version of the data that it read is still up to date. To this end, the database keeps track of which data has been read by which transaction.

## Sequence Number Ordering

Although causality is an important theoretical concept, actually keeping track of all causal dependencies can become impracticable. In many applications, clients read lots of data before writing something, and then it is not clear whether the write is causally dependent on all or only some of those prior reads. Explicitly tracking all the data that has been read would mean a large overhead.

However, there is a better way: we can use *sequence numbers* or *timestamps* to order events. A timestamp need not come from a time-of-day clock (or physical clock, which have many problems, as discussed in "Unreliable Clocks" on page 287). It can instead come from a *logical clock*, which is an algorithm to generate a sequence of numbers to identify operations, typically using counters that are incremented for every operation.

Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a *total order*: that is, every operation has a unique sequence number, and you can always compare two sequence numbers to determine which is greater (i.e., which operation happened later).

In particular, we can create sequence numbers in a total order that is *consistent with causality*:[vii] we promise that if operation A causally happened before B, then A occurs before B in the total order (A has a lower sequence number than B). Concurrent operations may be ordered arbitrarily. Such a total order captures all the causality information, but also imposes more ordering than strictly required by causality.

In a database with single-leader replication (see "Leaders and Followers" on page 152), the replication log defines a total order of write operations that is consistent with causality. The leader can simply increment a counter for each operation, and thus assign a monotonically increasing sequence number to each operation in the replication log. If a follower applies the writes in the order they appear in the replication log, the state of the follower is always causally consistent (even if it is lagging behind the leader).

### Noncausal sequence number generators

If there is not a single leader (perhaps because you are using a multi-leader or leaderless database, or because the database is partitioned), it is less clear how to generate sequence numbers for operations. Various methods are used in practice:

- Each node can generate its own independent set of sequence numbers. For example, if you have two nodes, one node can generate only odd numbers and the other only even numbers. In general, you could reserve some bits in the binary representation of the sequence number to contain a unique node identifier, and this would ensure that two different nodes can never generate the same sequence number.
- You can attach a timestamp from a time-of-day clock (physical clock) to each operation [55]. Such timestamps are not sequential, but if they have sufficiently high resolution, they might be sufficient to totally order operations. This fact is used in the last write wins conflict resolution method (see "Timestamps for ordering events" on page 291).
- You can preallocate blocks of sequence numbers. For example, node A might claim the block of sequence numbers from 1 to 1,000, and node B might claim

vii. A total order that is *inconsistent* with causality is easy to create, but not very useful. For example, you can generate a random UUID for each operation, and compare UUIDs lexicographically to define the total ordering of operations. This is a valid total order, but the random UUIDs tell you nothing about which operation actually happened first, or whether the operations were concurrent.

the block from 1,001 to 2,000. Then each node can independently assign sequence numbers from its block, and allocate a new block when its supply of sequence numbers begins to run low.

These three options all perform better and are more scalable than pushing all operations through a single leader that increments a counter. They generate a unique, approximately increasing sequence number for each operation. However, they all have a problem: the sequence numbers they generate are *not consistent with causality*.

The causality problems occur because these sequence number generators do not correctly capture the ordering of operations across different nodes:

- Each node may process a different number of operations per second. Thus, if one node generates even numbers and the other generates odd numbers, the counter for even numbers may lag behind the counter for odd numbers, or vice versa. If you have an odd-numbered operation and an even-numbered operation, you cannot accurately tell which one causally happened first.
- Timestamps from physical clocks are subject to clock skew, which can make them inconsistent with causality. For example, see Figure 8-3, which shows a scenario in which an operation that happened causally later was actually assigned a lower timestamp.[viii]
- In the case of the block allocator, one operation may be given a sequence number in the range from 1,001 to 2,000, and a causally later operation may be given a number in the range from 1 to 1,000. Here, again, the sequence number is inconsistent with causality.

### Lamport timestamps

Although the three sequence number generators just described are inconsistent with causality, there is actually a simple method for generating sequence numbers that *is* consistent with causality. It is called a *Lamport timestamp*, proposed in 1978 by Leslie Lamport [56], in what is now one of the most-cited papers in the field of distributed systems.

The use of Lamport timestamps is illustrated in Figure 9-8. Each node has a unique identifier, and each node keeps a counter of the number of operations it has processed. The Lamport timestamp is then simply a pair of (*counter*, *node ID*). Two

viii. It is possible to make physical clock timestamps consistent with causality: in "Synchronized clocks for global snapshots" on page 294 we discussed Google's Spanner, which estimates the expected clock skew and waits out the uncertainty interval before committing a write. This method ensures that a causally later transaction is given a greater timestamp. However, most clocks cannot provide the required uncertainty metric.

nodes may sometimes have the same counter value, but by including the node ID in the timestamp, each timestamp is made unique.

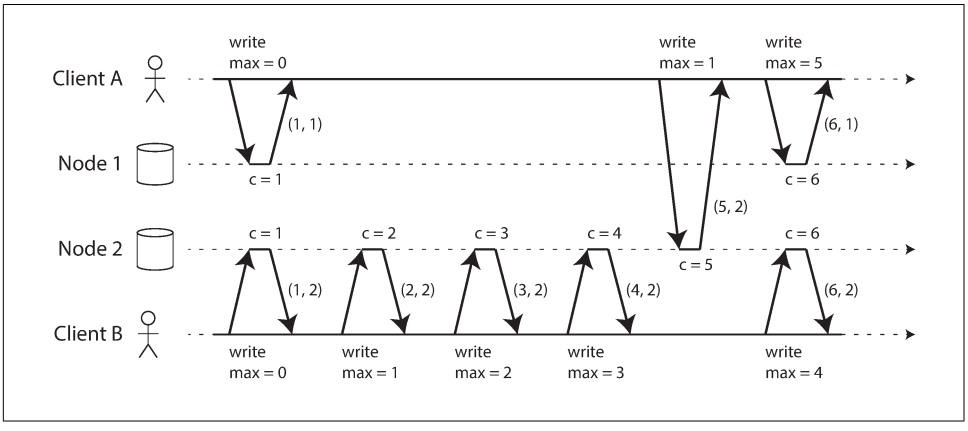



*Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.*

A Lamport timestamp bears no relationship to a physical time-of-day clock, but it provides total ordering: if you have two timestamps, the one with a greater counter value is the greater timestamp; if the counter values are the same, the one with the greater node ID is the greater timestamp.

So far this description is essentially the same as the even/odd counters described in the last section. The key idea about Lamport timestamps, which makes them consistent with causality, is the following: every node and every client keeps track of the *maximum* counter value it has seen so far, and includes that maximum on every request. When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum.

This is shown in Figure 9-8, where client A receives a counter value of 5 from node 2, and then sends that maximum of 5 to node 1. At that time, node 1's counter was only 1, but it was immediately moved forward to 5, so the next operation had an incremented counter value of 6.

As long as the maximum counter value is carried along with every operation, this scheme ensures that the ordering from the Lamport timestamps is consistent with causality, because every causal dependency results in an increased timestamp.

Lamport timestamps are sometimes confused with version vectors, which we saw in "Detecting Concurrent Writes" on page 184. Although there are some similarities, they have a different purpose: version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other, whereas Lamport timestamps always enforce a total ordering. From the total ordering of Lamport time-

stamps, you cannot tell whether two operations are concurrent or whether they are causally dependent. The advantage of Lamport timestamps over version vectors is that they are more compact.

### Timestamp ordering is not sufficient

Although Lamport timestamps define a total order of operations that is consistent with causality, they are not quite sufficient to solve many common problems in distributed systems.

For example, consider a system that needs to ensure that a username uniquely identifies a user account. If two users concurrently try to create an account with the same username, one of the two should succeed and the other should fail. (We touched on this problem previously in "The leader and the lock" on page 301.)

At first glance, it seems as though a total ordering of operations (e.g., using Lamport timestamps) should be sufficient to solve this problem: if two accounts with the same username are created, pick the one with the lower timestamp as the winner (the one who grabbed the username first), and let the one with the greater timestamp fail. Since timestamps are totally ordered, this comparison is always valid.

This approach works for determining the winner after the fact: once you have collected all the username creation operations in the system, you can compare their timestamps. However, this approach is not sufficient when a node has just received a request from a user to create a username, and needs to decide *right now* whether the request should succeed or fail. At that moment, the node does not know whether another node is concurrently in the process of creating an account with the same username, and what timestamp that other node may assign to the operation.

In order to be sure that no other node is in the process of concurrently creating an account with the same username and a lower timestamp, you would have to check with *every* other node to find out which timestamps it has generated [56]. If one of the other nodes has failed or cannot be reached due to a network problem, this system would grind to a halt. This is not the kind of fault-tolerant system that we need.

The problem here is that the total order of operations only emerges after you have collected all of the operations. If another node has generated some operations, but you don't yet know what they are, you cannot construct the final ordering of operations: the unknown operations from the other node may need to be inserted at various positions in the total order.

To conclude: in order to implement something like a uniqueness constraint for usernames, it's not sufficient to have a total ordering of operations—you also need to know when that order is finalized. If you have an operation to create a username, and you are sure that no other node can insert a claim for the same username ahead of your operation in the total order, then you can safely declare the operation successful.

This idea of knowing when your total order is finalized is captured in the topic of *total order broadcast*.

## Total Order Broadcast

If your program runs only on a single CPU core, it is easy to define a total ordering of operations: it is simply the order in which they were executed by the CPU. However, in a distributed system, getting all nodes to agree on the same total ordering of operations is tricky. In the last section we discussed ordering by timestamps or sequence numbers, but found that it is not as powerful as single-leader replication (if you use timestamp ordering to implement a uniqueness constraint, you cannot tolerate any faults, because you have to fetch the latest sequence number from every node).

As discussed, single-leader replication determines a total order of operations by choosing one node as the leader and sequencing all operations on a single CPU core on the leader. The challenge then is how to scale the system if the throughput is greater than a single leader can handle, and also how to handle failover if the leader fails (see "Handling Node Outages" on page 156). In the distributed systems literature, this problem is known as *total order broadcast* or *atomic broadcast* [25, 57, 58].[ix]

### Scope of ordering guarantee

Partitioned databases with a single leader per partition often maintain ordering only per partition, which means they cannot offer consistency guarantees (e.g., consistent snapshots, foreign key references) across partitions. Total ordering across all partitions is possible, but requires additional coordination [59].

Total order broadcast is usually described as a protocol for exchanging messages between nodes. Informally, it requires that two safety properties always be satisfied:

*Reliable delivery*
: No messages are lost: if a message is delivered to one node, it is delivered to all nodes.

*Totally ordered delivery*
: Messages are delivered to every node in the same order.

A correct algorithm for total order broadcast must ensure that the reliability and ordering properties are always satisfied, even if a node or the network is faulty. Of

ix. The term *atomic broadcast* is traditional, but it is very confusing as it's inconsistent with other uses of the word *atomic*: it has nothing to do with atomicity in ACID transactions and is only indirectly related to atomic operations (in the sense of multi-threaded programming) or atomic registers (linearizable storage). The term *total order multicast* is another synonym.

course, messages will not be delivered while the network is interrupted, but an algorithm can keep retrying so that the messages get through when the network is eventually repaired (and then they must still be delivered in the correct order).

### Using total order broadcast

Consensus services such as ZooKeeper and etcd actually implement total order broadcast. This fact is a hint that there is a strong connection between total order broadcast and consensus, which we will explore later in this chapter.

Total order broadcast is exactly what you need for database replication: if every message represents a write to the database, and every replica processes the same writes in the same order, then the replicas will remain consistent with each other (aside from any temporary replication lag). This principle is known as *state machine replication* [60], and we will return to it in Chapter 11.

Similarly, total order broadcast can be used to implement serializable transactions: as discussed in "Actual Serial Execution" on page 252, if every message represents a deterministic transaction to be executed as a stored procedure, and if every node processes those messages in the same order, then the partitions and replicas of the database are kept consistent with each other [61].

An important aspect of total order broadcast is that the order is fixed at the time the messages are delivered: a node is not allowed to retroactively insert a message into an earlier position in the order if subsequent messages have already been delivered. This fact makes total order broadcast stronger than timestamp ordering.

Another way of looking at total order broadcast is that it is a way of creating a *log* (as in a replication log, transaction log, or write-ahead log): delivering a message is like appending to the log. Since all nodes must deliver the same messages in the same order, all nodes can read the log and see the same sequence of messages.

Total order broadcast is also useful for implementing a lock service that provides fencing tokens (see "Fencing tokens" on page 303). Every request to acquire the lock is appended as a message to the log, and all messages are sequentially numbered in the order they appear in the log. The sequence number can then serve as a fencing token, because it is monotonically increasing. In ZooKeeper, this sequence number is called `zxid` [15].

### Implementing linearizable storage using total order broadcast

As illustrated in Figure 9-4, in a linearizable system there is a total order of operations. Does that mean linearizability is the same as total order broadcast? Not quite, but there are close links between the two.[x]

Total order broadcast is asynchronous: messages are guaranteed to be delivered reliably in a fixed order, but there is no guarantee about *when* a message will be delivered (so one recipient may lag behind the others). By contrast, linearizability is a recency guarantee: a read is guaranteed to see the latest value written.

However, if you have total order broadcast, you can build linearizable storage on top of it. For example, you can ensure that usernames uniquely identify user accounts.

Imagine that for every possible username, you can have a linearizable register with an atomic compare-and-set operation. Every register initially has the value `null` (indicating that the username is not taken). When a user wants to create a username, you execute a compare-and-set operation on the register for that username, setting it to the user account ID, under the condition that the previous register value is `null`. If multiple users try to concurrently grab the same username, only one of the compare-and-set operations will succeed, because the others will see a value other than `null` (due to linearizability).

You can implement such a linearizable compare-and-set operation as follows by using total order broadcast as an append-only log [62, 63]:

1. Append a message to the log, tentatively indicating the username you want to claim.
2. Read the log, and wait for the message you appended to be delivered back to you.[xi]
3. Check for any messages claiming the username that you want. If the first message for your desired username is your own message, then you are successful: you can commit the username claim (perhaps by appending another message to the log) and acknowledge it to the client. If the first message for your desired username is from another user, you abort the operation.

---

x. In a formal sense, a linearizable read-write register is an "easier" problem. Total order broadcast is equivalent to consensus [67], which has no deterministic solution in the asynchronous crash-stop model [68], whereas a linearizable read-write register *can* be implemented in the same system model [23, 24, 25]. However, supporting atomic operations such as compare-and-set or increment-and-get in a register makes it equivalent to consensus [28]. Thus, the problems of consensus and a linearizable register are closely related.

xi. If you don't wait, but acknowledge the write immediately after it has been enqueued, you get something similar to the memory consistency model of multi-core x86 processors [43]. That model is neither linearizable nor sequentially consistent.

Because log entries are delivered to all nodes in the same order, if there are several concurrent writes, all nodes will agree on which one came first. Choosing the first of the conflicting writes as the winner and aborting later ones ensures that all nodes agree on whether a write was committed or aborted. A similar approach can be used to implement serializable multi-object transactions on top of a log [62].

While this procedure ensures linearizable writes, it doesn't guarantee linearizable reads—if you read from a store that is asynchronously updated from the log, it may be stale. (To be precise, the procedure described here provides *sequential consistency* [47, 64], sometimes also known as *timeline consistency* [65, 66], a slightly weaker guarantee than linearizability.) To make reads linearizable, there are a few options:

- You can sequence reads through the log by appending a message, reading the log, and performing the actual read when the message is delivered back to you. The message's position in the log thus defines the point in time at which the read happens. (Quorum reads in etcd work somewhat like this [16].)
- If the log allows you to fetch the position of the latest log message in a linearizable way, you can query that position, wait for all entries up to that position to be delivered to you, and then perform the read. (This is the idea behind ZooKeeper's `sync()` operation [15].)
- You can make your read from a replica that is synchronously updated on writes, and is thus sure to be up to date. (This technique is used in chain replication [63]; see also "Research on Replication" on page 155.)

### Implementing total order broadcast using linearizable storage

The last section showed how to build a linearizable compare-and-set operation from total order broadcast. We can also turn it around, assume that we have linearizable storage, and show how to build total order broadcast from it.

The easiest way is to assume you have a linearizable register that stores an integer and that has an atomic increment-and-get operation [28]. Alternatively, an atomic compare-and-set operation would also do the job.

The algorithm is simple: for every message you want to send through total order broadcast, you increment-and-get the linearizable integer, and then attach the value you got from the register as a sequence number to the message. You can then send the message to all nodes (resending any lost messages), and the recipients will deliver the messages consecutively by sequence number.

Note that unlike Lamport timestamps, the numbers you get from incrementing the linearizable register form a sequence with no gaps. Thus, if a node has delivered message 4 and receives an incoming message with a sequence number of 6, it knows that it must wait for message 5 before it can deliver message 6. The same is not the case

with Lamport timestamps—in fact, this is the key difference between total order broadcast and timestamp ordering.

How hard could it be to make a linearizable integer with an atomic increment-and-get operation? As usual, if things never failed, it would be easy: you could just keep it in a variable on one node. The problem lies in handling the situation when network connections to that node are interrupted, and restoring the value when that node fails [59]. In general, if you think hard enough about linearizable sequence number generators, you inevitably end up with a consensus algorithm.

This is no coincidence: it can be proved that a linearizable compare-and-set (or increment-and-get) register and total order broadcast are both *equivalent to consensus* [28, 67]. That is, if you can solve one of these problems, you can transform it into a solution for the others. This is quite a profound and surprising insight!

It is time to finally tackle the consensus problem head-on, which we will do in the rest of this chapter.

## Distributed Transactions and Consensus

Consensus is one of the most important and fundamental problems in distributed computing. On the surface, it seems simple: informally, the goal is simply to *get several nodes to agree on something*. You might think that this shouldn't be too hard. Unfortunately, many broken systems have been built in the mistaken belief that this problem is easy to solve.

Although consensus is very important, the section about it appears late in this book because the topic is quite subtle, and appreciating the subtleties requires some prerequisite knowledge. Even in the academic research community, the understanding of consensus only gradually crystallized over the course of decades, with many misunderstandings along the way. Now that we have discussed replication (Chapter 5), transactions (Chapter 7), system models (Chapter 8), linearizability, and total order broadcast (this chapter), we are finally ready to tackle the consensus problem.

There are a number of situations in which it is important for nodes to agree. For example:

*Leader election*

> In a database with single-leader replication, all nodes need to agree on which node is the leader. The leadership position might become contested if some nodes can't communicate with others due to a network fault. In this case, consensus is important to avoid a bad failover, resulting in a split brain situation in which two nodes both believe themselves to be the leader (see "Handling Node Outages" on page 156). If there were two leaders, they would both accept writes and their data would diverge, leading to inconsistency and data loss.

*Atomic commit*

In a database that supports transactions spanning several nodes or partitions, we have the problem that a transaction may fail on some nodes but succeed on others. If we want to maintain transaction atomicity (in the sense of ACID; see "Atomicity" on page 223), we have to get all nodes to agree on the outcome of the transaction: either they all abort/roll back (if anything goes wrong) or they all commit (if nothing goes wrong). This instance of consensus is known as the *atomic commit* problem.[xii]

## The Impossibility of Consensus

You may have heard about the FLP result [68]—named after the authors Fischer, Lynch, and Paterson—which proves that there is no algorithm that is always able to reach consensus if there is a risk that a node may crash. In a distributed system, we must assume that nodes may crash, so reliable consensus is impossible. Yet, here we are, discussing algorithms for achieving consensus. What is going on here?

The answer is that the FLP result is proved in the asynchronous system model (see "System Model and Reality" on page 306), a very restrictive model that assumes a deterministic algorithm that cannot use any clocks or timeouts. If the algorithm is allowed to use timeouts, or some other way of identifying suspected crashed nodes (even if the suspicion is sometimes wrong), then consensus becomes solvable [67]. Even just allowing the algorithm to use random numbers is sufficient to get around the impossibility result [69].

Thus, although the FLP result about the impossibility of consensus is of great theoretical importance, distributed systems can usually achieve consensus in practice.

In this section we will first examine the atomic commit problem in more detail. In particular, we will discuss the *two-phase commit* (2PC) algorithm, which is the most common way of implementing atomic commit and which is used in various databases, messaging systems, and application servers. It turns out that 2PC is a kind of consensus algorithm—but not a very good one [70, 71].

By learning from 2PC we will then work our way toward better consensus algorithms, such as those used in ZooKeeper (Zab) and etcd (Raft).

xii. Atomic commit is formalized slightly differently from consensus: an atomic transaction can commit only if *all* participants vote to commit, and must abort if any participant needs to abort. Consensus is allowed to decide on *any* value that is proposed by one of the participants. However, atomic commit and consensus are reducible to each other [70, 71]. *Nonblocking* atomic commit is harder than consensus—see "Three-phase commit" on page 359.

## Atomic Commit and Two-Phase Commit (2PC)

In Chapter 7 we learned that the purpose of transaction atomicity is to provide simple semantics in the case where something goes wrong in the middle of making several writes. The outcome of a transaction is either a successful *commit*, in which case all of the transaction's writes are made durable, or an *abort*, in which case all of the transaction's writes are rolled back (i.e., undone or discarded).

Atomicity prevents failed transactions from littering the database with half-finished results and half-updated state. This is especially important for multi-object transactions (see "Single-Object and Multi-Object Operations" on page 228) and databases that maintain secondary indexes. Each secondary index is a separate data structure from the primary data—thus, if you modify some data, the corresponding change needs to also be made in the secondary index. Atomicity ensures that the secondary index stays consistent with the primary data (if the index became inconsistent with the primary data, it would not be very useful).

### From single-node to distributed atomic commit

For transactions that execute at a single database node, 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" on page 82) 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 [72]. 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.

However, what if multiple nodes are involved in a transaction? For example, perhaps you have a multi-object transaction in a partitioned database, or a term-partitioned secondary index (in which the index entry may be on a different node from the primary data; see "Partitioning and Secondary Indexes" on page 206). Most "NoSQL" distributed datastores do not support such distributed transactions, but various clustered relational systems do (see "Distributed Transactions in Practice" on page 360).

In these cases, it is not sufficient to simply send a commit request to all of the nodes and independently commit the transaction on each one. In doing so, it could easily

happen that the commit succeeds on some nodes and fails on other nodes, which would violate the atomicity guarantee:

- 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.

If some nodes commit the transaction but others abort it, the nodes become inconsistent with each other (like in Figure 7-3). 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. For this reason, a node must only commit once it is certain that all other nodes in the transaction are also going to commit.

A transaction commit must be irrevocable—you are not allowed to change your mind and retroactively abort a transaction after it has been committed. The reason for this rule is that once data has been committed, it becomes visible to other transactions, and thus other clients may start relying on that data; this principle forms the basis of *read committed* isolation, discussed in "Read Committed" on page 234. If a transaction was allowed to abort after committing, any transactions that read the committed data would be based on data that was retroactively declared not to have existed—so they would have to be reverted as well.

(It is possible for the effects of a committed transaction to later be undone by another, *compensating transaction* [73, 74]. However, from the database's point of view this is a separate transaction, and thus any cross-transaction correctness requirements are the application's problem.)

### Introduction to two-phase commit

Two-phase commit is an algorithm for achieving atomic transaction commit across multiple nodes—i.e., to ensure that either all nodes commit or all nodes abort. It is a classic algorithm in distributed databases [13, 35, 75]. 2PC is used internally in some databases and also made available to applications in the form of *XA transactions* [76, 77] (which are supported by the Java Transaction API, for example) or via WS-AtomicTransaction for SOAP web services [78, 79].

The basic flow of 2PC is illustrated in Figure 9-9. 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).

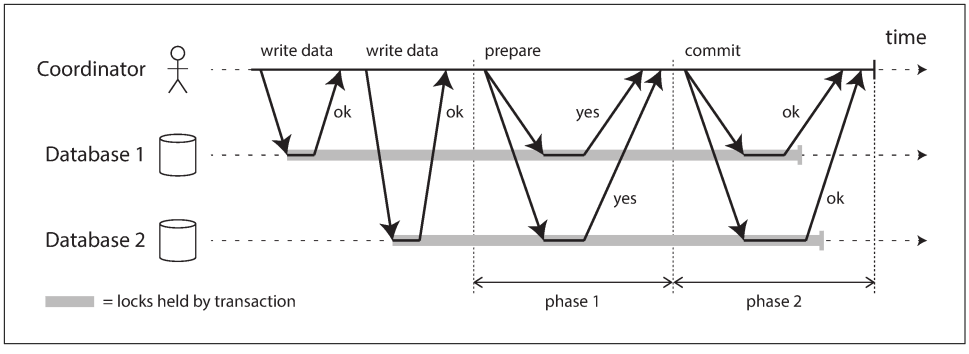



*Figure 9-9. A successful execution of two-phase commit (2PC).*

### Don't confuse 2PC and 2PL

Two-phase *commit* (2PC) and two-phase *locking* (see "Two-Phase Locking (2PL)" on page 257) are two very different things. 2PC provides atomic commit in a distributed database, whereas 2PL provides serializable isolation. To avoid confusion, it's best to think of them as entirely separate concepts and to ignore the unfortunate similarity in the names.

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.

A 2PC 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 [73].

### 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 com-

mitted 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 fail or time out, the coordinator aborts the transaction; if any of the commit or abort requests fail, 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 9-10. 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.

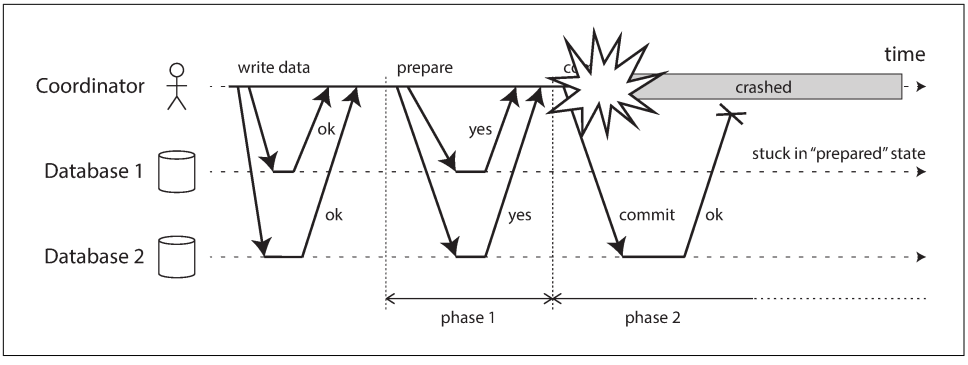



*Figure 9-10. The coordinator crashes after participants vote "yes." Database 1 does not know whether to commit or abort.*

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.

### 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. In theory, 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, 80]. 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 8), it cannot guarantee atomicity.

In general, nonblocking atomic commit requires a *perfect failure detector* [67, 71]—i.e., a reliable mechanism for telling whether a node has crashed or not. In a network with unbounded delay a timeout is not a reliable failure detector, because a request may time out due to a network problem even if no node has crashed. For this reason, 2PC continues to be used, despite the known problem with coordinator failure.

## Distributed Transactions in Practice

Distributed transactions, especially those implemented with 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 [81, 82, 83, 84]. Many cloud services choose not to implement distributed transactions due to the operational problems they engender [85, 86].

Some implementations of distributed transactions carry a heavy performance penalty —for example, distributed transactions in MySQL are reported to be over 10 times slower than single-node transactions [87], so it is not surprising when people advise against using them. Much of the performance cost inherent in two-phase commit is due to the additional disk forcing (`fsync`) that is required for crash recovery [88], 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 partitioning in their standard configuration) support internal transactions among the nodes of that database. For example, VoltDB 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.

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 message processing in Chapter 11. 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 [76, 77]. 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" on page 234, 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 (see "Two-Phase Locking (2PL)" on page 257).

The database cannot release those locks until the transaction commits or aborts (illustrated as a shaded area in Figure 9-9). 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 database, 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 [89, 90]—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 [76, 77, 91]. 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.

### Limitations of distributed transactions

XA transactions solve the real and important problem of keeping several participant data systems consistent with each other, but as we have seen, they also introduce major operational problems. In particular, the key realization is that the transaction coordinator is itself a kind of database (in which transaction outcomes are stored), and so it needs to be approached with the same care as any other important database:

- If the coordinator is not replicated but runs only on a single machine, it is a single point of failure for the entire system (since its failure causes other application servers to block on locks held by in-doubt transactions). Surprisingly, many coordinator implementations are not highly available by default, or have only rudimentary replication support.

- Many server-side applications are developed in a stateless model (as favored by HTTP), with all persistent state stored in a database, which has the advantage that application servers can be added and removed at will. However, when the coordinator is part of the application server, it changes the nature of the deployment. Suddenly, the coordinator's logs become a crucial part of the durable system state—as important as the databases themselves, since the coordinator logs are required in order to recover in-doubt transactions after a crash. Such application servers are no longer stateless.
- 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)" on page 261), since that would require a protocol for identifying conflicts across different systems.
- For database-internal distributed transactions (not XA), the limitations are not so great—for example, a distributed version of SSI is possible. However, there remains the problem that for 2PC to successfully commit a transaction, *all* participants must respond. Consequently, if *any* part of the system is broken, the transaction also fails. Distributed transactions thus have a tendency of *amplifying failures*, which runs counter to our goal of building fault-tolerant systems.

Do these facts mean we should give up all hope of keeping several systems consistent with each other? Not quite—there are alternative methods that allow us to achieve the same thing without the pain of heterogeneous distributed transactions. We will return to these in Chapters 11 and 12. But first, we should wrap up the topic of consensus.

## Fault-Tolerant Consensus

Informally, consensus means getting several nodes to agree on something. For example, if several people concurrently try to book the last seat on an airplane, or the same seat in a theater, or try to register an account with the same username, then a consensus algorithm could be used to determine which one of these mutually incompatible operations should be the winner.

The consensus problem is normally formalized as follows: one or more nodes may *propose* values, and the consensus algorithm *decides* on one of those values. In the seat-booking example, when several customers are concurrently trying to buy the last seat, each node handling a customer request may propose the ID of the customer it is serving, and the decision indicates which one of those customers got the seat.

In this formalism, a consensus algorithm must satisfy the following properties [25]:[xiii]

*Uniform agreement*
: No two nodes decide differently.

*Integrity*
: No node decides twice.

*Validity*
: If a node decides value *v*, then *v* was proposed by some node.

*Termination*
: Every node that does not crash eventually decides some value.

The uniform agreement and integrity properties define the core idea of consensus: everyone decides on the same outcome, and once you have decided, you cannot change your mind. The validity property exists mostly to rule out trivial solutions: for example, you could have an algorithm that always decides `null`, no matter what was proposed; this algorithm would satisfy the agreement and integrity properties, but not the validity property.

If you don't care about fault tolerance, then satisfying the first three properties is easy: you can just hardcode one node to be the "dictator," and let that node make all of the decisions. However, if that one node fails, then the system can no longer make any decisions. This is, in fact, what we saw in the case of two-phase commit: if the coordinator fails, in-doubt participants cannot decide whether to commit or abort.

The termination property formalizes the idea of fault tolerance. It essentially says that a consensus algorithm cannot simply sit around and do nothing forever—in other words, it must make progress. Even if some nodes fail, the other nodes must still reach a decision. (Termination is a liveness property, whereas the other three are safety properties—see "Safety and liveness" on page 308.)

The system model of consensus assumes that when a node "crashes," it suddenly disappears and never comes back. (Instead of a software crash, imagine that there is an earthquake, and the datacenter containing your node is destroyed by a landslide. You must assume that your node is buried under 30 feet of mud and is never going to come back online.) In this system model, any algorithm that has to wait for a node to recover is not going to be able to satisfy the termination property. In particular, 2PC does not meet the requirements for termination.



xiii. This particular variant of consensus is called *uniform consensus*, which is equivalent to regular consensus in asynchronous systems with unreliable failure detectors [71]. The academic literature usually refers to *processes* rather than *nodes*, but we use *nodes* here for consistency with the rest of this book.

Of course, if *all* nodes crash and none of them are running, then it is not possible for any algorithm to decide anything. There is a limit to the number of failures that an algorithm can tolerate: in fact, it can be proved that any consensus algorithm requires at least a majority of nodes to be functioning correctly in order to assure termination [67]. That majority can safely form a quorum (see "Quorums for reading and writing" on page 179).

Thus, the termination property is subject to the assumption that fewer than half of the nodes are crashed or unreachable. However, most implementations of consensus ensure that the safety properties—agreement, integrity, and validity—are always met, even if a majority of nodes fail or there is a severe network problem [92]. Thus, a large-scale outage can stop the system from being able to process requests, but it cannot corrupt the consensus system by causing it to make invalid decisions.

Most consensus algorithms assume that there are no Byzantine faults, as discussed in "Byzantine Faults" on page 304. That is, if a node does not correctly follow the protocol (for example, if it sends contradictory messages to different nodes), it may break the safety properties of the protocol. It is possible to make consensus robust against Byzantine faults as long as fewer than one-third of the nodes are Byzantine-faulty [25, 93], but we don't have space to discuss those algorithms in detail in this book.

### Consensus algorithms and total order broadcast

The best-known fault-tolerant consensus algorithms are Viewstamped Replication (VSR) [94, 95], Paxos [96, 97, 98, 99], Raft [22, 100, 101], and Zab [15, 21, 102]. There are quite a few similarities between these algorithms, but they are not the same [103]. In this book we won't go into full details of the different algorithms: it's sufficient to be aware of some of the high-level ideas that they have in common, unless you're implementing a consensus system yourself (which is probably not advisable—it's hard [98, 104]).

Most of these algorithms actually don't directly use the formal model described here (proposing and deciding on a single value, while satisfying the agreement, integrity, validity, and termination properties). Instead, they decide on a *sequence* of values, which makes them *total order broadcast* algorithms, as discussed previously in this chapter (see "Total Order Broadcast" on page 348).

Remember that total order broadcast requires messages to be delivered exactly once, in the same order, to all nodes. If you think about it, this is equivalent to performing several rounds of consensus: in each round, nodes propose the message that they want to send next, and then decide on the next message to be delivered in the total order [67].

So, total order broadcast is equivalent to repeated rounds of consensus (each consensus decision corresponding to one message delivery):

- Due to the agreement property of consensus, all nodes decide to deliver the same messages in the same order.
- Due to the integrity property, messages are not duplicated.
- Due to the validity property, messages are not corrupted and not fabricated out of thin air.
- Due to the termination property, messages are not lost.

Viewstamped Replication, Raft, and Zab implement total order broadcast directly, because that is more efficient than doing repeated rounds of one-value-at-a-time consensus. In the case of Paxos, this optimization is known as Multi-Paxos.

### Single-leader replication and consensus

In Chapter 5 we discussed single-leader replication (see "Leaders and Followers" on page 152), which takes all the writes to the leader and applies them to the followers in the same order, thus keeping replicas up to date. Isn't this essentially total order broadcast? How come we didn't have to worry about consensus in Chapter 5?

The answer comes down to how the leader is chosen. If the leader is manually chosen and configured by the humans in your operations team, you essentially have a "consensus algorithm" of the dictatorial variety: only one node is allowed to accept writes (i.e., make decisions about the order of writes in the replication log), and if that node goes down, the system becomes unavailable for writes until the operators manually configure a different node to be the leader. Such a system can work well in practice, but it does not satisfy the termination property of consensus because it requires human intervention in order to make progress.

Some databases perform automatic leader election and failover, promoting a follower to be the new leader if the old leader fails (see "Handling Node Outages" on page 156). This brings us closer to fault-tolerant total order broadcast, and thus to solving consensus.

However, there is a problem. We previously discussed the problem of split brain, and said that all nodes need to agree who the leader is—otherwise two different nodes could each believe themselves to be the leader, and consequently get the database into an inconsistent state. Thus, we need consensus in order to elect a leader. But if the consensus algorithms described here are actually total order broadcast algorithms, and total order broadcast is like single-leader replication, and single-leader replication requires a leader, then…

It seems that in order to elect a leader, we first need a leader. In order to solve consensus, we must first solve consensus. How do we break out of this conundrum?

### Epoch numbering and quorums

All of the consensus protocols discussed so far internally use a leader in some form or another, but they don't guarantee that the leader is unique. Instead, they can make a weaker guarantee: the protocols define an *epoch number* (called the *ballot number* in Paxos, *view number* in Viewstamped Replication, and *term number* in Raft) and guarantee that within each epoch, the leader is unique.

Every time the current leader is thought to be dead, a vote is started among the nodes to elect a new leader. This election is given an incremented epoch number, and thus epoch numbers are totally ordered and monotonically increasing. If there is a conflict between two different leaders in two different epochs (perhaps because the previous leader actually wasn't dead after all), then the leader with the higher epoch number prevails.

Before a leader is allowed to decide anything, it must first check that there isn't some other leader with a higher epoch number which might take a conflicting decision. How does a leader know that it hasn't been ousted by another node? Recall "The Truth Is Defined by the Majority" on page 300: a node cannot necessarily trust its own judgment—just because a node thinks that it is the leader, that does not necessarily mean the other nodes accept it as their leader.

Instead, it must collect votes from a *quorum* of nodes (see "Quorums for reading and writing" on page 179). For every decision that a leader wants to make, it must send the proposed value to the other nodes and wait for a quorum of nodes to respond in favor of the proposal. The quorum typically, but not always, consists of a majority of nodes [105]. A node votes in favor of a proposal only if it is not aware of any other leader with a higher epoch.

Thus, we have two rounds of voting: once to choose a leader, and a second time to vote on a leader's proposal. The key insight is that the quorums for those two votes must overlap: if a vote on a proposal succeeds, at least one of the nodes that voted for it must have also participated in the most recent leader election [105]. Thus, if the vote on a proposal does not reveal any higher-numbered epoch, the current leader can conclude that no leader election with a higher epoch number has happened, and therefore be sure that it still holds the leadership. It can then safely decide the proposed value.

This voting process looks superficially similar to two-phase commit. The biggest differences are that in 2PC the coordinator is not elected, and that fault-tolerant consensus algorithms only require votes from a majority of nodes, whereas 2PC requires a "yes" vote from *every* participant. Moreover, consensus algorithms define a recovery process by which nodes can get into a consistent state after a new leader is elected, ensuring that the safety properties are always met. These differences are key to the correctness and fault tolerance of a consensus algorithm.

### Limitations of consensus

Consensus algorithms are a huge breakthrough for distributed systems: they bring concrete safety properties (agreement, integrity, and validity) to systems where everything else is uncertain, and they nevertheless remain fault-tolerant (able to make progress as long as a majority of nodes are working and reachable). They provide total order broadcast, and therefore they can also implement linearizable atomic operations in a fault-tolerant way (see "Implementing linearizable storage using total order broadcast" on page 350).

Nevertheless, they are not used everywhere, because the benefits come at a cost.

The process by which nodes vote on proposals before they are decided is a kind of synchronous replication. As discussed in "Synchronous Versus Asynchronous Replication" on page 153, databases are often configured to use asynchronous replication. In this configuration, some committed data can potentially be lost on failover—but many people choose to accept this risk for the sake of better performance.

Consensus systems always require a strict majority to operate. This means you need a minimum of three nodes in order to tolerate one failure (the remaining two out of three form a majority), or a minimum of five nodes to tolerate two failures (the remaining three out of five form a majority). If a network failure cuts off some nodes from the rest, only the majority portion of the network can make progress, and the rest is blocked (see also "The Cost of Linearizability" on page 335).

Most consensus algorithms assume a fixed set of nodes that participate in voting, which means that you can't just add or remove nodes in the cluster. *Dynamic membership* extensions to consensus algorithms allow the set of nodes in the cluster to change over time, but they are much less well understood than static membership algorithms.

Consensus systems generally rely on timeouts to detect failed nodes. In environments with highly variable network delays, especially geographically distributed systems, it often happens that a node falsely believes the leader to have failed due to a transient network issue. Although this error does not harm the safety properties, frequent leader elections result in terrible performance because the system can end up spending more time choosing a leader than doing any useful work.

Sometimes, consensus algorithms are particularly sensitive to network problems. For example, Raft has been shown to have unpleasant edge cases [106]: if the entire network is working correctly except for one particular network link that is consistently unreliable, Raft can get into situations where leadership continually bounces between two nodes, or the current leader is continually forced to resign, so the system effectively never makes progress. Other consensus algorithms have similar problems, and designing algorithms that are more robust to unreliable networks is still an open research problem.

## Membership and Coordination Services

Projects like ZooKeeper or etcd are often described as "distributed key-value stores" or "coordination and configuration services." The API of such a service looks pretty much like that of a database: you can read and write the value for a given key, and iterate over keys. So if they're basically databases, why do they go to all the effort of implementing a consensus algorithm? What makes them different from any other kind of database?

To understand this, it is helpful to briefly explore how a service like ZooKeeper is used. As an application developer, you will rarely need to use ZooKeeper directly, because it is actually not well suited as a general-purpose database. It is more likely that you will end up relying on it indirectly via some other project: for example, HBase, Hadoop YARN, OpenStack Nova, and Kafka all rely on ZooKeeper running in the background. What is it that these projects get from it?

ZooKeeper and etcd are designed to hold small amounts of data that can fit entirely in memory (although they still write to disk for durability)—so you wouldn't want to store all of your application's data here. That small amount of data is replicated across all the nodes using a fault-tolerant total order broadcast algorithm. As discussed previously, total order broadcast is just what you need for database replication: if each message represents a write to the database, applying the same writes in the same order keeps replicas consistent with each other.

ZooKeeper is modeled after Google's Chubby lock service [14, 98], implementing not only total order broadcast (and hence consensus), but also an interesting set of other features that turn out to be particularly useful when building distributed systems:

*Linearizable atomic operations*
: Using an atomic compare-and-set operation, you can implement a lock: if several nodes concurrently try to perform the same operation, only one of them will succeed. The consensus protocol guarantees that the operation will be atomic and linearizable, even if a node fails or the network is interrupted at any point. A distributed lock is usually implemented as a *lease*, which has an expiry time so that it is eventually released in case the client fails (see "Process Pauses" on page 295).

*Total ordering of operations*
: As discussed in "The leader and the lock" on page 301, when some resource is protected by a lock or lease, you need a *fencing token* to prevent clients from conflicting with each other in the case of a process pause. The fencing token is some number that monotonically increases every time the lock is acquired. ZooKeeper provides this by totally ordering all operations and giving each operation a monotonically increasing transaction ID (`zxid`) and version number (`cversion`) [15].

*Failure detection*
: Clients maintain a long-lived session on ZooKeeper servers, and the client and server periodically exchange heartbeats to check that the other node is still alive. Even if the connection is temporarily interrupted, or a ZooKeeper node fails, the session remains active. However, if the heartbeats cease for a duration that is longer than the session timeout, ZooKeeper declares the session to be dead. Any locks held by a session can be configured to be automatically released when the session times out (ZooKeeper calls these *ephemeral nodes*).

*Change notifications*
: Not only can one client read locks and values that were created by another client, but it can also watch them for changes. Thus, a client can find out when another client joins the cluster (based on the value it writes to ZooKeeper), or if another client fails (because its session times out and its ephemeral nodes disappear). By subscribing to notifications, a client avoids having to frequently poll to find out about changes.

Of these features, only the linearizable atomic operations really require consensus. However, it is the combination of these features that makes systems like ZooKeeper so useful for distributed coordination.

### Allocating work to nodes

One example in which the ZooKeeper/Chubby model works well is if you have several instances of a process or service, and one of them needs to be chosen as leader or primary. If the leader fails, one of the other nodes should take over. This is of course useful for single-leader databases, but it's also useful for job schedulers and similar stateful systems.

Another example arises when you have some partitioned resource (database, message streams, file storage, distributed actor system, etc.) and need to decide which partition to assign to which node. As new nodes join the cluster, some of the partitions need to be moved from existing nodes to the new nodes in order to rebalance the load (see "Rebalancing Partitions" on page 209). As nodes are removed or fail, other nodes need to take over the failed nodes' work.

These kinds of tasks can be achieved by judicious use of atomic operations, ephemeral nodes, and notifications in ZooKeeper. If done correctly, this approach allows the application to automatically recover from faults without human intervention. It's not easy, despite the appearance of libraries such as Apache Curator [17] that have sprung up to provide higher-level tools on top of the ZooKeeper client API—but it is still much better than attempting to implement the necessary consensus algorithms from scratch, which has a poor success record [107].

An application may initially run only on a single node, but eventually may grow to thousands of nodes. Trying to perform majority votes over so many nodes would be terribly inefficient. Instead, ZooKeeper runs on a fixed number of nodes (usually three or five) and performs its majority votes among those nodes while supporting a potentially large number of clients. Thus, ZooKeeper provides a way of "outsourcing" some of the work of coordinating nodes (consensus, operation ordering, and failure detection) to an external service.

Normally, the kind of data managed by ZooKeeper is quite slow-changing: it represents information like "the node running on IP address 10.1.1.23 is the leader for partition 7," and such assignments usually change on a timescale of minutes or hours. ZooKeeper is not intended for storing the runtime state of the application, which may change thousands or even millions of times per second. If application state needs to be replicated from one node to another, other tools (such as Apache BookKeeper [108]) can be used.

### Service discovery

ZooKeeper, etcd, and Consul are also often used for *service discovery*—that is, to find out which IP address you need to connect to in order to reach a particular service. In cloud datacenter environments, where it is common for virtual machines to continually come and go, you often don't know the IP addresses of your services ahead of time. Instead, you can configure your services such that when they start up they register their network endpoints in a service registry, where they can then be found by other services.

However, it is less clear whether service discovery actually requires consensus. DNS is the traditional way of looking up the IP address for a service name, and it uses multiple layers of caching to achieve good performance and availability. Reads from DNS are absolutely not linearizable, and it is usually not considered problematic if the results from a DNS query are a little stale [109]. It is more important that DNS is reliably available and robust to network interruptions.

Although service discovery does not require consensus, leader election does. Thus, if your consensus system already knows who the leader is, then it can make sense to also use that information to help other services discover who the leader is. For this purpose, some consensus systems support read-only caching replicas. These replicas asynchronously receive the log of all decisions of the consensus algorithm, but do not actively participate in voting. They are therefore able to serve read requests that do not need to be linearizable.

### Membership services

ZooKeeper and friends can be seen as part of a long history of research into *membership services*, which goes back to the 1980s and has been important for building highly reliable systems, e.g., for air traffic control [110].

A membership service determines which nodes are currently active and live members of a cluster. As we saw throughout Chapter 8, due to unbounded network delays it's not possible to reliably detect whether another node has failed. However, if you couple failure detection with consensus, nodes can come to an agreement about which nodes should be considered alive or not.

It could still happen that a node is incorrectly declared dead by consensus, even though it is actually alive. But it is nevertheless very useful for a system to have agreement on which nodes constitute the current membership. For example, choosing a leader could mean simply choosing the lowest-numbered among the current members, but this approach would not work if different nodes have divergent opinions on who the current members are.

## Summary

In this chapter we examined the topics of consistency and consensus from several different angles. We looked in depth at linearizability, a popular consistency model: its goal is to make replicated data appear as though there were only a single copy, and to make all operations act on it atomically. Although linearizability is appealing because it is easy to understand—it makes a database behave like a variable in a single-threaded program—it has the downside of being slow, especially in environments with large network delays.

We also explored causality, which imposes an ordering on events in a system (what happened before what, based on cause and effect). Unlike linearizability, which puts all operations in a single, totally ordered timeline, causality provides us with a weaker consistency model: some things can be concurrent, so the version history is like a timeline with branching and merging. Causal consistency does not have the coordination overhead of linearizability and is much less sensitive to network problems.

However, even if we capture the causal ordering (for example using Lamport timestamps), we saw that some things cannot be implemented this way: in "Timestamp ordering is not sufficient" on page 347 we considered the example of ensuring that a username is unique and rejecting concurrent registrations for the same username. If one node is going to accept a registration, it needs to somehow know that another node isn't concurrently in the process of registering the same name. This problem led us toward *consensus*.

We saw that achieving consensus means deciding something in such a way that all nodes agree on what was decided, and such that the decision is irrevocable. With

some digging, it turns out that a wide range of problems are actually reducible to consensus and are equivalent to each other (in the sense that if you have a solution for one of them, you can easily transform it into a solution for one of the others). Such equivalent problems include:

*Linearizable compare-and-set registers*
: The register needs to atomically *decide* whether to set its value, based on whether its current value equals the parameter given in the operation.

*Atomic transaction commit*
: A database must *decide* whether to commit or abort a distributed transaction.

*Total order broadcast*
: The messaging system must *decide* on the order in which to deliver messages.

*Locks and leases*
: When several clients are racing to grab a lock or lease, the lock *decides* which one successfully acquired it.

*Membership/coordination service*
: Given a failure detector (e.g., timeouts), the system must *decide* which nodes are alive, and which should be considered dead because their sessions timed out.

*Uniqueness constraint*
: When several transactions concurrently try to create conflicting records with the same key, the constraint must *decide* which one to allow and which should fail with a constraint violation.

All of these are straightforward if you only have a single node, or if you are willing to assign the decision-making capability to a single node. This is what happens in a single-leader database: all the power to make decisions is vested in the leader, which is why such databases are able to provide linearizable operations, uniqueness constraints, a totally ordered replication log, and more.

However, if that single leader fails, or if a network interruption makes the leader unreachable, such a system becomes unable to make any progress. There are three ways of handling that situation:

1. Wait for the leader to recover, and accept that the system will be blocked in the meantime. Many XA/JTA transaction coordinators choose this option. This approach does not fully solve consensus because it does not satisfy the termination property: if the leader does not recover, the system can be blocked forever.
2. Manually fail over by getting humans to choose a new leader node and reconfigure the system to use it. Many relational databases take this approach. It is a kind of consensus by "act of God"—the human operator, outside of the computer sys-

tem, makes the decision. The speed of failover is limited by the speed at which humans can act, which is generally slower than computers.

3. Use an algorithm to automatically choose a new leader. This approach requires a consensus algorithm, and it is advisable to use a proven algorithm that correctly handles adverse network conditions [107].

Although a single-leader database can provide linearizability without executing a consensus algorithm on every write, it still requires consensus to maintain its leadership and for leadership changes. Thus, in some sense, having a leader only "kicks the can down the road": consensus is still required, only in a different place, and less frequently. The good news is that fault-tolerant algorithms and systems for consensus exist, and we briefly discussed them in this chapter.

Tools like ZooKeeper play an important role in providing an "outsourced" consensus, failure detection, and membership service that applications can use. It's not easy to use, but it is much better than trying to develop your own algorithms that can withstand all the problems discussed in Chapter 8. If you find yourself wanting to do one of those things that is reducible to consensus, and you want it to be fault-tolerant, then it is advisable to use something like ZooKeeper.

Nevertheless, not every system necessarily requires consensus: for example, leaderless and multi-leader replication systems typically do not use global consensus. The conflicts that occur in these systems (see "Handling Write Conflicts" on page 171) are a consequence of not having consensus across different leaders, but maybe that's okay: maybe we simply need to cope without linearizability and learn to work better with data that has branching and merging version histories.

This chapter referenced a large body of research on the theory of distributed systems. Although the theoretical papers and proofs are not always easy to understand, and sometimes make unrealistic assumptions, they are incredibly valuable for informing practical work in this field: they help us reason about what can and cannot be done, and help us find the counterintuitive ways in which distributed systems are often flawed. If you have the time, the references are well worth exploring.

This brings us to the end of Part II of this book, in which we covered replication (Chapter 5), partitioning (Chapter 6), transactions (Chapter 7), distributed system failure models (Chapter 8), and finally consistency and consensus (Chapter 9). Now that we have laid a firm foundation of theory, in Part III we will turn once again to more practical systems, and discuss how to build powerful applications from heterogeneous building blocks.

## References


[1] Peter Bailis and Ali Ghodsi: "Eventual Consistency Today: Limitations, Extensions, and Beyond," *ACM Queue*, volume 11, number 3, pages 55-63, March 2013. doi:10.1145/2460276.2462076

[2] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin: "Consistency, Availability, and Convergence," University of Texas at Austin, Department of Computer Science, Tech Report UTCS TR-11-22, May 2011.

[3] Alex Scotti: "Adventures in Building Your Own Database," at *All Your Base*, November 2015.

[4] Peter Bailis, Aaron Davidson, Alan Fekete, et al.: "Highly Available Transactions: Virtues and Limitations," at *40th International Conference on Very Large Data Bases* (VLDB), September 2014. Extended version published as pre-print arXiv:1302.0309 [cs.DB].

[5] Paolo Viotti and Marko Vukolić: "Consistency in Non-Transactional Distributed Storage Systems," arXiv:1512.00168, 12 April 2016.

[6] Maurice P. Herlihy and Jeannette M. Wing: "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 12, number 3, pages 463–492, July 1990. doi:10.1145/78969.78972

[7] Leslie Lamport: "On interprocess communication," *Distributed Computing*, volume 1, number 2, pages 77–101, June 1986. doi:10.1007/BF01786228

[8] David K. Gifford: "Information Storage in a Decentralized Computer System," Xerox Palo Alto Research Centers, CSL-81-8, June 1981.

[9] Martin Kleppmann: "Please Stop Calling Databases CP or AP," *martin.kleppmann.com*, May 11, 2015.

[10] Kyle Kingsbury: "Call Me Maybe: MongoDB Stale Reads," *aphyr.com*, April 20, 2015.

[11] Kyle Kingsbury: "Computational Techniques in Knossos," *aphyr.com*, May 17, 2014.

[12] Peter Bailis: "Linearizability Versus Serializability," *bailis.org*, September 24, 2014.

[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 *research.microsoft.com*.

[14] Mike Burrows: "The Chubby Lock Service for Loosely-Coupled Distributed Systems," at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

[15] Flavio P. Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013. ISBN: 978-1-449-36130-3

[16] "etcd Documentation," The Linux Foundation, *etcd.io*.

[17] "Apache Curator," Apache Software Foundation, *curator.apache.org*, 2015.

[18] Morali Vallath: *Oracle 10g RAC Grid, Services & Clustering*. Elsevier Digital Press, 2006. ISBN: 978-1-555-58321-7

[19] Peter Bailis, Alan Fekete, Michael J Franklin, et al.: "Coordination-Avoiding Database Systems," *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.

[20] Kyle Kingsbury: "Call Me Maybe: etcd and Consul," *aphyr.com*, June 9, 2014.

[21] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini: "Zab: High-Performance Broadcast for Primary-Backup Systems," at *41st IEEE International Conference on Dependable Systems and Networks* (DSN), June 2011. doi:10.1109/DSN.2011.5958223

[22] Diego Ongaro and John K. Ousterhout: "In Search of an Understandable Consensus Algorithm," at *USENIX Annual Technical Conference* (ATC), June 2014.

[23] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev: "Sharing Memory Robustly in Message-Passing Systems," *Journal of the ACM*, volume 42, number 1, pages 124–142, January 1995. doi:10.1145/200836.200869

[24] Nancy Lynch and Alex Shvartsman: "Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts," at *27th Annual International Symposium on Fault-Tolerant Computing* (FTCS), June 1997. doi:10.1109/FTCS.1997.614100

[25] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues: *Introduction to Reliable and Secure Distributed Programming*, 2nd edition. Springer, 2011. ISBN: 978-3-642-15259-7, doi:10.1007/978-3-642-15260-3

[26] Sam Elliott, Mark Allen, and Martin Kleppmann: personal communication, thread on *twitter.com*, October 15, 2015.

[27] Niklas Ekström, Mikhail Panchenko, and Jonathan Ellis: "Possible Issue with Read Repair?," email thread on *cassandra-dev* mailing list, October 2012.

[28] Maurice P. Herlihy: "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 13, number 1, pages 124–149, January 1991. doi:10.1145/114005.102808

[29] Armando Fox and Eric A. Brewer: "Harvest, Yield, and Scalable Tolerant Systems," at *7th Workshop on Hot Topics in Operating Systems* (HotOS), March 1999. doi:10.1109/HOTOS.1999.798396

[30] Seth Gilbert and Nancy Lynch: "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News*, volume 33, number 2, pages 51–59, June 2002. doi:10.1145/564585.564601

[31] Seth Gilbert and Nancy Lynch: "Perspectives on the CAP Theorem," *IEEE Computer Magazine*, volume 45, number 2, pages 30–36, February 2012. doi:10.1109/MC.2011.389

[32] Eric A. Brewer: "CAP Twelve Years Later: How the 'Rules' Have Changed," *IEEE Computer Magazine*, volume 45, number 2, pages 23–29, February 2012. doi:10.1109/MC.2012.37

[33] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen: "Consistency in Partitioned Networks," *ACM Computing Surveys*, volume 17, number 3, pages 341–370, September 1985. doi:10.1145/5505.5508

[34] Paul R. Johnson and Robert H. Thomas: "RFC 677: The Maintenance of Duplicate Databases," Network Working Group, January 27, 1975.

[35] Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, et al.: "Notes on Distributed Databases," IBM Research, Research Report RJ2571(33471), July 1979.

[36] Michael J. Fischer and Alan Michael: "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," at *1st ACM Symposium on Principles of Database Systems* (PODS), March 1982. doi:10.1145/588111.588124

[37] Eric A. Brewer: "NoSQL: Past, Present, Future," at *QCon San Francisco*, November 2012.

[38] Henry Robinson: "CAP Confusion: Problems with 'Partition Tolerance,'" *blog.cloudera.com*, April 26, 2010.

[39] Adrian Cockcroft: "Migrating to Microservices," at *QCon London*, March 2014.

[40] Martin Kleppmann: "A Critique of the CAP Theorem," arXiv:1509.05393, September 17, 2015.

[41] Nancy A. Lynch: "A Hundred Impossibility Proofs for Distributed Computing," at *8th ACM Symposium on Principles of Distributed Computing* (PODC), August 1989. doi:10.1145/72981.72982

[42] Hagit Attiya, Faith Ellen, and Adam Morrison: "Limitations of Highly-Available Eventually-Consistent Data Stores," at *ACM Symposium on Principles of Distributed Computing* (PODC), July 2015. doi:10.1145/2767386.2767419

[43] Peter Sewell, Susmit Sarkar, Scott Owens, et al.: "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Communications of the ACM*, volume 53, number 7, pages 89–97, July 2010. doi:10.1145/1785414.1785443

[44] Martin Thompson: "Memory Barriers/Fences," *mechanical-sympathy.blogspot.co.uk*, July 24, 2011.

[45] Ulrich Drepper: "What Every Programmer Should Know About Memory," *akkadia.org*, November 21, 2007.

[46] Daniel J. Abadi: "Consistency Tradeoffs in Modern Distributed Database System Design," *IEEE Computer Magazine*, volume 45, number 2, pages 37–42, February 2012. doi:10.1109/MC.2012.33

[47] Hagit Attiya and Jennifer L. Welch: "Sequential Consistency Versus Linearizability," *ACM Transactions on Computer Systems* (TOCS), volume 12, number 2, pages 91–122, May 1994. doi:10.1145/176575.176576

[48] Mustaque Ahamad, Gil Neiger, James E. Burns, et al.: "Causal Memory: Definitions, Implementation, and Programming," *Distributed Computing*, volume 9, number 1, pages 37–49, March 1995. doi:10.1007/BF01784241

[49] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen: "Stronger Semantics for Low-Latency Geo-Replicated Storage," at *10th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2013.

[50] Marek Zawirski, Annette Bieniusa, Valter Balegas, et al.: "SwiftCloud: Fault-Tolerant Geo-Replication Integrated All the Way to the Client Machine," INRIA Research Report 8347, August 2013.

[51] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica: "Bolt-on Causal Consistency," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

[52] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: "Challenges to Adopting Stronger Consistency at Scale," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

[53] Peter Bailis: "Causality Is Expensive (and What to Do About It)," *bailis.org*, February 5, 2014.

[54] Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte: "Concise Server-Wide Causality Management for Eventually Consistent Data Stores," at *15th IFIP International Conference on Distributed Applications and Interoperable Systems* (DAIS), June 2015. doi:10.1007/978-3-319-19129-4_6

[55] Rob Conery: "A Better ID Generator for PostgreSQL," *rob.conery.io*, May 29, 2014.

[56] Leslie Lamport: "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563

[57] Xavier Défago, André Schiper, and Péter Urbán: "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," *ACM Computing Surveys*, volume 36, number 4, pages 372–421, December 2004. doi:10.1145/1041680.1041682

[58] Hagit Attiya and Jennifer Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, 2nd edition. John Wiley & Sons, 2004. ISBN: 978-0-471-45324-6, doi:10.1002/0471478210

[59] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, et al.: "CORFU: A Shared Log Design for Flash Clusters," at *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.

[60] Fred B. Schneider: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, volume 22, number 4, pages 299–319, December 1990.

[61] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, et al.: "Calvin: Fast Distributed Transactions for Partitioned Database Systems," at *ACM International Conference on Management of Data* (SIGMOD), May 2012.

[62] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, et al.: "Tango: Distributed Data Structures over a Shared Log," at *24th ACM Symposium on Operating Systems Principles* (SOSP), November 2013. doi:10.1145/2517349.2522732

[63] Robbert van Renesse and Fred B. Schneider: "Chain Replication for Supporting High Throughput and Availability," at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

[64] Leslie Lamport: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, volume 28, number 9, pages 690–691, September 1979. doi:10.1109/TC.1979.1675439

[65] Enis Söztutar, Devaraj Das, and Carter Shanklin: "Apache HBase High Availability at the Next Level," *hortonworks.com*, January 22, 2015.

[66] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, et al.: "PNUTS: Yahoo!'s Hosted Data Serving Platform," at *34th International Conference on Very Large Data Bases* (VLDB), August 2008. doi:10.14778/1454159.1454167

[67] Tushar Deepak Chandra and Sam Toueg: "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, volume 43, number 2, pages 225–267, March 1996. doi:10.1145/226643.226647

[68] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson: "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, volume 32, number 2, pages 374–382, April 1985. doi:10.1145/3149.214121

[69] Michael Ben-Or: "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," at *2nd ACM Symposium on Principles of Distributed Computing* (PODC), August 1983. doi:10.1145/800221.806707

[70] Jim N. Gray and Leslie Lamport: "Consensus on Transaction Commit," *ACM Transactions on Database Systems* (TODS), volume 31, number 1, pages 133–160, March 2006. doi:10.1145/1132863.1132867

[71] Rachid Guerraoui: "Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus," at *9th International Workshop on Distributed Algorithms* (WDAG), September 1995. doi:10.1007/BFb0022140

[72] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, et al.: "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.

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

[74] Hector Garcia-Molina and Kenneth Salem: "Sagas," at *ACM International Conference on Management of Data* (SIGMOD), May 1987. doi:10.1145/38713.38742

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

[76] "Distributed Transaction Processing: The XA Specification," X/Open Company Ltd., Technical Standard XO/CAE/91/300, December 1991. ISBN: 978-1-872-63024-3

[77] Mike Spille: "XA Exposed, Part II," *jroller.com* [URL inactive], April 3, 2004.

[78] 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

[79] 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

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

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

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

[83] Jonathan Oliver: "My Beef with MSDTC and Two-Phase Commits," *blog.jonathanoliver.com*, April 4, 2011.

[84] Oren Eini (Ahende Rahien): "The Fallacy of Distributed Transactions," *ayende.com*, July 17, 2014.

[85] Clemens Vasters: "Transactions in Windows Azure (with Service Bus) – An Email Discussion," *vasters.com*, July 30, 2012.

[86] "Understanding Transactionality in Azure," NServiceBus Documentation, Particular Software, 2015.

[87] Randy Wigginton, Ryan Lowe, Marcos Albe, and Fernando Ipar: "Distributed Transactions in MySQL," at *MySQL Conference and Expo*, April 2013.

[88] Mike Spille: "XA Exposed, Part I," *jroller.com*, April 3, 2004.

[89] Ajmer Dhariwal: "Orphaned MSDTC Transactions (-2 spids)," *eraofdata.com*, December 12, 2008.

[90] Paul Randal: "Real World Story of DBCC PAGE Saving the Day," *sqlskills.com*, June 19, 2013.

[91] "in-doubt xact resolution Server Configuration Option," SQL Server 2016 documentation, Microsoft, Inc., 2016.

[92] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:10.1145/42282.42283

[93] Miguel Castro and Barbara H. Liskov: "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems*, volume 20, number 4, pages 396–461, November 2002. doi:10.1145/571637.571640

[94] Brian M. Oki and Barbara H. Liskov: "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," at *7th ACM Symposium on Principles of Distributed Computing* (PODC), August 1988. doi:10.1145/62546.62549

[95] Barbara H. Liskov and James Cowling: "Viewstamped Replication Revisited," Massachusetts Institute of Technology, Tech Report MIT-CSAIL-TR-2012-021, July 2012.

[96] Leslie Lamport: "The Part-Time Parliament," *ACM Transactions on Computer Systems*, volume 16, number 2, pages 133–169, May 1998. doi:10.1145/279227.279229

[97] Leslie Lamport: "Paxos Made Simple," *ACM SIGACT News*, volume 32, number 4, pages 51–58, December 2001.

[98] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone: "Paxos Made Live – An Engineering Perspective," at *26th ACM Symposium on Principles of Distributed Computing* (PODC), June 2007.

[99] Robbert van Renesse: "Paxos Made Moderately Complex," *cs.cornell.edu*, March 2011.

[100] Diego Ongaro: "Consensus: Bridging Theory and Practice," PhD Thesis, Stanford University, August 2014.

[101] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft: "Raft Refloated: Do We Have Consensus?," *ACM SIGOPS Operating Systems Review*, volume 49, number 1, pages 12–21, January 2015. doi:10.1145/2723872.2723876

[102] André Medeiros: "ZooKeeper's Atomic Broadcast Protocol: Theory and Practice," Aalto University School of Science, March 20, 2012.

[103] Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider: "Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab," *IEEE Transactions on Dependable and Secure Computing*, volume 12, number 4, pages 472–484, September 2014. doi:10.1109/TDSC.2014.2355848

[104] Will Portnoy: "Lessons Learned from Implementing Paxos," *blog.willportnoy.com*, June 14, 2012.

[105] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman: "Flexible Paxos: Quorum Intersection Revisited," at *20th International Conference on Principles of Distributed Systems* (OPODIS), December 2016. doi:10.4230/LIPIcs.OPODIS.2016.25

[106] Heidi Howard and Jon Crowcroft: "Coracle: Evaluating Consensus at the Internet Edge," at *Annual Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM), August 2015. doi:10.1145/2829988.2790010

[107] Kyle Kingsbury: "Call Me Maybe: Elasticsearch 1.5.0," *aphyr.com*, April 27, 2015.

[108] Ivan Kelly: "BookKeeper Tutorial," *github.com*, October 2014.

[109] Camille Fournier: "Consensus Systems for the Skeptical Architect," at *Philly ETE*, Philadelphia, PA, USA, April 2014.

[110] Kenneth P. Birman: "A History of the Virtual Synchrony Replication Model," in *Replication: Theory and Practice*, Springer LNCS volume 5959, chapter 6, pages 91–120, 2010. ISBN: 978-3-642-11293-5, doi:10.1007/978-3-642-11294-2_6