Chapter 6. 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 "Distributed Versus Single-Node Systems", 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 and durability)
- 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 small enough that each machine can hold a copy of the entire dataset. In Chapter 7 we will relax that assumption and discuss *sharding* (*partitioning*) 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 families of 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. Despite being so old, concepts such as *eventual consistency* still cause confusion. In "Problems with Replication Lag" we will get more precise about eventual consistency and discuss things like the *read-your-writes* and *monotonic reads* guarantees.

BACKUPS AND REPLICATION

You might be wondering whether you still need backups if you have replication. The answer is yes, because they have different purposes: replicas quickly reflect writes from one node on other nodes, but backups store old snapshots of the data so that you can go back in time. If you accidentally delete some data, replication doesn't help since the deletion will have also been propagated to the replicas, so you need a backup if you want to restore the deleted data.

In fact, replication and backups are often complementary to each other. Backups are sometimes part of the process of setting up replication, as we shall see in "Setting Up New Followers". Conversely, archiving replication logs can be part of a backup process.

Some databases internally maintain immutable snapshots of past states, which serve as a kind of internal backup. However, this means keeping old versions of the data on the same storage media as the current state. If you have a large amount of data, it can be cheaper to keep the backups of old data in an object store that is optimized for infrequently-accessed data, and to store only the current state of the database in primary storage.

Single-Leader Replication

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 is called *leader-based replication*, *primary-backup*, or *active/passive*. It works as follows (see Figure 6-1):

1. One of the replicas is designated the *leader* (also known as *primary* or *source* [2]). 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*, *secondaries*, or *hot standbys*). 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.

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).

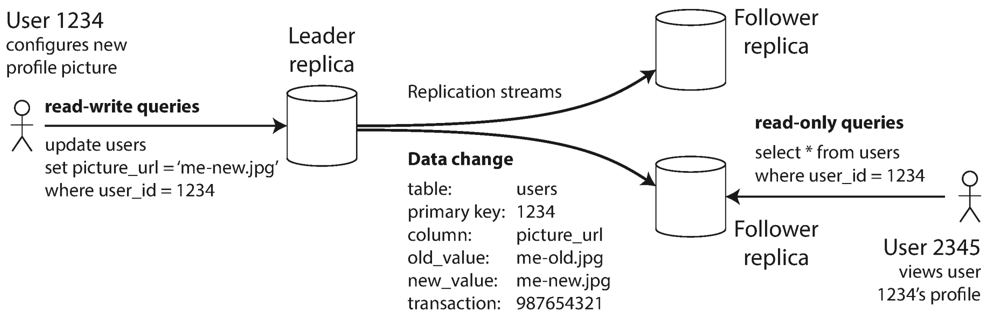


Figure 6-1. Single-leader replication directs all writes to a designated leader, which sends a stream of changes to the follower replicas.

If the database is sharded (see Chapter 7), each shard has one leader. Different shards may have their leaders on different nodes, but each shard must nevertheless have one leader node. In "Multi-Leader Replication" we will discuss an alternative model in which a system may have multiple leaders for the same shard at the same time.

Single-leader replication is very widely used. It's a built-in feature of many relational databases, such as PostgreSQL, MySQL, Oracle Data Guard [3], and SQL Server's Always On Availability Groups [4]. It is also used in some document databases such as MongoDB and DynamoDB [5], message brokers such as Kafka, replicated block devices such as DRBD, and some network filesystems. Many consensus algorithms such as Raft, which is used for replication in CockroachDB [6], TiDB [7], etcd, and RabbitMQ quorum queues (among others), are also based on a single leader, and automatically elect a new leader if the old one fails (we will discuss consensus in more detail in Chapter 10).

NOTE

In older documents you may see the term *master–slave replication*. It means the same as leader-based replication, but the term should be avoided as it is widely considered offensive [8].

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 6-1, 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 6-2 shows one possible way how the timings could work out.

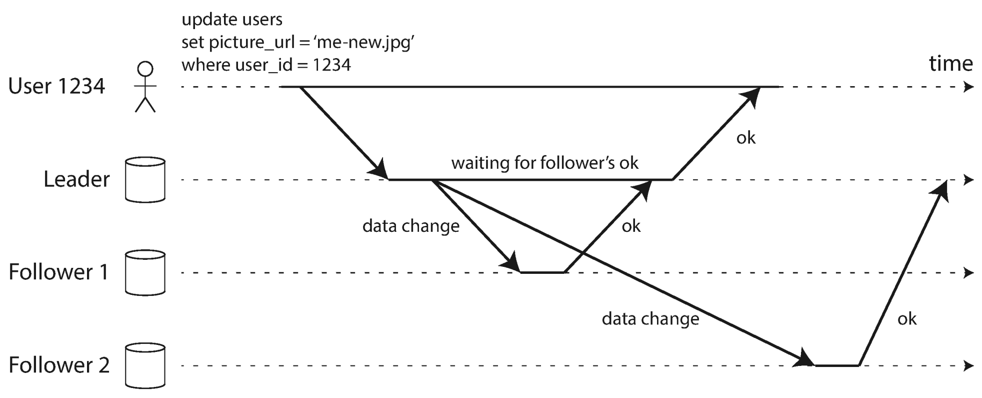


Figure 6-2. Leader-based replication with one synchronous and one asynchronous follower.

In the example of Figure 6-2, 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* (or *non-blocking*): 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 a database offers synchronous replication, it often 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*.

In some systems, a *majority* (e.g., 3 out of 5 replicas, including the leader) of replicas is updated synchronously, and the remaining minority is asynchronous. This is an example of a *quorum*, which we will discuss further in "Quorums for reading and writing". Majority quorums are often used in eventually consistent systems or systems that use a consensus protocol for automatic leader election. We will return to these systems in Chapter 10.

Sometimes, 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 [9]. We will return to this issue in "Problems with Replication Lag".

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 Percona XtraBackup for MySQL.
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*; MySQL has two mechanisms, *binlog coordinates* and *global transaction identifiers* (GTIDs).
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.

You can also archive the replication log to an object store along with periodic snapshots of the whole database. This is a good way of implementing database backups and disaster recovery. You can also perform steps 1 and 2 of setting up a new follower by downloading those files from the object store. For example, WAL-G does this for PostgreSQL, MySQL, and SQL Server, and Litestream does the equivalent for SQLite.

DATABASES BACKED BY OBJECT STORAGE

Object storage can be used for more than archiving data. Many databases are beginning to use object stores such as Amazon Web Services S3, Google Cloud Storage, and Azure Blob Storage to serve data for live queries. Storing database data in object storage has many benefits:

- Object storage is inexpensive compared to other cloud storage options, which allow cloud databases to store less-often queried data on cheaper, higher-latency storage while serving the working set from memory, SSDs, and NVMe.
- Object stores also provide multi-zone, dual-region, or multi-region replication with very high durability guarantees. This also allows databases to bypass inter-zone network fees.
- Databases can use an object store's *conditional write* feature—essentially, a *compare-and-set* (CAS) operation—to implement transactions and leadership election [10, 11]).
- Storing data from multiple databases in the same object store can simplify data integration (see "Cloud Data Warehouses"), particularly when open formats such as Apache Parquet and Apache Iceberg are used.

These benefits dramatically simplify the database architecture by shifting the responsibility of transactions, leadership election, and replication to object storage.

Systems that adopt object storage for replication must grapple with some tradeoffs. Notably, object stores have much higher read and write latencies than local disks or virtual block devices such as EBS. Many cloud providers also charge a per-API call fee, which forces systems to batch reads and writes to reduce cost. Such batching further increases latency. Objects are often immutable, as well, which makes random writes in a large object an extremely resource intensive operation. Moreover, many object stores do not offer standard filesystem interfaces. This prevents systems that lack object storage integration from leveraging object storage. Interfaces such as *filesystem in userspace* (FUSE) allow operators to mount object store buckets as filesystems that applications can use without knowing their data is stored on object storage. Still, many FUSE interfaces to object stores lack POSIX features such as non-sequential writes or symlinks, which systems might depend on.

Different systems deal with these trade-offs in various ways. Some introduce a *tiered storage* architecture that places less frequently accessed data on object storage while new or frequently accessed data is kept on faster storage devices such as SSDs, NVMe, or even in memory. Other systems use object storage as their primary storage tier, but use a separate low-latency storage system such as Amazon's EBS or Neon's Safekeepers [12]) to store their WAL. Recently, some systems have gone even farther by adopting a *zero-disk architecture* (ZDA). ZDA-based systems persist all data to object storage and use disks and memory strictly for caching. This allows nodes to have no persistent state, which dramatically simplifies operations. WarpStream, Confluent Freight, Buf's Bufstream, and Redpanda Serverless are all Kafka-compatible systems built using a zero-disk architecture. Nearly every modern cloud data warehouse also adopts such an architecture, as does Turbopuffer (a vector search engine), and SlateDB (a cloud-native LSM storage engine).

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.

Although follower recovery is conceptually simple, it can be challenging in terms of performance: if the database has a high write throughput or if the follower has been offline for a long time, there might be a lot of writes to catch up on. There will be high load on both the recovering follower and the leader (which needs to send the backlog of writes to the follower) while this catch-up is ongoing.

The leader can delete its log of writes once all followers have confirmed that they have processed it, but if a follower is unavailable for a long time, the leader faces a choice: either it retains the log until the follower recovers and catches up (at the risk of running out of disk space on the leader), or it deletes the log that the unavailable follower has not yet acknowledged (in which case the follower won't be able to recover from the log, and will have to be restored from a backup when it comes back).

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 since the leader can trigger a safe handoff before shutting down.)
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 established *controller node* [13]. 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 10.
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”). 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 means that writes you believed to be committed actually weren’t durable after all.
- 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 [14], 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 9), 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”), 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. However, if this mechanism is not carefully designed, you can end up with both nodes being shut down [15]. Moreover, there is a risk that by the time the split brain is detected and the old node is shut down, it is already too late and data has already been corrupted.
- 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.

NOTE

Guarding against split brain by limiting or shutting down old leaders is known as *fencing* or, more emphatically, *Shoot The Other Node In The Head* (STONITH). We will discuss fencing in more detail in "Distributed Locks and Leases".

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.

The most important thing with failover is to pick an up-to-date follower as the new leader—if synchronous or semi-synchronous replication is used, this would be the follower that the old leader waited for before acknowledging writes. With asynchronous replication, you can pick the follower with the greatest log sequence number. This minimizes the amount of data that is lost during failover: losing a fraction of a second of writes may be tolerable, but picking a follower that is behind by several days could be catastrophic.

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 9 and Chapter 10 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 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. The idea of executing deterministic statements in a fixed order is similar to the event sourcing model that we previously discussed in "Event Sourcing and CQRS". This approach is also known as *state machine replication*, and we will discuss the theory behind it in "Using shared logs".

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 [16]. However, determinism can be hard to guarantee in practice, so many databases prefer other replication methods.

Write-ahead log (WAL) shipping

In Chapter 4 we saw that a write-ahead log is needed to make B-tree storage engines robust: every modification is first written to the WAL so that the tree can be restored to a consistent state after a crash. Since the WAL contains all the information necessary to restore the indexes and heap into a consistent state, 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 files as found on the leader.

This method of replication is used in PostgreSQL and Oracle, among others [17, 18]. 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 tightly 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 keeps a separate logical replication log, called the *binlog*, in addition to the WAL (when configured to use row-based replication). PostgreSQL implements logical replication by decoding the physical WAL into row insertion/update/delete events [19].

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. This in turn enables upgrading to a new version with minimal downtime [20].

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 [21]. This technique is called *change data capture*, and we will return to it in Chapter 12.

Problems with Replication Lag

Being able to tolerate node failures is just one reason for wanting replication. As mentioned in "Distributed Versus Single-Node Systems", 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 (which is often the case with online services), 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].

NOTE

The term *eventual consistency* was coined by Douglas Terry et al. [23], popularized by Werner Vogels [24], 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.

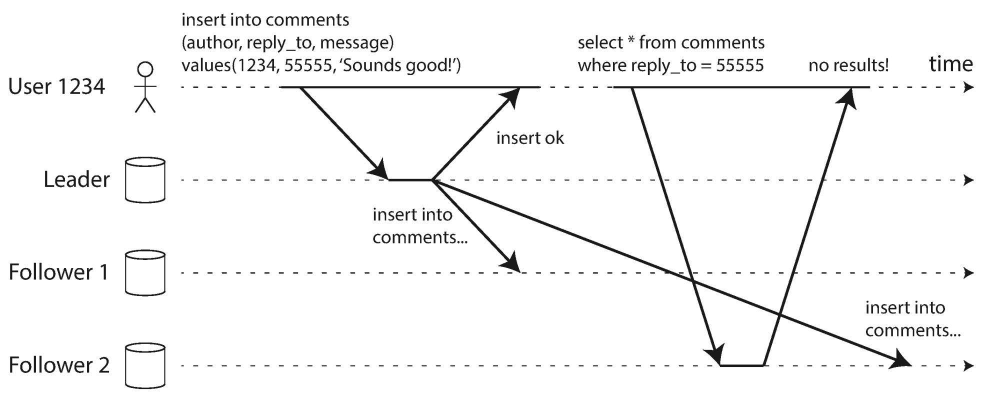
The term "eventually" is deliberately vague: in general, there is no limit to how far a replica can fall behind. In normal operation, the delay between a write happening on the leader and being reflected on a follower—the *replication lag*—may be only a fraction of a second, and not noticeable in practice. 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. We'll also 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 6-3: 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.



User 1234
Leader
Follower 1
Follower 2
insert into comments (author, reply_to, message) values(1234, 55555, 'Sounds good!')
insert ok
insert into comments...
select * from comments where reply_to = 55555
no results!
time
insert into comments...

Figure 6-3. 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* [23]. 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 or a synchronously updated follower; otherwise, read it from an asynchronously updated 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 [25]. 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 [26]. 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").
- If your replicas are distributed across regions (for geographical proximity to users, for availability, or for durability), there is additional complexity. Any request that needs to be served by the leader must be routed to the region 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 regions, there is no guarantee that connections from different devices will be routed to the same region. (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 region.

REGIONS AND AVAILABILITY ZONES

We use the term *region* to refer to one or more datacenters in a single geographic location. Cloud providers locate multiple datacenters in the same geographic region. Each datacenter is referred to as an *availability zone* or simply *zone*. Thus, a single cloud region is made up of multiple zones. Each zone is a separate datacenter located in separate physical facility with its own power, cooling, and so on.

Zones in the same region are connected by very high speed network connections. Latency is low enough that most distributed systems can run with nodes spread across multiple zones in the same region as though they were in a single zone. Multi-zone configurations allow distributed systems to survive zonal outages where one zone goes offline, but they do not protect against regional outages where all zones in a region are unavailable. To survive a regional outage, a distributed system must be deployed across multiple regions, which can result in higher latencies, lower throughput, and increased cloud networking bills. We will discuss these tradeoffs more in "Multi-leader replication topologies". For now, just know that when we say region, we mean a collection of zones/datacenters in a single geographic location.

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 6-4 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 comment. However, it's very confusing for user 2345 if they first see user 1234's comment appear, and then see it disappear again.

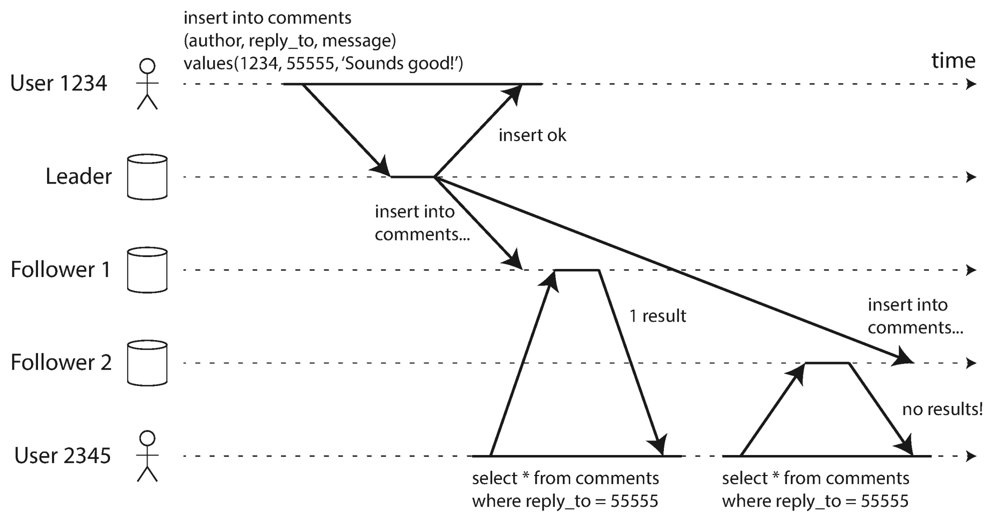


Figure 6-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

Monotonic reads [22] 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 6-5). 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 [27].

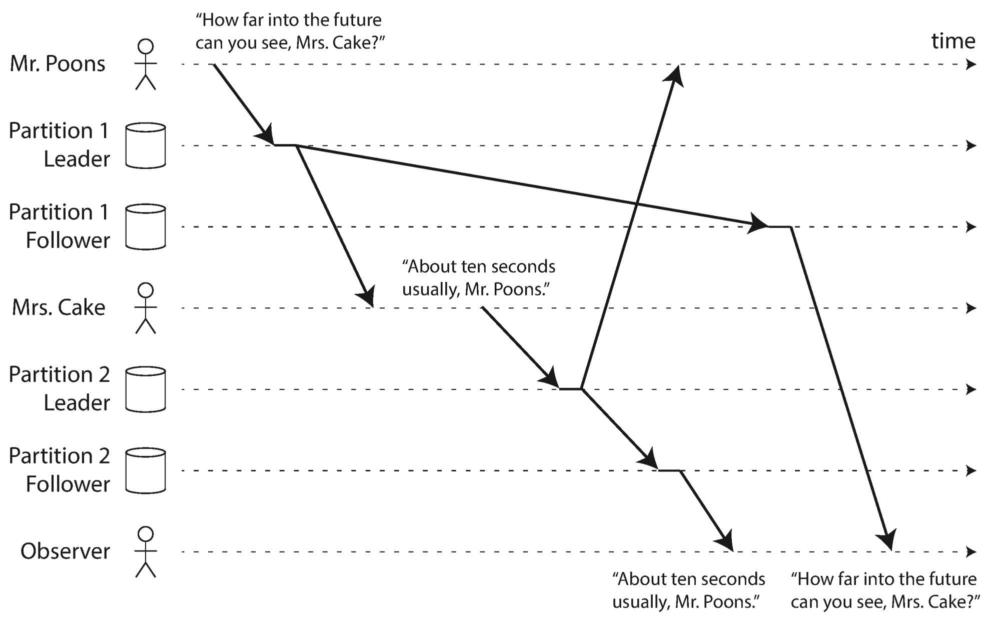

"How far into the future can you see, Mrs. Cake?"
time
Mr. Poons
Partition 1 Leader
Partition 1 Follower
"About ten seconds usually, Mr. Poons."
Mrs. Cake
Partition 2 Leader
Partition 2 Follower
Observer
"About ten seconds usually, Mr. Poons."
"How far into the future can you see, Mrs. Cake?"

Figure 6-5. If some shards 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* [22]. 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 sharded (partitioned) databases, which we will discuss in Chapter 7. 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 shards 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 shard—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” relation and concurrency”.

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 or a synchronously updated follower. However, dealing with these issues in application code is complex and easy to get wrong.

The simplest programming model for application developers is to choose a database that provides a strong consistency guarantee for replicas such as linearizability (see Chapter 10), and ACID transactions (see Chapter 8). This allows you to mostly ignore the challenges that arise from replication, and treat the database as if it had just a single node. In the early 2010s the *NoSQL* movement promoted the view that these features limited scalability, and that large-scale systems would have to embrace eventual consistency.

However, since then, a number of databases started providing strong consistency and transactions while also offering the fault tolerance, high availability, and scalability advantages of a distributed database. As mentioned in “Relational Model versus Document Model”, this trend is known as *NewSQL* to contrast with NoSQL (although it’s less about SQL specifically, and more about new approaches to scalable transaction management).

Even though scalable, strongly consistent distributed databases are now available, there are still good reasons why some applications choose to use different forms of replication that offer weaker consistency guarantees: they can offer stronger resilience in the face of network interruptions, and have lower overheads compared to transactional systems. We will explore such approaches in the rest of this chapter.

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.

Single-leader replication has one major downside: all writes must go through the one leader. 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 single-leader 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 *active/active* or *bidirectional* replication). In this setup, each leader simultaneously acts as a follower to the other leaders.

As with single-leader replication, there is a choice between making it synchronous or asynchronous. Let's say you have two leaders, *A* and *B*, and you're trying to write to *A*. If writes are synchronously replicated from *A* to *B*, and the network between the two nodes is interrupted, you can't write to *A* until the network comes back. Synchronous multi-leader replication thus gives you a model that is very similar to single-leader replication, i.e. if you had made *B* the leader and *A* simply forwards any write requests to *B* to be executed.

For that reason, we won't go further into synchronous multi-leader replication, and simply treat it as equivalent to single-leader replication. The rest of this section focusses on asynchronous multi-leader replication, in which any leader can process writes even when its connection to the other leaders is interrupted.

Geographically Distributed Operation

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

Imagine you have a database with replicas in several different regions (perhaps so that you can tolerate the failure of an entire region, or perhaps in order to be closer to your users). This is known as a *geographically distributed*, *geo-distributed* or *geo-replicated* setup. With single-leader replication, the leader has to be in *one* of the regions, and all writes must go through that region.

In a multi-leader configuration, you can have a leader in *each* region. Figure 6-6 shows what this architecture might look like. Within each region, regular leader–follower replication is used (with followers maybe in a different availability zone from the leader); between regions, each region's leader replicates its changes to the leaders in other regions.

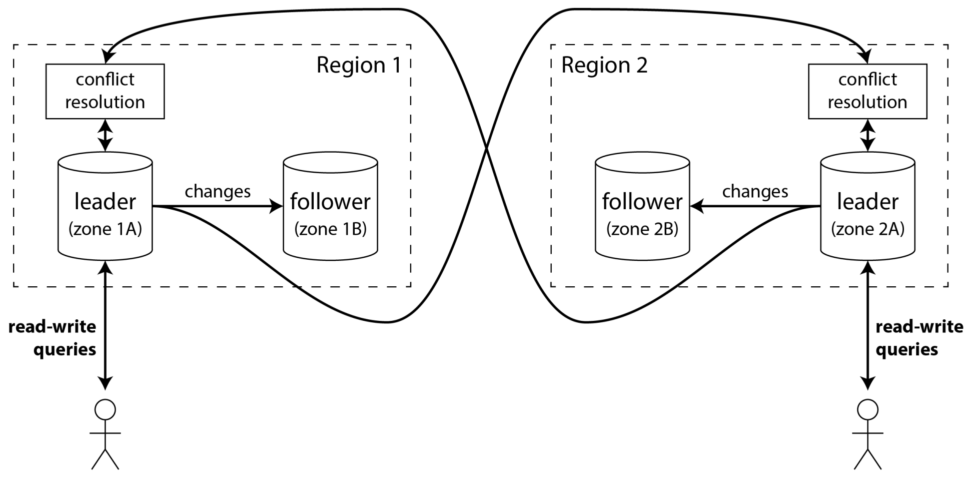


Figure 6-6. Multi-leader replication across multiple regions.

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

Performance

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

Tolerance of regional outages

In a single-leader configuration, if the region with the leader becomes unavailable, failover can promote a follower in another region to be leader. In a multi-leader configuration, each region can continue operating independently of the others, and replication catches up when the offline region comes back online.

Tolerance of network problems

Even with dedicated connections, traffic between regions can be less reliable than traffic between zones in the same region or within a single zone. A single-leader configuration is very sensitive to problems in this inter-region link, because when a client in one region wants to write to a leader in another region, it has to send its request over that link and wait for the response before it can complete.

A multi-leader configuration with asynchronous replication can tolerate network problems better: during a temporary network interruption, each region's leader can continue independently processing writes.

Consistency

A single-leader system can provide strong consistency guarantees, such as serializable transactions, which we will discuss in Chapter 8. The biggest downside of multi-leader systems is that the consistency they can achieve is much weaker. For example, you can't guarantee that a bank account won't go negative or that a username is unique: it's always possible for different leaders to process writes that are individually fine (paying out some of the money in an account, registering a particular username), but which violate the constraint when taken together with another write on another leader.

This is simply a fundamental limitation of distributed systems [28]. If you need to enforce such constraints, you're therefore better off with a single-leader system. However, as we will see in "Dealing with Conflicting Writes", multi-leader systems can still achieve consistency properties that are useful in a wide range of apps that don't need such constraints.

Multi-leader replication is less common than single-leader replication, but it is still supported by many databases, including MySQL, Oracle, SQL Server, and YugabyteDB. In some cases it is an external add-on feature, for example in Redis Enterprise, EDB Postgres Distributed, and pglogical [29].

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 [30].

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 6-9, 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 6-7.

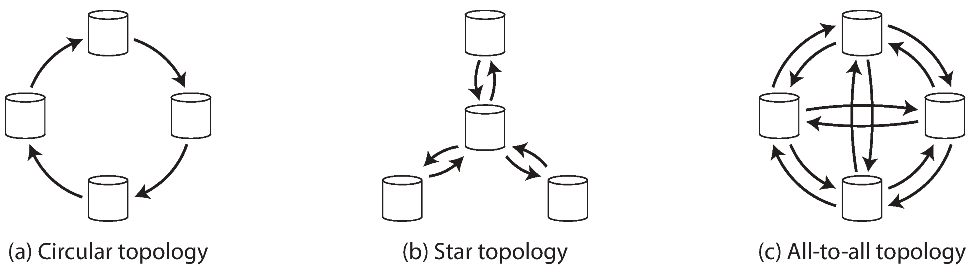


Figure 6-7. Three example topologies in which multi-leader replication can be set up.

The most general topology is *all-to-all*, shown in Figure 6-7(c), in which every leader sends its writes to every other leader. However, more restricted topologies are also used: for example a *circular topology* 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*: one designated root node forwards writes to all of the other nodes. The star topology can be generalized to a tree.

NOTE

A star-shaped network topology is unrelated to a *star schema* (see "Stars and Snowflakes: Schemas for Analytics"), which describes the structure of a data model.

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 [31]. When a node receives a data change that is tagged with its own identifier, that data change is ignored, because the node knows that it has already been processed.

Problems with different topologies

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, leaving them 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 6-8.

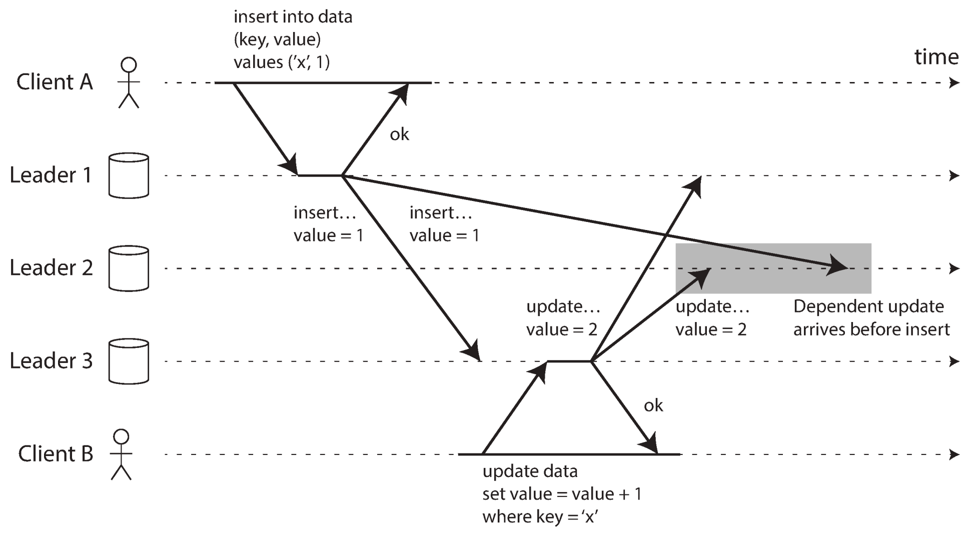


Figure 6-8. With multi-leader replication, writes may arrive in the wrong order at some replicas.

In Figure 6-8, 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": 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 9).

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"). However, many multi-leader replication systems don't use good techniques for ordering updates, leaving them vulnerable to issues like the one in Figure 6-8. If you are using 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.

Sync Engines and Local-First Software

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 replica 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 very similar to multi-leader replication between regions, taken to the extreme: each device is a "region," and the network connection between them is extremely unreliable.

Real-time collaboration, offline-first, and local-first apps

Moreover, many modern web apps offer *real-time collaboration* features, such as Google Docs and Sheets for text documents and spreadsheets, Figma for graphics, and Linear for project management. What makes these apps so responsive is that user input is immediately reflected in the user interface, without waiting for a network round-trip to the server, and edits by one user are shown to their collaborators with low latency [32, 33, 34].

This again results in a multi-leader architecture: each web browser tab that has opened the shared file is a replica, and any updates that you make to the file are asynchronously replicated to the devices of the other users who have opened the same file. Even if the app does not allow you to continue editing a file while offline, the fact that multiple users can make edits without waiting for a response from the server already makes it multi-leader.

Both offline editing and real-time collaboration require a similar replication infrastructure: the application needs to capture any changes that the user makes to a file, and either send them to collaborators immediately (if online), or store them locally for sending later (if offline). Additionally, the application needs to receive changes from collaborators, merge them into the user's local copy of the file, and update the user interface to reflect the latest version. If multiple users have changed the file concurrently, conflict resolution logic may be needed to merge those changes.

A software library that supports this process is called a *sync engine*. Although the idea has existed for a long time, the term has recently gained attention [35, 36, 37]. An application that allows a user to continue editing a file while offline (which may be implemented using a sync engine) is called *offline-first* [38]. The term *local-first software* refers to collaborative apps that are not only offline-first, but are also designed to continue working even if the developer who made the software shuts down all of their online services [39]. This can be achieved by using a sync engine with an open standard sync protocol for which multiple service providers are available [40]. For example, Git is a local-first collaboration system (albeit one that doesn't support real-time collaboration) since you can sync via GitHub, GitLab, or any other repository hosting service.

Pros and cons of sync engines

The dominant way of building web apps today is to keep very little persistent state on the client, and to rely on making requests to a server whenever a new piece of data needs to be displayed or some data needs to be updated. In contrast, when using a sync engine, you have persistent state on the client, and communication with the server is moved into a background process. The sync engine approach has a number of advantages:

- Having the data locally means the user interface can be much faster to respond than if it had to wait for a service call to fetch some data. Some apps aim to respond to user input in the *next frame* of the graphics system, which means rendering within 16 ms on a display with a 60 Hz refresh rate.
- Allowing users to continue working while offline is valuable, especially on mobile devices with intermittent connectivity. With a sync engine, an app doesn't need a separate offline mode: being offline is the same as having very large network delay.
- A sync engine simplifies the programming model for frontend apps, compared to performing explicit service calls in application code. Every service call requires error handling, as discussed in "The problems with remote procedure calls (RPCs)": for example, if a request to update data on a server fails, the user interface needs to somehow reflect that error. A sync engine allows the app to perform reads and writes on local data, which almost never fails, leading to a more declarative programming style [41].
- In order to display edits from other users in real-time, you need to receive notifications of those edits and efficiently update the user interface accordingly. A sync engine combined with a *reactive programming* model is a good way of implementing this [42].

Sync engines work best when all the data that the user may need is downloaded in advance and stored persistently on the client. This means that the data is available for offline access when needed, but it also means that sync engines are not suitable if the user has access to a very large amount of data. For example, downloading all the files that the user themselves created is probably fine (one user generally doesn't generate that much data), but downloading the entire catalog of an e-commerce website probably doesn't make sense.

The sync engine was pioneered by Lotus Notes in the 1980s [43] (without using that term), and sync for specific apps such as calendars has also existed for a long time. Today there are a number of general-purpose sync engines, some of which use a proprietary backend service (e.g., Google Firestore, Realm, or Ditto), and some have an open source backend, making them suitable for creating local-first software (e.g., PouchDB/CouchDB, Automerge, or Yjs).

Multiplayer video games have a similar need to respond immediately to the user's local actions, and reconcile them with other players' actions received asynchronously over the network. In game development jargon the equivalent of a sync engine is called *netcode*. The techniques used in netcode are quite specific to the requirements of games [44], and don't directly carry over to other types of software, so we won't consider them further in this book.

Dealing with Conflicting Writes

The biggest problem with multi-leader replication—both in a geo-distributed server-side database and a local-first sync engine on end user devices—is that concurrent writes on different leaders can lead to conflicts that need to be resolved.

For example, consider a wiki page that is simultaneously being edited by two users, as shown in Figure 6-9. User 1 changes the title of the page from A to B, and user 2 independently changes the title from A to C. Each user's change is successfully applied to their local leader. However, when the changes are asynchronously replicated, a conflict is detected. This problem does not occur in a single-leader database.

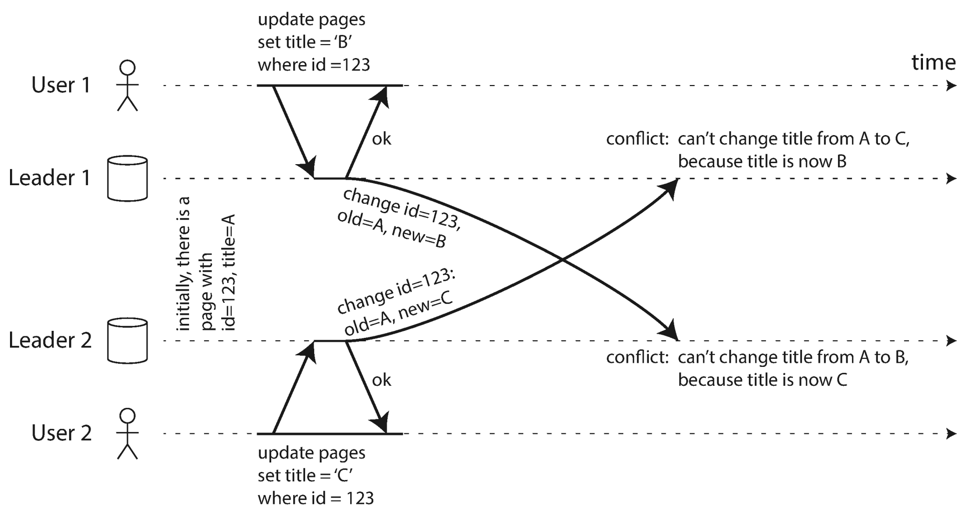


Figure 6-9. A write conflict caused by two leaders concurrently updating the same record.

NOTE

We say that the two writes in Figure 6-9 are *concurrent* because neither was "aware" of the other at the time the write was originally made. It doesn't matter whether the writes literally happened at the same time; indeed, if the writes were made while offline, they might have actually happened some time apart. What matters is whether one write occurred in a state where the other write has already taken effect.

In "Detecting Concurrent Writes" we will tackle the question of how a database can determine whether two writes are concurrent. For now we will assume that we can detect conflicts, and we want to figure out the best way of resolving them.

Conflict avoidance

One strategy for conflicts is to avoid them occurring in the first place. For example, if the application can ensure that all writes for a particular record go through the same leader, then conflicts cannot occur, even if the database as a whole is multi-leader. This approach is not possible in the case of a sync engine client being updated offline, but it is sometimes possible in geo-replicated server systems [30].

For example, in an application where a user can only edit their own data, you can ensure that requests from a particular user are always routed to the same region and use the leader in that region for reading and writing. Different users may have different "home" regions (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 region is unavailable and you need to reroute traffic to another region, or perhaps because a user has moved to a different location and is now closer to a different region. There is now a risk that the user performs a write while the change of designated leader is in progress, leading to a conflict that would have to be resolved using one of the methods below. Thus, conflict avoidance breaks down if you allow the leader to be changed.

Another example of conflict avoidance: imagine you want to insert new records and generate unique IDs for them based on an auto-incrementing counter. If you have two leaders, you could set them up so that one leader only generates odd numbers and the other only generates even numbers. That way you can be sure that the two leaders won't concurrently assign the same ID to different records. We will discuss other ID assignment schemes in "ID Generators and Logical Clocks".

Last write wins (discarding concurrent writes)

If conflicts can't be avoided, the simplest way of resolving them is to attach a timestamp to each write, and to always use the value with the greatest timestamp. For example, in Figure 6-9, let's say that the timestamp of user 1's write is greater than the timestamp of user 2's write. In that case, both leaders will determine that the new title of the page should be B, and they discard the write that sets it to C. If the writes coincidentally have the same timestamp, the winner can be chosen by comparing the values (e.g., in the case of strings, taking the one that's earlier in the alphabet).

This approach is called *last write wins* (LWW) because the write with the greatest timestamp can be considered the "last" one. The term is misleading though, because when two writes are concurrent like in Figure 6-9, which one is older and which is later is undefined, and so the timestamp order of concurrent writes is essentially random.

Therefore the real meaning of LWW is: when the same record is concurrently written on different leaders, one of those writes is randomly chosen to be the winner, and the other writes are silently discarded, even though they were successfully processed at their respective leaders. This achieves the goal that eventually all replicas end up in a consistent state, but at the cost of data loss.

If you can avoid conflicts—for example, by only inserting records with a unique key such as a UUID, and never updating them—then LWW is no problem. But if you update existing records, or if different leaders may insert records with the same key, then you have to decide whether lost updates are a problem for your application. If lost updates are not acceptable, you need to use one of the conflict resolution approaches described below.

Another problem with LWW is that if a real-time clock (e.g. a Unix timestamp) is used as timestamp for the writes, the system becomes very sensitive to clock synchronization. If one node has a clock that is ahead of the others, and you try to overwrite a value written by that node, your write may be ignored as it may have a lower timestamp, even though it clearly occurred later. This problem can be solved by using a *logical clock*, which we will discuss in "ID Generators and Logical Clocks".

Manual conflict resolution

If randomly discarding some of your writes is not desirable, the next option is to resolve the conflict manually. You may be familiar with manual conflict resolution from Git and other version control systems: if commits on two different branches edit the same lines of the same file, and you try to merge those branches, you will get a merge conflict that needs to be resolved before the merge is complete.

In a database, it would be impractical for a conflict to stop the entire replication process until a human has resolved it. Instead, databases typically store all the concurrently written values for a given record—for example, both B and C in Figure 6-9. These values are sometimes called *siblings*. The next time you query that record, the database returns *all* those values, rather than just the latest one. You can then resolve those values in whatever way you want, either automatically in application code (for example, you could concatenate B and C into "B/C"), or by asking the user. You then write back a new value to the database to resolve the conflict.

This approach to conflict resolution is used in some systems, such as CouchDB. However, it also suffers from a number of problems:

- The API of the database changes: for example, where previously the title of the wiki page was just a string, it now becomes a set of strings that usually contains one element, but may sometimes contain multiple elements if there is a conflict. This can make the data awkward to work with in application code.
- Asking the user to manually merge the siblings is a lot of work, both for the app developer (who needs to build the user interface for conflict resolution) and for the user (who may be confused about what they are being asked to do, and why). In many cases, it's better to merge automatically than to bother the user.
- Merging siblings automatically can lead to surprising behavior if it is not done carefully. For example, the shopping cart on Amazon used to allow concurrent updates, which were then merged by keeping all the shopping cart items that appeared in any of the siblings (i.e., taking the set union of

the carts). This meant that if the customer had removed an item from their cart in one sibling, but another sibling still contained that old item, the removed item would unexpectedly reappear in the customer's cart [45]. Figure 6-10 shows an example where Device 1 removes Book from the shopping cart and concurrently Device 2 removes DVD, but after merging the conflict both items reappear.

- If multiple nodes observe the conflict and concurrently resolve it, the conflict resolution process can itself introduce a new conflict. Those resolutions could even be inconsistent: for example, one node may merge B and C into "B/C" and another may merge them into "C/B" if you are not careful to order them consistently. When the conflict between "B/C" and "C/B" is merged, it may result in "B/C/C/B" or something similarly surprising.

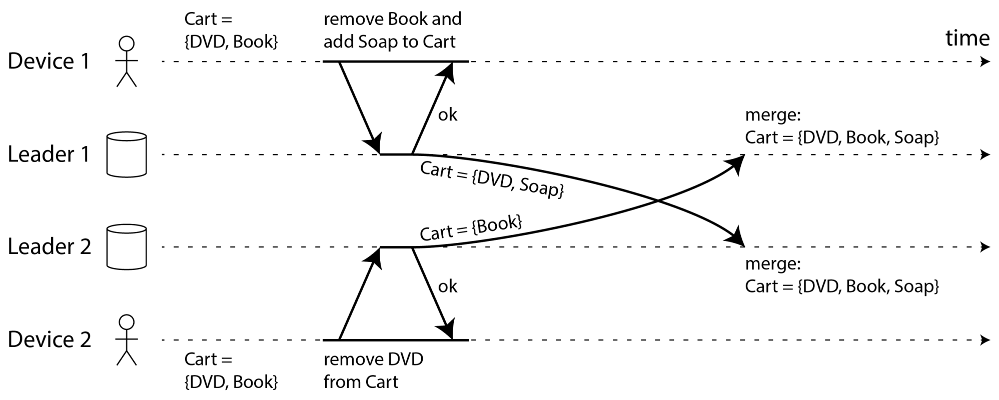


Figure 6-10. Example of Amazon's shopping cart anomaly: if conflicts on a shopping cart are merged by taking the union, deleted items may reappear.

Automatic conflict resolution

For many applications, the best way of handling conflicts is to use an algorithm that automatically merges concurrent writes into a consistent state. Automatic conflict resolution ensures that all replicas *converge* to the same state—i.e., all replicas that have processed the same set of writes have the same state, regardless of the order in which the writes arrived. Combining eventual consistency with a convergence guarantee is known as *strong eventual consistency* [46].

LWW is a simple example of a conflict resolution algorithm. More sophisticated merge algorithms have been developed for different types of data, with the goal of preserving the intended effect of all updates as much as possible, and hence avoiding data loss:

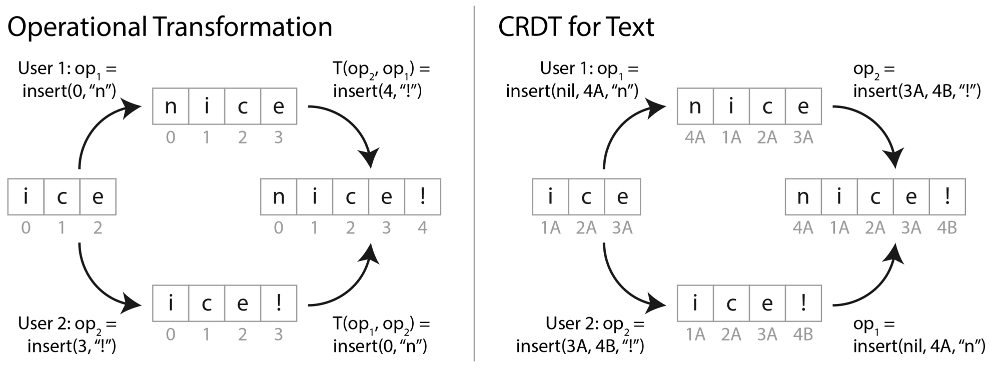
- If the data is text (e.g., the title or body of a wiki page), we can detect which characters have been inserted or deleted from one version to the next. The merged result then preserves all the insertions and deletions made in any of the siblings. If users concurrently insert text at the same position, it can be ordered deterministically so that all nodes get the same merged outcome.
- If the data is a collection of items (ordered like a to-do list, or unordered like a shopping cart), we can merge it similarly to text by tracking insertions and deletions. To avoid the shopping cart issue in Figure 6-10, the algorithms track the fact that Book and DVD were deleted, so the merged result is Cart = {Soap}.
- If the data is an integer representing a counter that can be incremented or decremented (e.g., the number of likes on a social media post), the merge algorithm can tell how many increments and decrements happened on each sibling, and add them together correctly so that the result does not double-count and does not drop updates.
- If the data is a key-value mapping, we can merge updates to the same key by applying one of the other conflict resolution algorithms to the values under that key. Updates to different keys can be handled independently from each other.

There are limits to what is possible with conflict resolution. For example, if you want to enforce that a list contains no more than five items, and multiple users concurrently add items to the list so that there are more than five in total, your only option is to drop some of the items. Nevertheless, automatic conflict resolution is sufficient to build many useful apps. And if you start from the requirement of wanting to build a collaborative offline-first or local-first app, then conflict resolution is inevitable, and automating it is often the best approach.

CRDTs and Operational Transformation

Two families of algorithms are commonly used to implement automatic conflict resolution: *Conflict-free replicated datatypes* (CRDTs) [46] and *Operational Transformation* (OT) [47]. They have different design philosophies and performance characteristics, but both are able to perform automatic merges for all the aforementioned types of data.

Figure 6-11 shows an example of how OT and a CRDT merge concurrent updates to a text. Assume you have two replicas that both start off with the text “ice”. One replica prepends the letter “n” to make “nice”, while concurrently the other replica appends an exclamation mark to make “ice!”.



Operational Transformation
User 1: op_1 = insert(0, “n”)
n i c e
0 1 2 3
$T(op_2, op_1)$ = insert(4, “!”)
i c e
0 1 2
n i c e !
0 1 2 3 4
User 2: op_2 = insert(3, “!”)
i c e !
0 1 2 3
$T(op_1, op_2)$ = insert(0, “n”)
CRDT for Text
User 1: op_1 = insert(nil, 4A, “n”)
n i c e
4A 1A 2A 3A
op_2 = insert(3A, 4B, “!”)
i c e
1A 2A 3A
n i c e !
4A 1A 2A 3A 4B
User 2: op_2 = insert(3A, 4B, “!”)
i c e !
1A 2A 3A 4B
op_1 = insert(nil, 4A, “n”)

Figure 6-11. How two concurrent insertions into a string are merged by OT and a CRDT respectively.

The merged result “nice!” is achieved differently by both types of algorithms:

OT

We record the index at which characters are inserted or deleted: “n” is inserted at index 0, and “!” at index 3. Next, the replicas exchange their operations. The insertion of “n” at 0 can be applied as-is, but if the insertion of “!” at 3 were applied to the state “nice” we would get “nic!e”, which is incorrect. We therefore need to transform the index of each operation to account for concurrent operations that have already been applied; in this case, the insertion of “!” is transformed to index 4 to account for the insertion of “n” at an earlier index.

CRDT

Most CRDTs give each character a unique, immutable ID and use those to determine the positions of insertions/deletions, instead of indexes. For example, in Figure 6-11 we assign the ID 1A to “i”, the ID 2A to “c”, etc. When inserting the exclamation mark, we generate an operation containing the ID of the new character (4B) and the ID of the existing character after which we want to insert (3A). To insert at the beginning of the string we give “nil” as the preceding character ID. Concurrent insertions at the same position are ordered by the IDs of the characters. This ensures that replicas converge without performing any transformation.

There are many algorithms based on variations of these ideas. Lists/arrays can be supported similarly, using list elements instead of characters, and other datatypes such as key-value maps can be added quite easily. There are some performance and functionality trade-offs between OT and CRDTs, but it’s possible to combine the advantages of CRDTs and OT in one algorithm [48].

OT is most often used for real-time collaborative editing of text, e.g. in Google Docs [32], whereas CRDTs can be found in distributed databases such as Redis Enterprise, Riak, and Azure Cosmos DB [49]. Sync engines for JSON data can be implemented both with CRDTs (e.g., Automerge or Yjs) and with OT (e.g., ShareDB).

Types of conflict

Some kinds of conflict are obvious. In the example in Figure 6-9, 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. Rather than updating a specific field when booking a meeting, this system inserts a new record into the database for each booking. The 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 close enough that they both see the room unbooked prior to inserting their new record.

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 8, and in Chapter 13 we will discuss scalable approaches for detecting and resolving conflicts in a replicated system.

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, 50], 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 in 2007 [45]. Riak, Cassandra, and ScyllaDB are open source datastores with leaderless replication models inspired by Dynamo, so this kind of database is also known as *Dynamo-style*.

NOTE

The original *Dynamo* system was only described in a paper [45], but never released outside of Amazon. The similarly-named *DynamoDB* is a more recent cloud database from AWS, but it has a completely different architecture: it uses single-leader replication based on the Multi-Paxos consensus algorithm [5, 51].

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 single-leader configuration, if you want to continue processing writes, you may need to perform a failover (see "Handling Node Outages").

On the other hand, in a leaderless configuration, failover does not exist. Figure 6-12 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.

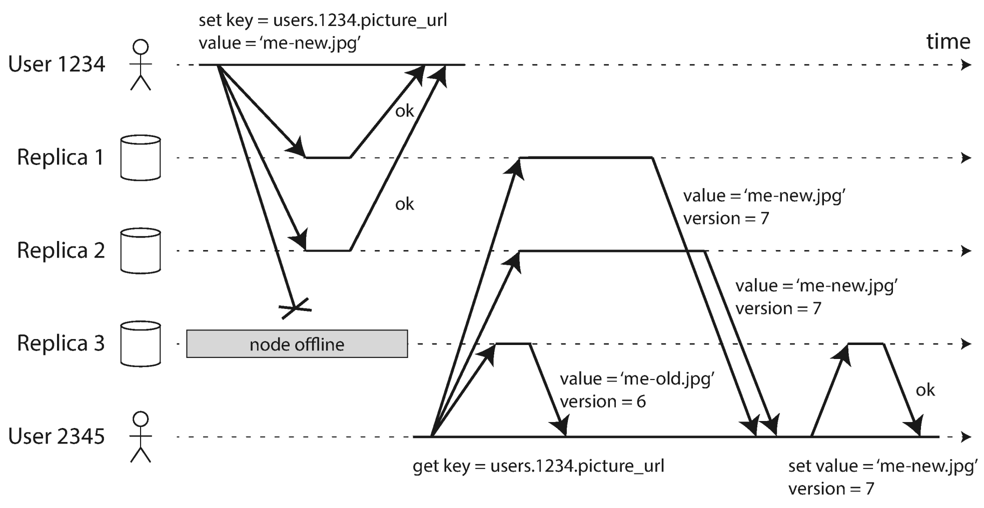


Figure 6-12. A quorum write, quorum read, and read repair after a node outage.

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; for example, the up-to-date value from one node and a stale value from another.

In order to tell which responses are up-to-date and which are outdated, every value that is written needs to be tagged with a version number or timestamp, similarly to what we saw in "Last write wins (discarding concurrent writes)". When a client receives multiple values in response to a read, it uses the one with the greatest timestamp (even if that value was only returned by one replica, and several other replicas returned older values). See "Detecting Concurrent Writes" for more details.

Catching up on missed writes

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? Several mechanisms are 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 6-12, 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.

Hinted handoff

> If one replica is unavailable, another replica may store writes on its behalf in the form of *hints*. When the replica that was supposed to receive those writes comes back, the replica storing the hints sends them to the recovered replica, and then deletes the hints. This *handoff* process helps bring replicas up-to-date even for values that are never read, and therefore not handled by read repair.

Anti-entropy

> In addition, there is a background process that periodically 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.

Quorums for reading and writing

In the example of Figure 6-12, 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 [50]. You can think of r and w as the minimum number of votes required for the read or write to be valid.

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.

NOTE

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 sharded, supporting datasets that are larger than you can fit on one node. We will return to sharding in Chapter 7.

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, like in Figure 6-12.
- With $n = 5$, $w = 3$, $r = 3$ we can tolerate two unavailable nodes. This case is illustrated in Figure 6-13.

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.

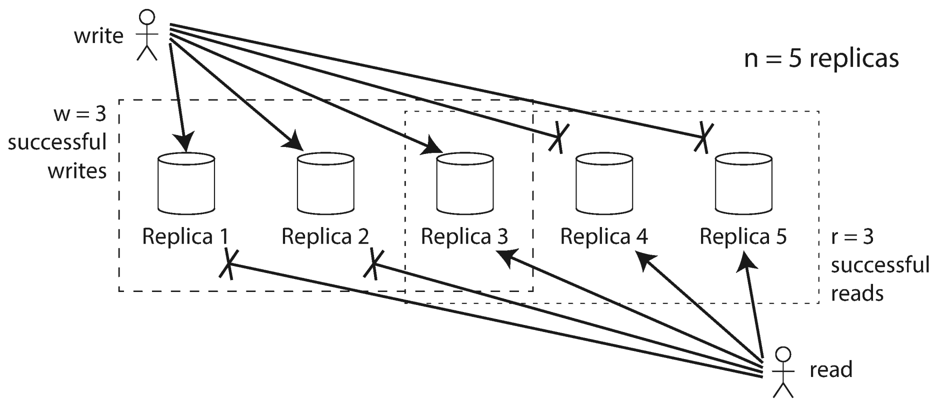


Figure 6-13. If $w + r > n$, at least one of the r replicas you read from must have seen the most recent successful write.

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 faults.

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 6-13).

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 [52].

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, which is particularly beneficial with *synchronous* (*blocking*) replication. This setup is also more highly available: 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 edge cases in which the consistency properties can be confusing. Some scenarios include:

- 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.
- While a rebalancing is in progress, where some data is moved from one node to another (see Chapter 7), nodes may have inconsistent views of which nodes should be holding the *n* replicas for a particular value. This can result in the read and write quorums no longer overlapping.
- If a read is concurrent with a write operation, the read may or may not see the concurrently written value. In particular, it's possible for one read to see the new value, and a subsequent read to see the old value, as we shall see in "Linearizability and quorums".
- 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 [53].
- If the database uses timestamps from a real-time clock to determine which write is newer (as Cassandra and ScyllaDB do, for example), writes might be silently dropped if another node with a faster clock has written to the same key—an issue we previously saw in "Last write wins (discarding concurrent writes)". We will discuss this in more detail in "Relying on Synchronized Clocks".
- If two writes occur concurrently, one of them might be processed first on one replica, and the other might be processed first on another replica. This leads to a conflict, similarly to what we saw for multi-leader replication (see "Dealing with Conflicting Writes"). We will return to this topic in "Detecting Concurrent Writes".

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 [54], but it's wise to not take them as absolute guarantees.

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. The number of hints that a replica stores for handoff can be one measure of system health, but it's difficult to interpret usefully [55]. Eventual consistency is a deliberately vague guarantee, but for operability it's important to be able to quantify "eventual."

Single-Leader Versus Leaderless Replication Performance

A replication system based on a single leader can provide strong consistency guarantees that are difficult or impossible to achieve in a leaderless system. However, as we have seen in "Problems with Replication Lag", reads in a leader-based replicated system can also return stale values if you make them on an asynchronously updated follower.

Reading from the leader ensures up-to-date responses, but it suffers from performance problems:

- Read throughput is limited by the leader's capacity to handle requests (in contrast with read scaling, which distributes reads across asynchronously updated replicas that may return stale values).
- If the leader fails, you have to wait for the fault to be detected, and for the failover to complete before you can continue handling requests. Even if the failover process is very quick, users will notice it because of the temporarily increased response times; if failover takes a long time, the system is unavailable for its duration.
- The system is very sensitive to performance problems on the leader: if the leader is slow to respond, e.g. due to overload or some resource contention, the increased response times immediately affect users as well.

A big advantage of a leaderless architecture is that it is more resilient against such issues. Because there is no failover, and requests go to multiple replicas in parallel anyway, one replica becoming slow or unavailable has very little impact on response times: the client simply uses the responses from the other replicas that are faster to respond. Using the fastest responses is called *request hedging*, and it can significantly reduce tail latency [56]).

At its core, the resilience of a leaderless system comes from the fact that it doesn't distinguish between the normal case and the failure case. This is especially helpful when handling so-called *gray failures*, in which a node isn't completely down, but running in a degraded state where it is unusually slow to handle requests [57], or when a node is simply overloaded (for example, if a node has been offline for a while, recovery via hinted handoff can cause a lot of additional load). A leader-based system has to decide whether the situation is bad enough to warrant a failover (which can itself cause further disruption), whereas in a leaderless system that question doesn't even arise.

That said, leaderless systems can have performance problems as well:

- Even though the system doesn't need to perform failover, one replica does need to detect when another replica is unavailable so that it can store hints about writes that the unavailable replica missed. When the unavailable replica comes back, the handoff process needs to send it those hints. This puts additional load on the replicas at a time when the system is already under strain [55].
- The more replicas you have, the bigger the size of your quorums, and the more responses you have to wait for before a request can complete. Even if you wait only for the fastest *r* or *w* replicas to respond, and even if you

make the requests in parallel, a bigger *r* or *w* increases the chance that you hit a slow replica, increasing the overall response time (see "Use of Response Time Metrics"). In practice, quorums are seldom more than 4 out of 7 nodes or 5 out of 9 nodes.

- A large-scale network interruption that disconnects a client from a large number of replicas can make it impossible to form a quorum. Some leaderless databases offer a configuration option that allows any reachable replica to accept writes, even if it's not one of the usual replicas for that key (Riak and Dynamo call this a *sloppy quorum* [45]; Cassandra and ScyllaDB call it *consistency level ANY*). There is no guarantee that subsequent reads will see the written value, but depending on the application it may still be better than having the write fail.

Multi-leader replication can offer even greater resilience against network interruptions than leaderless replication, since reads and writes only require communication with one leader, which can be co-located with the client. However, since a write on one leader is propagated asynchronously to the others, reads can be arbitrarily out-of-date. Quorum reads and writes provide a compromise: good fault tolerance while also having a high likelihood of reading up-to-date data.

Multi-region operation

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

In Cassandra and ScyllaDB, a client that wants to perform a multi-region write first chooses a node in its local region, called the *coordinator node*, and sends its write to that node. The coordinator node forwards the write to all replicas in its own region, and also to one replica in every other region. In the other region, the first replica to receive the write forwards it to the other replicas in the same region. This optimization avoids making the cross-region request multiple times.

You can choose from a variety of consistency levels that determine how many responses are required for a request to be successful. For example, you can request a quorum across the replicas in all the regions, a separate quorum in each of the regions, or a quorum only in the client's local region. A local quorum avoids having to wait for slow requests to other regions, but it is also more likely to return stale results.

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

Detecting Concurrent Writes

Like with multi-leader replication, leaderless databases allow concurrent writes to the same key, resulting in conflicts that need to be resolved. Such conflicts may occur as the writes happen, but not always: they could also be detected later during read repair, hinted handoff, or anti-entropy.

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 6-14 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.

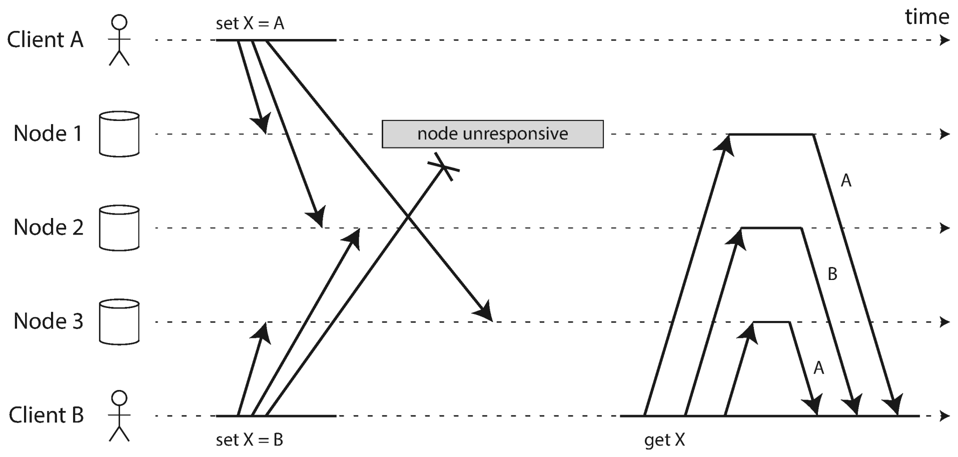


Figure 6-14. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

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 6-14: 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. For this, we can use any of the conflict resolution mechanisms we previously discussed in "Dealing with Conflicting Writes", such as last-write-wins (used by Cassandra and ScyllaDB), manual resolution, or CRDTs (described in "CRDTs and Operational Transformation", and used by Riak).

Last-write-wins is easy to implement: each write is tagged with a timestamp, and a value with a higher timestamp always overwrites a value with a lower timestamp. However, a timestamp doesn't tell you whether two values are actually conflicting (i.e., they were written concurrently) or not (they were written one after another). If you want to resolve conflicts explicitly, the system needs to take more care to detect concurrent writes.

The "happens-before" relation 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 6-8, 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 6-14 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) [58].

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 9.

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 [58], 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 database 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. 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 siblings, i.e., 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, e.g. using a CRDT or by asking the user. The response from a write request is like a read, returning all siblings, 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).

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).

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.

Figure 6-15 shows this algorithm in action.

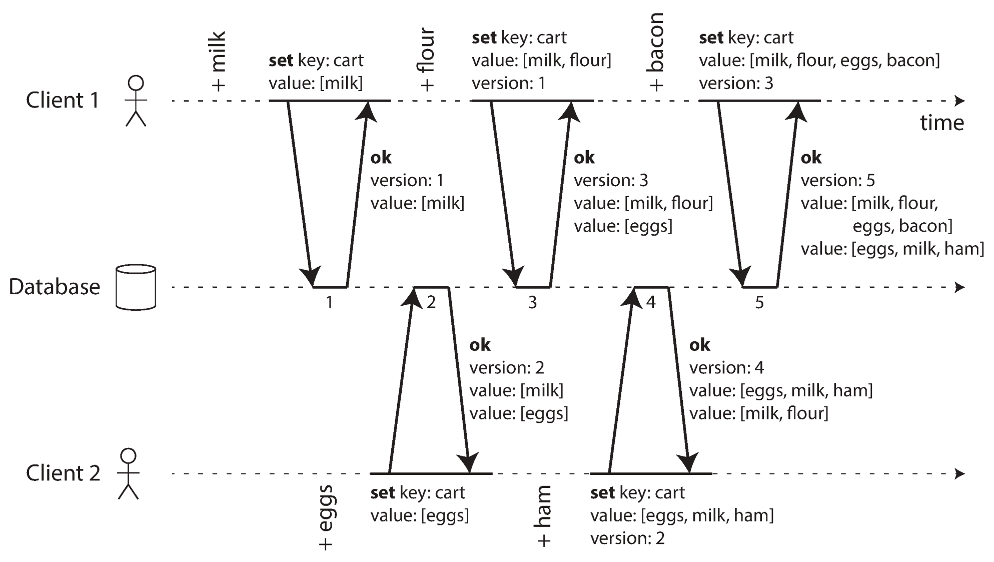


Figure 6-15. Capturing causal dependencies between two clients concurrently editing a shopping cart.

In this example, two clients are 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 (siblings). 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.

The dataflow between the operations in Figure 6-15 is illustrated graphically in Figure 6-16. 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.

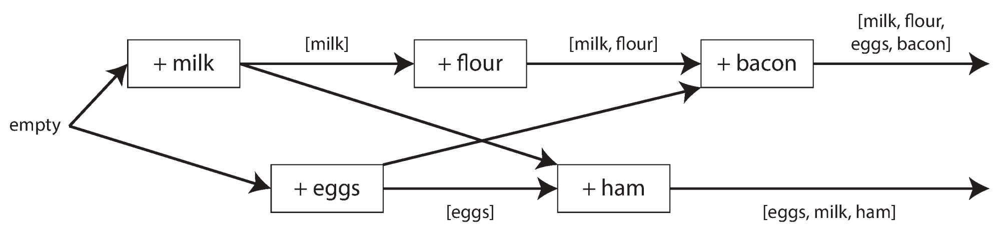


Figure 6-16. Graph of causal dependencies in Figure 6-15.

Version vectors

The example in Figure 6-15 used only a single replica. How does the algorithm change when there are multiple replicas, but no leader?

Figure 6-15 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* [59]. A few variants of this idea are in use, but the most interesting is probably the *dotted version vector* [60, 61], which is used in Riak 2.0 [62, 63]. 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 6-15, 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.

The version vector also 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 [61, 64, 65]. 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, a zone, or even an entire region) goes down

Durability

Ensuring you don't lose data, even if a whole machine (or even an entire region) fails permanently

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, all the things that can go wrong, and how to deal 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 or hardware errors).

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 it offers strong consistency. Multi-leader and leaderless replication can be more robust in the presence of faulty nodes, network interruptions, and latency spikes—at the cost of requiring conflict resolution and providing weaker 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 how multi-leader and leaderless replication ensure that all replicas eventually converge to a consistent state: by using a version vector or similar algorithm to detect which writes are concurrent, and by using a conflict resolution algorithm such as a CRDT to merge the concurrently written values. Last-write-wins and manual conflict resolution are also possible.

This chapter has assumed that every replica stores a full copy of the whole database, which is unrealistic for large datasets. In the next chapter we will look at *sharding*, which allows each machine to store only a subset of the data.

FOOTNOTES

REFERENCES

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

[2] Kenny Gryp. MySQL Terminology Updates. *dev.mysql.com*, July 2020. Archived at perma.cc/S62G-6RJ2

[3] Oracle Corporation. Oracle (Active) Data Guard 19c: Real-Time Data Protection and Availability. White Paper, *oracle.com*, March 2019. Archived at perma.cc/P5ST-RPKE

[4] Microsoft. What is an Always On availability group? *learn.microsoft.com*, September 2024. Archived at perma.cc/ABH6-3MXF

[5] Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully

Managed NoSQL Database Service. At *USENIX Annual Technical Conference* (ATC), July 2022.

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

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

[8] Mallory Knodel and Niels ten Oever. Terminology, Power, and Inclusive Language in Internet-Drafts and RFCs. *IETF Internet-Draft*, August 2023. Archived at perma.cc/5ZY9-725E

[9] Buck Hodges. Postmortem: VSTS 4 September 2018. *devblogs.microsoft.com*, September 2018. Archived at perma.cc/ZF5R-DYZS

[10] Gunnar Morling. Leader Election With S3 Conditional Writes. *www.morling.dev*, August 2024. Archived at perma.cc/7V2N-J78Y

[11] Vignesh Chandramohan, Rohan Desai, and Chris Riccomini. SlateDB Manifest Design. *github.com*, May 2024. Archived at perma.cc/8EUY-P32Z

[12] Stas Kelvich. Why does Neon use Paxos instead of Raft, and what's the difference? *neon.tech*, August 2022. Archived at perma.cc/SEZ4-2GXU

[13] Dimitri Fontaine. An introduction to the pg_auto_failover project. *tapoueh.org*, November 2021. Archived at perma.cc/3WH5-6BAF

[14] Jesse Newland. GitHub availability this week. *github.blog*, September 2012. Archived at perma.cc/3YRF-FTFJ

[15] Mark Imbriaco. Downtime last Saturday. *github.blog*, December 2012. Archived at perma.cc/M7X5-E8SQ

[16] John Hugg. 'All In' with Determinism for Performance and Testing in Distributed Systems. At *Strange Loop*, September 2015.

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

[18] Amit Kapila. WAL Internals of PostgreSQL. At *PostgreSQL Conference* (PGCon), May 2012. Archived at perma.cc/6225-3SUX

[19] Amit Kapila. Evolution of Logical Replication. *amitkapila16.blogspot.com*, September 2023. Archived at perma.cc/F9VX-JLER

[20] Aru Petchimuthu. Upgrade your Amazon RDS for PostgreSQL or Amazon Aurora PostgreSQL database, Part 2: Using the pglogical extension. *aws.amazon.com*, August 2021. Archived at perma.cc/RXT8-FS2T

[21] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services. At *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

[22] Douglas B. Terry. Replicated Data Consistency Explained Through Baseball. Microsoft Research, Technical Report MSR-TR-2011-137, October 2011. Archived at perma.cc/F4KZ-AR38

[23] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theher, and Brent B. Welch. 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

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

[25] Simon Willison. Reply to: "My thoughts about Fly.io (so far) and other newish technology I'm getting into". *news.ycombinator.com*, May 2022. Archived at perma.cc/ZRV4-WWV8

[26] Nithin Tharakan. Scaling Bitbucket's Database. *atlassian.com*, October 2020. Archived at perma.cc/JAB7-9FGX

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

[28] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proceedings of the VLDB*

Endowment, volume 8, issue 3, pages 185–196, November 2014. doi:10.14778/2735508.2735509

[29] Yaser Raja and Peter Celentano. PostgreSQL bi-directional replication using pglogical. *aws.amazon.com*, January 2022. Archived at https://perma.cc/BUQ2-5QWN

[30] Robert Hodges. If You *Must* Deploy Multi-Master Replication, Read This First. *scale-out-blog.blogspot.com*, April 2012. Archived at perma.cc/C2JN-F6Y8

[31] Lars Hofhansl. HBASE-7709: Infinite Loop Possible in Master/Master Replication. *issues.apache.org*, January 2013. Archived at perma.cc/24G2-8NLC

[32] John Day-Richter. What's Different About the New Google Docs: Making Collaboration Fast. *drive.googleblog.com*, September 2010. Archived at perma.cc/5TL8-TSJ2

[33] Evan Wallace. How Figma's multiplayer technology works. *figma.com*, October 2019. Archived at perma.cc/L49H-LY4D

[34] Tuomas Artman. Scaling the Linear Sync Engine. *linear.app*, June 2023.

[35] Amr Saafan. Why Sync Engines Might Be the Future of Web Applications. *nilebits.com*, September 2024. Archived at perma.cc/5N73-5M3V

[36] Isaac Hagoel. Are Sync Engines The Future of Web Applications? *dev.to*, July 2024. Archived at perma.cc/R9HF-BKKL

[37] Sujay Jayakar. A Map of Sync. *stack.convex.dev*, October 2024. Archived at perma.cc/82R3-H42A

[38] Alex Feyerke. Designing Offline-First Web Apps. *alistapart.com*, December 2013. Archived at perma.cc/WH7R-S2DS

[39] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. At *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward!), October 2019, pages 154–178. doi:10.1145/3359591.3359737

[40] Martin Kleppmann. The past, present, and future of local-first. At *Local-First Conference*, May 2024.

[41] Conrad Hofmeyr. API Calling is to Sync Engines as jQuery is to React. *powersync.com*, November 2024. Archived at perma.cc/2FP9-7WJJ

[42] Peter van Hardenberg and Martin Kleppmann. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. At *7th Workshop on Principles and Practice of Consistency for Distributed Data* (PaPoC), April 2020. doi:10.1145/3380787.3393683

[43] Leonard Kawell, Jr., Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. At *ACM Conference on Computer-Supported Cooperative Work* (CSCW), September 1988. doi:10.1145/62266.1024798

[44] Ricky Pusch. Explaining how fighting games use delay-based and rollback netcode. *words.infil.net* and *arstechnica.com*, October 2019. Archived at perma.cc/DE7W-RDJ8

[45] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. At *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007. doi:10.1145/1323293.1294281

[46] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. At *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems* (SSS), pages 386–400, October 2011. doi:10.1007/978-3-642-24550-3_29

[47] 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. doi:10.1145/289444.289469

[48] Joseph Gentle and Martin Kleppmann. Collaborative Text Editing with Eg-walker: Better, Faster, Smaller. At *20th European Conference on Computer Systems* (EuroSys), March 2025. doi:10.1145/3689031.3696076

[49] Dharma Shukla. Azure Cosmos DB: Pushing the frontier of globally distributed databases. *azure.microsoft.com*, September 2018. Archived at perma.cc/UT3B-HH6R

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

[51] Marc Brooker. Dynamo, DynamoDB, and Aurora DSQL. *brooker.co.za*, August 2025. Archived at perma.cc/XG3C-ALDQ

[52] 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

[53] Joseph Blomstedt. Bringing Consistency to Riak. At *RICON West*, October 2012.

[54] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, volume 23, pages 279–302, April 2014. doi:10.1007/s00778-013-0330-1

[55] Colin Breck. Shared-Nothing Architectures for Server Replication and Synchronization. *blog.colinbreck.com*, December 2019. Archived at perma.cc/48P3-J6CJ

[56] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, volume 56, issue 2, pages 74–80, February 2013. doi:10.1145/2408776.2408794

[57] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. At *16th Workshop on Hot Topics in Operating Systems* (HotOS), May 2017. doi:10.1145/3102980.3103005

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

[59] D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, volume SE-9, issue 3, pages 240–247, May 1983. doi:10.1109/TSE.1983.236733

[60] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted Version Vectors: Logical Clocks for Optimistic Replication. arXiv:1011.5808, November 2010.

[61] Giridhar Manepalli. Clocks and Causality - Ordering Events in Distributed Systems. *exhypothesi.com*, November 2022. Archived at perma.cc/8REU-KVLQ

[62] Sean Cribbs. A Brief History of Time in Riak. At *RICON*, October 2014. Archived at perma.cc/7U9P-6JFX

[63] Russell Brown. Vector Clocks Revisited Part 2: Dotted Version Vectors. *riak.com*, November 2015. Archived at perma.cc/96QP-W98R

[64] Carlos Baquero. Version Vectors Are Not Vector Clocks. *haslab.wordpress.com*, July 2011. Archived at perma.cc/7PNU-4AMG

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