Modern applications can handle thousands of transactions per minute. One of the things that you will definitely do to achieve scalability, is to do replication of your [database](https://hackernoon.com/tagged/database). There are several reasons for doing so:

1. You want to keep data geographically close to users
2. Balance read/write across multiple machines to prevent throttling of a single database server
3. Build redundancies within your system

There are a lot of interesting subtleties involved in replications that are concerned with durability guarantees and eventual consistencies. As a product person or application developer, it’s very important to cut through the vagueness of jargons and understand how replication actually works.

A copy of your data on another machine or node is known as a replica. There are three ways to achieve replication of data.

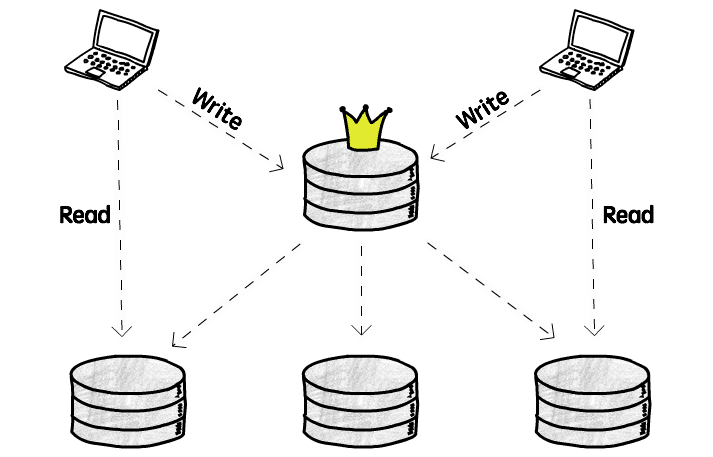
1. Single master slave replication : All writes go to one node, called master. The changes in master node are replicated to other nodes, called slaves. Read requests can go to either master or slave.
2. Many master replication: Instead of writes going to one master, they can go to many masters. Masters can then replicate changes made to their data stores to other slaves nodes. Read requests can go to either master or slave.
3. No master replication: There are no master and slaves in this setup. Writes and reads can be sent to any node.

I will cover single master slave replication in some detail before talking about the other configurations.

#### **Single leader replication**

The most common replication topology is to have a single leader, that then replicate the changes to all the followers.

In this setup, the clients always send writes (in the case of databases, INSERT, UPDATE and DELETE queries) to the leader, and never to a follower. These followers can, however, answer read queries.



The main benefit of having a single leader is that we avoid conflicts caused by concurrent writes. All the clients are writing to the same server, so the coordination is easier. If we instead allow clients to write to 2 different servers at the same time, we need to somehow resolve the conflict that will happen if they both try to change the same *object*, with different values (more on that later).

So, what are the problems that we need to keep in mind if we decide to go with the single leader approach? The first one is that we need to make sure that just one node is able to handle all the writes. Although we can split the read work across the entire cluster, all the writes are going to a single server, and if your application is very write-intensive that might be a problem. Keep in mind though, that most applications read a lot more data than they write, so you need to analyze if that’s really a problem for you.

Another problem is that you will need to pay the latency price on writes. Remember our colleagues in Asia? Well, when they want to update some data, that query will still need to travel the globe before they get a response.

Lastly, although this is not really a problem just for single leader replication, you need to think about what will happen when the leader node dies. Is the entire system going to stop working? Will it be available just for reads (from the replicas), but not for writes? Is there a process to *elect* a new leader (i.e. promoting one of the replicas to a leader status)? Is this election process automated or will it need someone to tell the system who is the new king in town?

At first glance it seems like the best approach is to just have an automatic failover strategy, that will elect a new leader and everything will keep working wonderfully. That, unfortunately, is easier said than done.

##### **The challenges of an automatic failover**

Let me list *some* of the challenges in implementing this automatic failover strategy.

The first question we need to ask is: How can we be sure that the leader is dead? And the answer is: We probably can’t.

There are a billion things that can go wrong, and, like in any distributed system, it is impossible to distinguish a slow-to-answer from a dead node. Databases usually use a timeout to decide that (e.g. if I don’t hear from you in 20 seconds you are dead to me!). That is usually good enough, but certainly not perfect. If you wait more, it is less likely that you will identify a node as dead by mistake, but it will also take more time start your failover process, and in the meantime your system is probably unusable. On the other hand, if you don’t give it enough time you might start a failover process that was not necessary. So that is challenge number one.

Challenge number two: You need to decide who is the new leader. You have all these followers, living in an anarchy, and they need to somehow agree on who should be the new leader. For example, one relatively simple (at least conceptually) approach it to have a predefined successor node, that will assume the leader position when the original leader dies. Or you can choose the node that has the most recent update (e.g. the one that is closer to the leader), to minimize data loss. Any way you decide to choose the new leader, all the nodes still need to *agree* on that decision, and that’s the hard part. This is known as a [consensus problem](https://en.wikipedia.org/wiki/Consensus_(computer_science)), and can be quite tricky to get right.

Alright, you detected that the leader is really dead and selected a new leader, now you need to somehow tell the clients to start sending writes to this new leader, instead of the dead one. This is a *request routing* problem, and we can also approach it from several different angles. For example, you can allow clients to send writes to any node, and have these nodes redirect this request to the leader. Or you can have a *routing layer* that receives this messages and redirect them to the appropriate node.

If you are using asynchronous replication, the new leader might not have all the data from the previous leader. In that case, if the old leader resurrects (maybe it was just a network glitch or a server restart) and the new leader received conflicting updates in the meantime, how do we handle these conflicts?

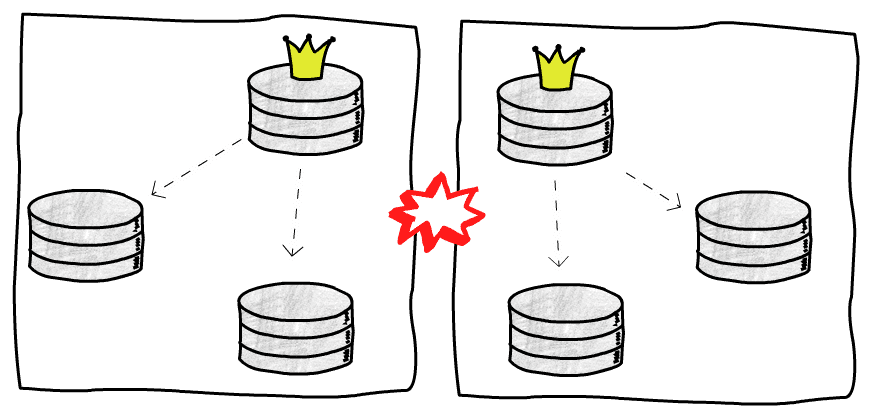
One common approach is to just discard these conflicts (using a last-write-win approach), but that can also be dangerous (take this [Github issue](https://github.com/blog/1261-github-availability-this-week) (from 2012) as an example).

We can also have a funny (well, maybe it’s not that funny when it happens in production) situation where the previous leader comes back up and thinks it is still the leader. That is called a *split brain*, and can lead to a weird situation.

If both leaders starts accepting writes and we are not ready to handle conflicts it is possible to lose data.

Some systems have fencing mechanisms that will force one node to shut down if it detects that there are multiple leaders. This approach is known by the great name STONITH, Shoot The Other Node In The Head.

*This is also what happens when there’s a network partition and we end up with what appears to be two isolated clusters, each one with its own leader, as each part of this cluster cannot see the other, and therefore thinks they are all dead.*

**

As you can see, automatic failovers are not simple. There are a lot of things to take into consideration, and for that reason sometimes it’s better to have a human manually perform this procedure. Of course, if your leader database dies at 7pm and there’s no one on-call, it might not be the best solution to wait until tomorrow morning, so, as always, trade-offs.

An important detail worth mentioning is the manner in which replication happens. It can either be synchronous or asynchronous.

Synchronous replication means that master sends its writes to all slaves and waits for write confirmation from ‘all’ slaves. Once the master receives write confirmations, it then returns a successful write message to client. After all this, the writes are made visible to other clients. This mode provides the highest durability guarantee. Every write will be propagated to all nodes and no client connecting to any of the slaves will ever see stale data. If these guarantees make sense for your application, you should choose synchronous replication.

The biggest drawback with synchronous replication is that it is very slow. If the network connectivity to some nodes is choppy, it can take forever to make one successful write. In practice, people always invariably choose ‘asynchronous’ replication. In asynchronous replication, a master confirms write to the client, after a successful write on master node. It also sends changes to other slave replicas but does not wait for a successful write confirmation from other slaves. This effectively means that durability guarantees are weakened as some of the slave nodes update their data stores later and in the meantime a client connecting to one of the slave nodes will be presented with old data. However, using asynchronous replication makes up for this by providing significant performance gains. As a result asynchronous replication is the dominant replication strategy.

There’s some middle ground. Some databases and replication tools allow us to define a number of followers to replicate synchronously, and the others just use the asynchronous approach. This is sometimes called semi-synchronous replication.

As an example, in Postgres you can define a configuration called synchronous\_standby\_names to specify which replicas will receive the updates synchronously, and the other replicas will just receive them asynchronously.

Most applications have read/write ratio skewed heavily in favour of reads i.e. writes will be fewer. In order to take advantage of this fact, reads are handled by slaves. In an asynchronous master slave configuration, this presents a problem because of replication lags. Let me go through a few examples of problems due to replication lags and steps to save them.

1. Reading writes: User submits a write on master and then decides to read his/her write. The read request can be serviced by any slave and the slave servicing read request might not have the most recent write on account of replication lag. This can clearly cause user frustration, not able to see his/her most recent post. You can resolve this by ensuring all reads go to master after 1 minutes of any write. Another approach can be client remembers the last timestamp of the write made and then uses this information to read only from those replicas which have a more recent timestamp than the client’s timestamp
2. Moving backwards in time: User submits a write that gets replicated on one of the slaves, s1 and not on the others due to replication delays. User then reads from slave s1 and sees the write made by him/her earlier. However he/she reads again, and this time say the read is handled by a slave which is not s1. User can be quite frustrated to watch his/her writes disappearing. Such problems can be resolved by handling reads for one client only from one replica.

There can be other subtle but operationally annoying issues, especially if you are operating in a choppy internet or full capacity. It’s important to consider replication lags in your overall system [design](https://hackernoon.com/tagged/design) for replicated data stores.

Now on to many master configuration. The most common use case for this configuration is when you have multiple data centres. Each data centre can have one master and rest of the nodes can be slaves.

The biggest advantage of many master configuration is performance. Your writes are distributed and no longer throttled by the capacity of single master. However, there are no free lunches in this world. What you have gained in performance is made up by handling additional complexity in case of concurrent writes.

In case of many master configuration, you can tie users editing their own data, to one master. Essentially from a user’s perspective, the configuration becomes single master. This is really neat as now you can avoid all merge conflicts due to concurrent writes on many masters. However if you cannot do this, then you again need to resolve conflicts by using one of the following strategies:

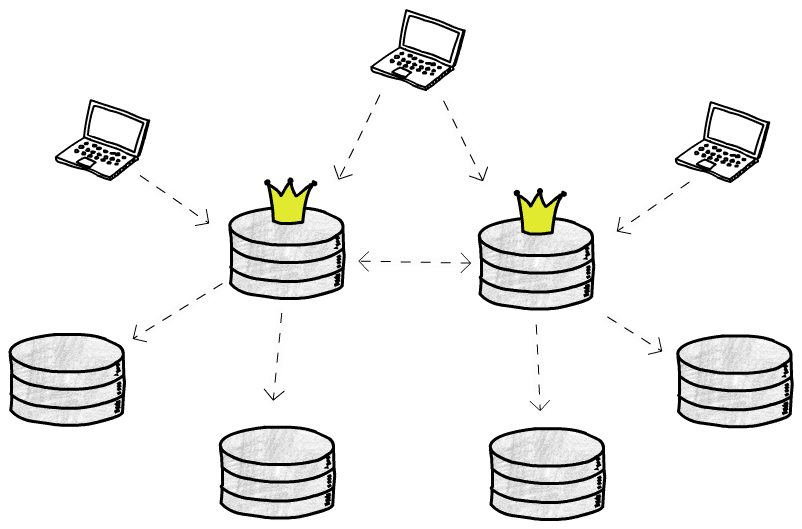
1. Make the most recent write win
2. Write custom code on detecting conflicts in replication logs
3. Present the user with a list of values, and let the user decide the merge and discard strategy in case of a conflict

Clearly the right strategy is dependent on the nature of your application and user expectations.

#### **Multi leader replication**

So, we talked a lot about single leader replication, now let’s discuss an alternative, and also explore its own challenges and try to identify scenarios where it might make sense to use it.

The main reason to consider a multi leader approach is that is solves some of the problems that we face when we have just one leader node. Namely, we have more than one node handling writes, and these writes can be performed by databases that are closer to the clients.



If your application needs to handle a very high number of writes, it might make sense to split that work across multiple leaders. Also, if the latency price to write in a database that is very far is too high, you could have one leader in each location (for example, one in North America, one in Europe and another in Asia).

Another good use case is when you need to support offline clients, that might be writing to their own (leader) database, and these writes need to be synchronized with the rest of the databases once this client gets online again.

The main problem you will face with multiple leaders accepting writes is that you need some way to solve conflicts. For example, let’s say you have a database constraint to ensure that your users’ emails are unique. If two clients write to two different leaders that are not yet in sync, both writes will succeed in their respective leaders, but we will have problems when we try to replicate that data. Let’s talk a bit more about these conflicts.

*Here we are assuming that these leaders replicate data asynchronously, that’s why we can have conflicts. You could, in theory, have multi leader synchronous replication, but that doesn’t really make a lot of sense, as you lose the main benefit of having leaders accepting writes independently, and might just use single leader replication instead. There are some projects, though, like* [*PgCluster*](https://wiki.postgresql.org/wiki/PgCluster)*, that implement multi master synchronous replication, but they are mostly abandoned, and I will not talk about this type of replication here.*

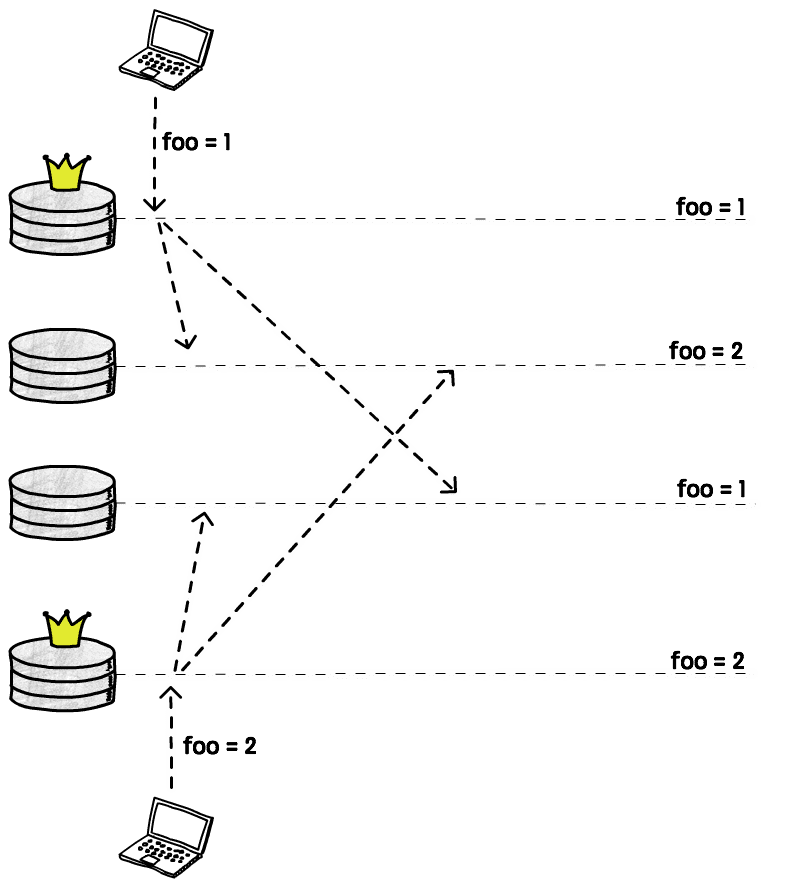
##### **Dealing with conflicts**

The easiest way to handle conflicts is to not have conflicts in the first place. Not everyone is lucky enough to be able to do that, but let’s see how that could be achieved.

Let’s use as an example an application to manage the projects in your company. You can ensure that all the updates in the projects related to the American office are sent to the leader in North America, and all the European projects are written to the leader in Europe. This way you can avoid conflicts, as the writes to the same projects will be sent to the same leader. Also, if we assume that the clients updating these projects will probably be in their respective offices (e.g. people in the New York office will update the American projects, that will be sent to the leader in North America), we can ensure that they are accessing a database geographically close to them.

Of course, this is a very biased example, and not every application can “partition” its data in such an easy way, but it’s something to keep in mind. If that’s not your case, we need another way to make sure we end up in a consistent state.

We cannot let each node just apply the writes in the order that they see them, because a node A may first receive an update setting foo=1 and then another update setting foo=2, while node B receive these updates in the opposite order (remember, these messages are going through the network and can arrive out of order), and if we just blindly apply them we would end up with foo=2 on node A and foo=1 on node B. Not good.



One common solution is to attach some sort of timestamp to each write, and then just apply the write with the highest value. This is called LWW (last write wins). As we discussed previously, with this approach we may lose data, but that’s still very widely used.

*Just be aware that physical clocks* [*are not reliable*](http://books.cs.luc.edu/distributedsystems/clocks.html)*, and when using timestamps you will probably need at least some sort to clock synchronization, like* [*NTP*](https://en.wikipedia.org/wiki/Network_Time_Protocol)*.*

Another solution is to record these conflicts, and then write application code to allow the user to manually resolve them later. This may not be feasible in some cases, like in our previous example with the unique constraint for the email column. In other cases, though, it may be just a matter of showing two values and letting the user decide which one should be kept and which should be thrown away.

Lastly, some databases and replication tools allow us to write custom conflict resolution code. This code can be executed on write or on read time. For instance, when a conflict is detected a stored procedure can be called with the conflicting values and it decides what to do with them. This is a *on write* conflict resolution. [Bucardo](https://bucardo.org/wiki/Bucardo) and [BDR](http://bdr-project.org/docs/stable/conflicts.html) are example of tools that use this approach.

Other tools use a different approach, storing all the conflicting writes, and also returning all of them when a client tries to read that value. The client is then responsible for deciding what to do with those values, and write it back to the database. [CouchDB](http://couchdb.apache.org/), for example, does that.

There is also a relatively new family of data structures that provide automatic conflict resolution. They are called *Conflict-free replicated data type*, or *CRDT*, and to steal the [wikipedia](https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type) definition:

*CRDT is a data structure which can be replicated across multiple computers in a network, where the replicas can be updated independently and concurrently without coordination between the replicas, and where it is always mathematically possible to resolve inconsistencies which might result.*

Unfortunately there are some limitations to where these data structured can be used (otherwise our lives would be too easy, right?), and as far as I know they are still not very widely used for conflict resolution in databases, although some CRDTs were implemented in [Riak](https://gist.github.com/russelldb/f92f44bdfb619e089a4d).

##### **DDL replication**

Handing DDLs (changes in the structure of the database, like adding/removing a column) can also be tricky in a multi leader scenario. It’s, in some sense, also a conflict issue, we cannot change the database structure while other nodes are still writing to the old structure, so we usually need to get a global database lock, wait until all the pending replications take place, and then execute this DDL. In the meantime, all the writes will either be blocked or fail. Of course, the specific details will depend on the database or replication tool used, and some of them will [not even try](https://bucardo.org/wiki/Bucardo/FAQ#Can_Bucardo_replicate_DDL.3F) to replicate DDLs, so you need to somehow do that manually, and other tools will replicate *some* types of DDLs, but not other (for instance, DDLs that need to rewrite the entire table are forbidden in [BDR](http://bdr-project.org/docs/1.0/ddl-replication-statements.html#DDL-REPLICATION-PROHIBITED-COMMANDS)).

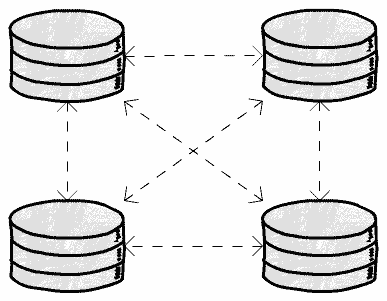
The point is, there is a lot more coordination involved in replicating DDLs when you have multiple leaders, so that’s also something to keep in mind when considering this setup.

##### **The topologies of a multi leader setup**

There are several different kinds of topologies that we can use with multiple leaders. A topology defines the communication patterns between your nodes, and different ways to arrange your communication paths have different characteristics.

If you have only two leaders, there are not a lot of options: Node A sends updates to node B, and node B sends updates to node A. Things start to get more interesting when you have three or more leaders.

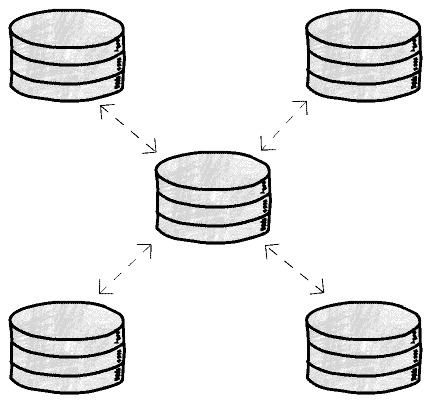
A common topology is to have each leader sending its updates to every other leader.



The main problem here is that messages can arrive out of order. For example, if a node A inserts a row, and then node B updates this row, but node C receives the update before the insert, we will have problems.

This is a *causality problem*, we need to make sure that all the nodes first process the insert event before processing the update event. There are different ways to solve this problem (for instance, using [logical clocks](https://en.wikipedia.org/wiki/Logical_clock)), but the point is: You need to make sure your database or replication tool is actually handling this issue, or, in case it’s not, be aware that this is a failure that can happen.

Another alternative is to use what some databases call the *star topology*.



In this case one node receives the updates and sends them to everyone else. With this topology we can avoid the causality problem but, on the other hand, introduce a single point of failure. If this central node dies, the replication will stop. It’s high price to pay, in some cases.

Of course, these are just 2 examples, but the imagination is the limit for all the different topologies you can have, and there’s no perfect answer, each one will have their pros and cons.

Finally let me introduce no master replication. In this setup, every node can accept both read and write request. You can immediately see a problem with this approach. Not only your reads can be stale because of replication lags, your writes can also be inconsistent.

To solve this problem reads and writes are sent in parallel to many nodes. Lets say r reads are made in parallel, w writes are made in parallel and there are a total of n nodes. As long as r + w > n, you can be sure that at least one of the r nodes must be up to date and consequently reads will not be stale.

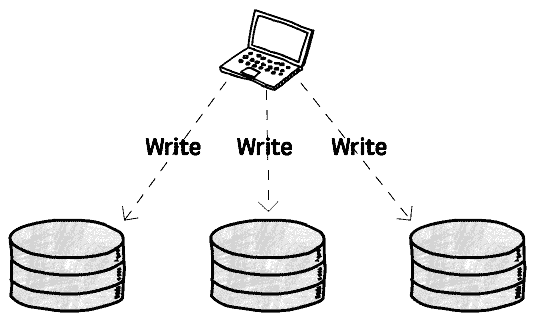
#### **leaderless replication**

Another idea that was popularized by Amazon’s [DynamoDB](https://aws.amazon.com/dynamodb/) (although it first appeared some decades ago) is to simply have no leaders, every replica can accept writes (maybe it should be called leaderful?).

It seems like this is going to be a mess, doesn’t it? If we had lots of conflicts to handle with a few leaders, imagine what will happen when writes are taking place everywhere. Chaos!

Well, it turns out these database folks are quite smart, and there are some clever ways to deal with this chaos.

The basic idea is that clients will send writes not only to one replica, but to several (or, in some cases, to all of them).



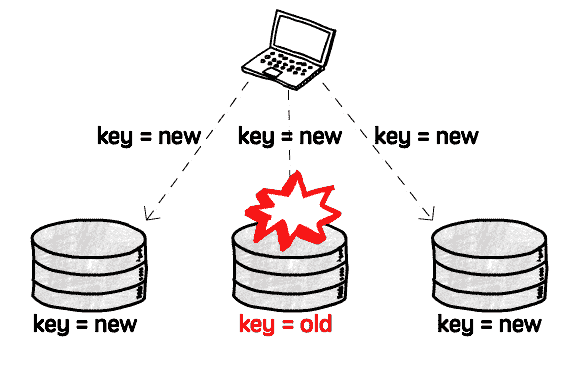
The client sends this write request concurrently to several replicas, and as soon as it gets a confirmation from some of them (we will talk about how many are “some” in a bit) it can consider that write a success and move on.

One advantage we have here is that we can tolerate node failures more easily. Think about what would happen in a scenario where we had to send a write to a single leader and for some reason that leader didn’t respond. The write would fail and we would need to start a failover process to elect a new leader that could start receiving writes again. No leaders, no failover, and if you remember what we’ve talked about failovers, you can probably see why this can be a big deal.

But, again, there is no free lunch, so let’s take a look at the price tag here.

What happens if, say, your write succeeds in 2 replicas, but fails in 1 (maybe that server was being rebooted when you sent the write request)?

You now have 2 replicas with the new value and 1 with the old value. Remember, these replicas are not talking to each other, there’s no leader handling any kind of synchronization.



Now if you read from this replica, BOOM, you get stale data.

To deal with this problem, a client will not read data from one replica, but also send requests to several replicas concurrently (like it did for writes). The replicas then return their values, and also some kind of version number, that the clients can use to decide which value it should use, and which it should discard.

We still have a problem, though. One of the replicas still has the old value, and we need to somehow synchronize it with the rest of the replicas (after all, replication is the process of keeping the *same* data in several places).

There are usually two ways to do that: We can make the client responsible for this update, or we can have another process that is responsible just for finding differences in the data and fixing them.

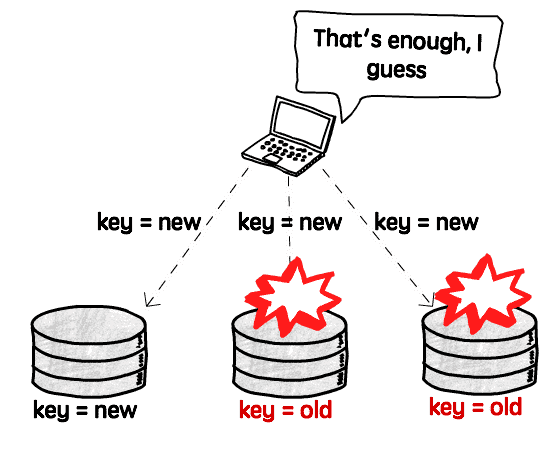
Making the client fix it is conceptually simple, when the client reads data from several nodes and detects that one of them is stale, it sends a write request with the correct value. This is usually called *read repair*.

The other solution, having a background process fixing the data, really depends on the database implementation, and there are several ways to do that, depending on how the data is stored. For example, DynamoDB has an *anti-entropy* process using Merkle trees.

##### **Quorums**

So we said we need to send the write/read requests to “some” replicas. There are good ways to define how many are enough, and what we are compromising (and gaining) if we decide to decrease this number.

Let’s first talk about the most obvious problematic scenario, when we require just one successful response to consider a value written, and also read from just one replica. From there we can expand the problem to more realistic scenarios.



As there is no synchronization between these replicas, we will read stale values every time we send a read request to a node other than the only one that succeeded.

Now let’s imagine we have 5 nodes and require a successful write in 2 of them, and also read from 2. Well, we will have the exact same problem. If we write to nodes A and B and read from nodes C and D, we will always get stale data.

What we need is some way to guarantee that at least one of the nodes that we are reading from is a node that received the write, and that’s what quorums are.

For example, if we have 5 replicas and require that 3 of them accept the write, and also read from 3 replicas, we can be sure that *at least* one of these replicas that we are reading from accepted the write and therefore has the most recent data. There’s always an overlap.

Most databases allow us to configure how many replicas need to accept a write (w) and how many we want to read from (r). A good rule of thumb is to always have w + r > number of replicas.

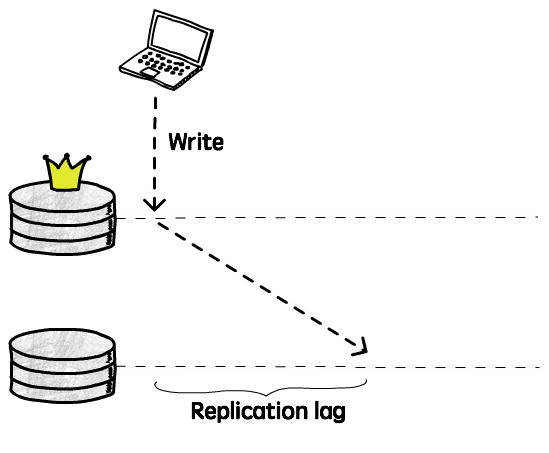
Now you can start playing with these numbers. For example, if your application writes to the database very rarely, but reads very frequently, maybe you can set w = number of replicas and n = 1. What that means is that writes need to be confirmed by every replica, but you can then read from just one of them as you are sure every replica has the latest value. Of course, you are then making your writes slower and less available, as just a single replica failure will prevent any write from happening, so you need to measure your specific needs and what is the right balance.

#### **Replication lag**

In a leader-based replication, as we have seen, writes need to be sent to a leader, but reads can be performed by any replica. When we have applications that are mostly reading from the database, and writing a lot less often (which is the most common case), it can be tempting to add many replicas to handle all these read requests, creating what can be called a *read scaling architecture*. Not only that, but we can have many replicas geographically close to our clients to also improve latency.

The more replicas we have, though, the harder it is to use synchronous replication, as the probability of one these many nodes being down when we need to replicate an update increases, and our availability decreases. The only feasible solution in this case is to use asynchronous replication, that is, we can still perform updates even if a node is not responding, and when this replica is back up it should catch up with the leader.

We’ve already discussed the benefits and challenges in using synchronous and asynchronous replication, so I’ll not talk about that again, but assuming we are replicating updates asynchronously, we need to be aware of the problems we can have with the replication lag, or, in other words, the delay between the time an update is applied in the leader node and the time it’s applied in a given replica.



If a client reads from this replica during this period, it will receive outdated information, because the latest update(s) were not applied yet. In other words, if you send the same query to 2 different server, you may get 2 different answers. As you may remember when we talked about the CAP theorem, this breaks the *consistency* guarantee. This is just temporary, though, eventually all the nodes replicas will get this update, and if you stop writing new data, they will all end up being identical. This is what we call *eventual consistency*.

In theory there is no limit for how long it will take to a replica to be consistent with its leader (the only guarantee we have is that *eventually* it will be), but in practice we usually expect this to happen fairly quickly, maybe in a couple of milliseconds.

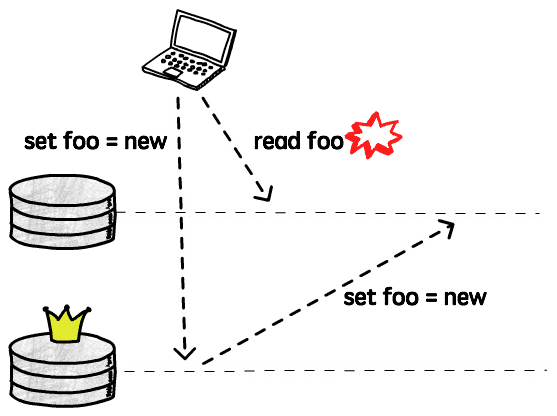
Unfortunately, we cannot expect that to always be the case, and we need to plan for the worst. Maybe the network is slow, or the server is operating near capacity and is not replicating the updates as fast as we’d except, and this replication lag can increase. Maybe it will increase a couple of seconds, maybe minutes. What happens then?

Well, the first step is to understand the guarantees we need to provide. For example, is it really a problem that, when facing an issue that increases the lag, it will take 30 seconds for your friend to be able to see that last cat picture you just posted on Facebook? Probably not.

In a lot of cases this replication lag, and eventual consistency in general, will not be a problem (after all, the physical world is eventually consistent), so let’s focus on some cases where this *can* be an issue, and see some alternatives to handle them.

##### **Read-your-writes consistency**

The most common problem we can have with asynchronous replicas is when a client sends a write to the leader, and shortly after tries to read that same value from a replica. If this read happens before the leader had enough time to replicate the update, it will look like the write didn’t actually work.



So, although it might not be a big issue if a client doesn’t see other clients’ updates right away, it’s pretty bad if they don’t see their own writes. This is what is called *read-your-writes consistency*, we want to make sure that a client never reads the database in a state it was before it performed a write.

Let’s talk about some techniques that can be used to achieve this type of consistency.

A simple solution is to actually read from the leader when we are trying to read something that the user might have changed. For example, if we are implementing something like Twitter, we can read other people’s timeline from a replica (as the user will not be able to write/change it), but when viewing their own timeline, read from the leader, to ensure we don’t miss any update.

Now, if there are lots of things that can be changed by every user, that doesn’t really work very well, as we would end up sending all the reads to the leader, defeating the whole purpose of having replicas, so in this case we need a different strategy.

Another technique that can be used is to track the timestamp of the last write request and for the next, say, 10 seconds, send all the read requests to the leader. Then you need to find the right balance here, because if there is a new write every 9 seconds you will also end up sending all of your reads to the leader. Also, you will probably want to monitor the replication lag to make sure replicas that fall more than 10 seconds behind stop receiving requests until they catch up.

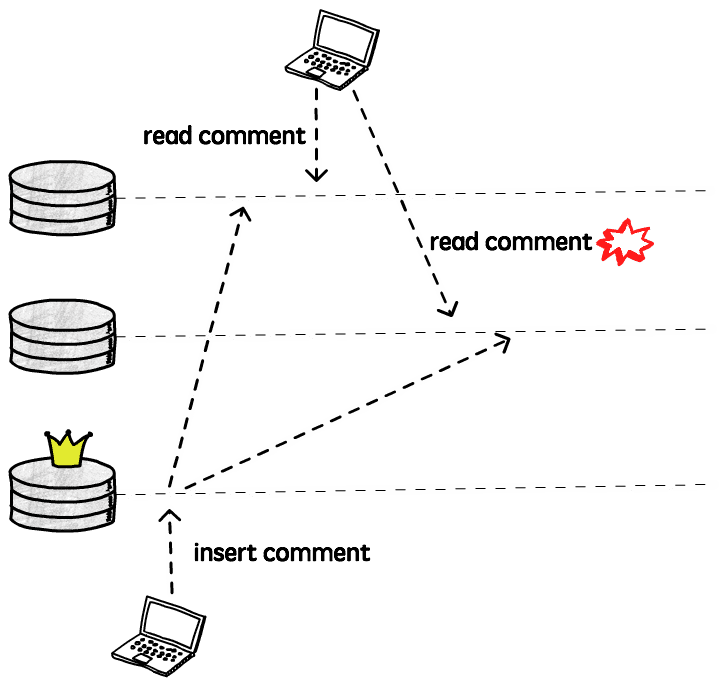
Then there are also more sophisticated ways to handle this, that requires more collaboration of your database. For example, [Berkeley DB](http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html) will generate a *commit token* when you write something to the leader. The client can then send this token to the replica it’s trying to read from, and with this token the replica knows if it’s current enough (i.e. if it has already applied that commit). If so, it can just serve that read without any problem, otherwise it can either block until it receives that update, and then answer the request, or it can just reject it, and the client can try another replica.

As always, there are no right answers here, and I am sure there lots of other techniques that can be used to work around this problem, but you need to know how your system will behave when facing a large replication lag and if *read-your-writes* is a guarantee you really need to provide, as there are databases and replication tools that will simply ignore this issue.

##### **Monotonic Reads consistency**

This is a fancy name to say that we don’t want clients to see time moving backwards: If I read from a replica that has already applied commits 1, 2 and 3, I don’t want my next read to go to a replica that only has commits 1 and 2.

Imagine, for example, that I’m reading the comments of a blog post. When I refresh the page to check if there’s any new comment, what actually happens is that the last comment disappears, as if it was deleted. Then I refresh again, and it’s back there. Very confusing.



Although you can still see stale data, what monotonic read guarantees is that if you make several reads to a given value, all the successive reads will be at least as recent as the previous one. Time never moves backwards.

The simplest way to achieve monotonic reads is to make each client send their read requests to the same replica. Different clients can still read from different replicas, but having a given client always (or at least for the duration of a session) connected to the same replica will ensure it never reads data from the past.

Another alternative is to have something similar to the commit token that we talked about in the *read-your-writes* discussion. Every time that a client reads from a replica it receives its latest commit token, that is then sent in the next read, that can go to another replica. This replica can then check this commit token to know if it’s eligible to answer that query (i.e. if its own commit token is “greater” than the one received). If that’s not the case, it can wait until more data is replicated before responding, or it can return an error.

##### **Bounded Staleness consistency**

This consistency guarantee means, as the name indicates, that there should be a limit on how stale the data we are reading is. For example, we may want to guarantee that clients will not read data that is more than 3 minutes old. Alternatively, this staleness can be defined in terms of number of missing updates, or anything that is meaningful the application.

#### **Delayed replicas**

We talked about replication lags, some of the problems that we can have when this lag increases too much, and how to deal with these problems, but sometimes we may actually *want* this lag. In other words, we want a *delayed replica*.

We will not really read (or write) from this replica, it will just sit there, lagging behind the leader, maybe by a couple of hours, while no one is using it. So, why would anyone want that?

Well, imagine that you release a new version of your application, and a bug introduced in this release starts deleting all the records from your orders table. You notice the issue and rollback this release, but the deleted data is gone. Your replicas are not very useful at this point, as all these deletions were already replicated and you have the same messy database replicated. You could start to restore a backup, but if you have a big database you probably won’t have a backup running every couple of minutes, and the process to restore a database can take a lot of time.

That’s were a delayed replica can save the day. Let’s say you have a replica that is always 1 hour behind the leader. As long as you noticed the issue in less than 1 hour (as you probably will when your orders evaporate) you can just start using this replica and, although you will still probably lose some data, the damage could be a lot worse.

A replica will almost never replace a proper backup, but in some cases having a delayed replica can be extremely helpful (as the developer that shipped that bug can confirm).

#### **Replication under the hood**

We talked about several different replication setups, consistency guarantees, benefits and disadvantages of each approach. Now let’s go one level below, and see how one node can actually send its data to another, after all, replication is all about copying bytes from one place to another, right?

##### **Statement-based replication**

Statement-based replication basically means that one node will send the same statements it received to its replicas. For example, if you send an UPDATE foo

= bar statement to the leader, it will execute this update and send the same instruction to its replicas, that will also execute the update, hopefully getting to the same result.

Although this is a very simple solution, there are some things to be considered here. The main problem is that not every statement is deterministic, meaning that each time you execute them, you can get a different result. Think about functions like CURRENT\_TIME() or RANDOM(), if you simply execute these functions twice in a row, you will get different results, so just letting each replica re-execute them would lead to inconsistent data.

Most databases and replication tools that use statement-based replication (e.g. MySQL before 5.1) will try to replace these nondeterministic function calls with fixed values to avoid these problems, but it’s hard to account for every case. For example, a user-defined function can be used, or a trigger can be called after an update, and it’s hard to guarantee determinism in these cases. [VoltDB](https://www.voltdb.com/), for instance, uses logical replication but [requires stored procedures to be deterministic](https://docs.voltdb.com/UsingVoltDB/DesignProc.php#DesignProcDeterminism).

Another important requirement is that we need to make sure that all transactions either commit or abort on every replica, so we don’t have a change being applied in some replicas and not in others.

##### **Log Shipping replication**

Most databases use a [log](https://en.wikipedia.org/wiki/Write-ahead_logging) (an append-only data structure) to provide durability and atomicity (from the *ACID* properties). Every change is first written to this log before being applied, so the database can recover in case of crashes during a write operation.

The log describes changes to the database at a very low level, describing, for example, which bytes were changed and where exactly in the disk. It’s not meant to be read by humans, but machines can interpret them pretty efficiently.

The idea of log shipping replication is to transfer these log files to the replicas, that can then apply them to get the exact same result.

The main limitation that we have when shipping these logs is that, as it describes the changes at such a low level, we probably won’t be able to replicate a log generated by a different version of the database, for example, as the way the data is physically stored may have changed.

Another issue is that we cannot use multi-master replication with log shipping, as there’s no way to unify multiple logs into one, and if data is changing in multiple locations at the same time, that would be necessary.

This technique is used by Postgres’ [streaming replication](https://www.postgresql.org/docs/current/static/warm-standby.html) and also to provide [incremental backups and Point-in-Time Recovery](https://www.postgresql.org/docs/9.6/static/continuous-archiving.html).

This is also known as *physical replication*.

##### **Row-based replication**

Row-based, or *logical* replication, is kind of a mix of these two techniques. Instead of shipping the internal log (WAL), it uses a different log just for replication. This log can then be decoupled from the storage engine and therefore can be used, in most cases, to replica data across different database versions.

This row-based log will include enough information to uniquely identify a row, and a set of changes that need to be performed.

A benefit of using a row-based approach is that we can, for example, upgrade the database version with zero downtime. We can take one node down to upgrade it, and in the meantime the other replicas handle all the requests, and after it’s back up, with the new version, we do the same thing for the other nodes.

The main disadvantage here when compared to a statement-based replication is that sometimes we need to log a lot more data. For example, if we want to replicate UPDATE foo = bar, and this update changes 100 rows, with the statement-based replication we would log just this simple SQL, while we would need to log all the 100 rows when using the row-based technique. In the same way, if you use an user-defined function that generates a lot of data, all that data needs to be logged, instead of just the function call.

MySQL for example, allows us to define a [MIXED logging format](https://dev.mysql.com/doc/refman/8.0/en/binary-log-mixed.html), that will switch between statement and row-base replication, trying to use the best technique for each case.

<https://www.brianstorti.com/replication/>