Redis Architecture

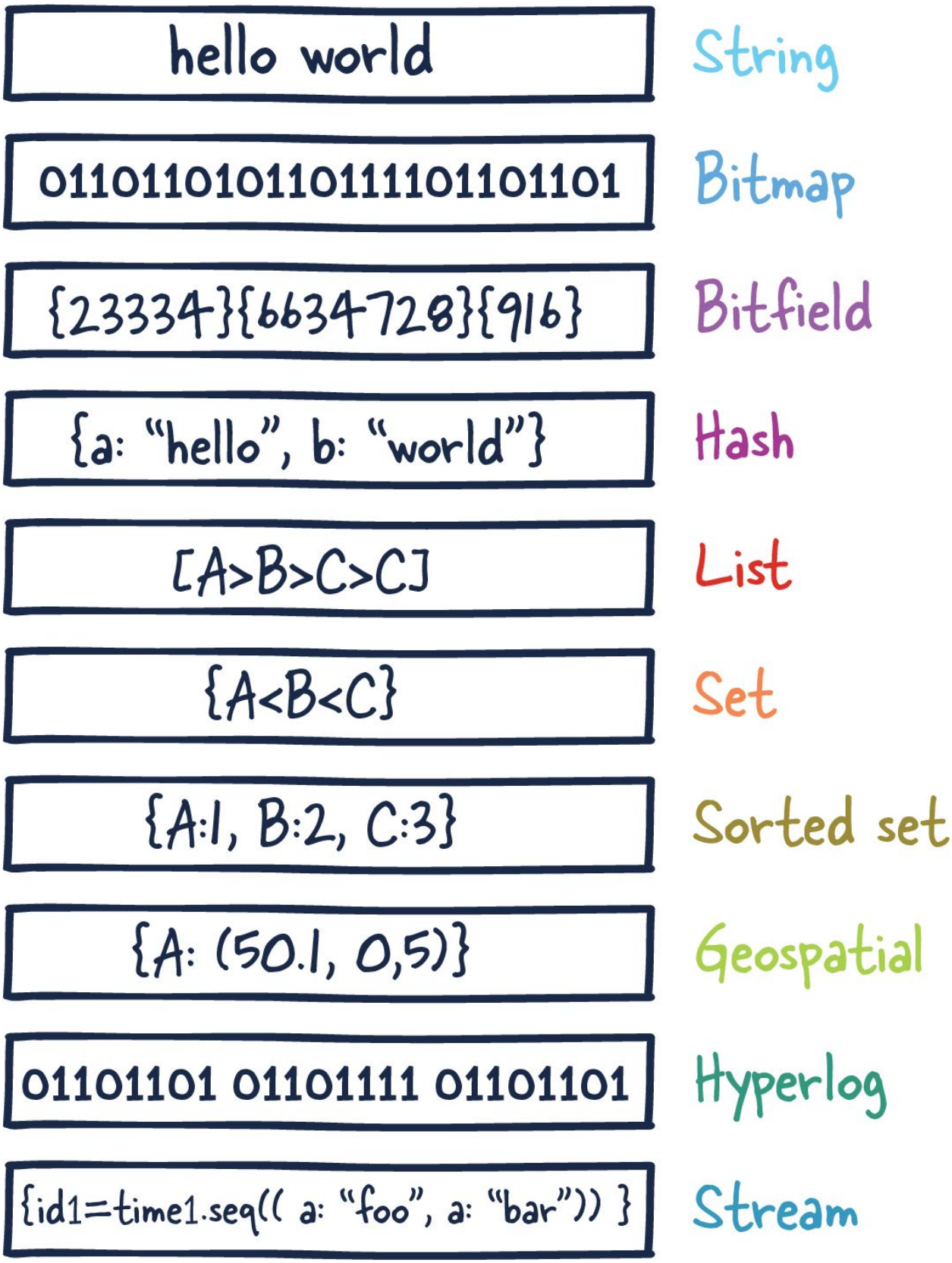
What is Redis?

Redis (“***RE***mote ***DI***ctionary ***S***ervice”) is an open-source key-value database server.

The most accurate description of Redis is that it's a data structure server.

Rather than iterating over, sorting, and ordering rows, what if the data was in data structures you wanted from the ground up? Early on

it was used much like Memcached, but as Redis improved, it became viable for many other use cases, including publish-subscribe mechanisms, streaming, and queues.

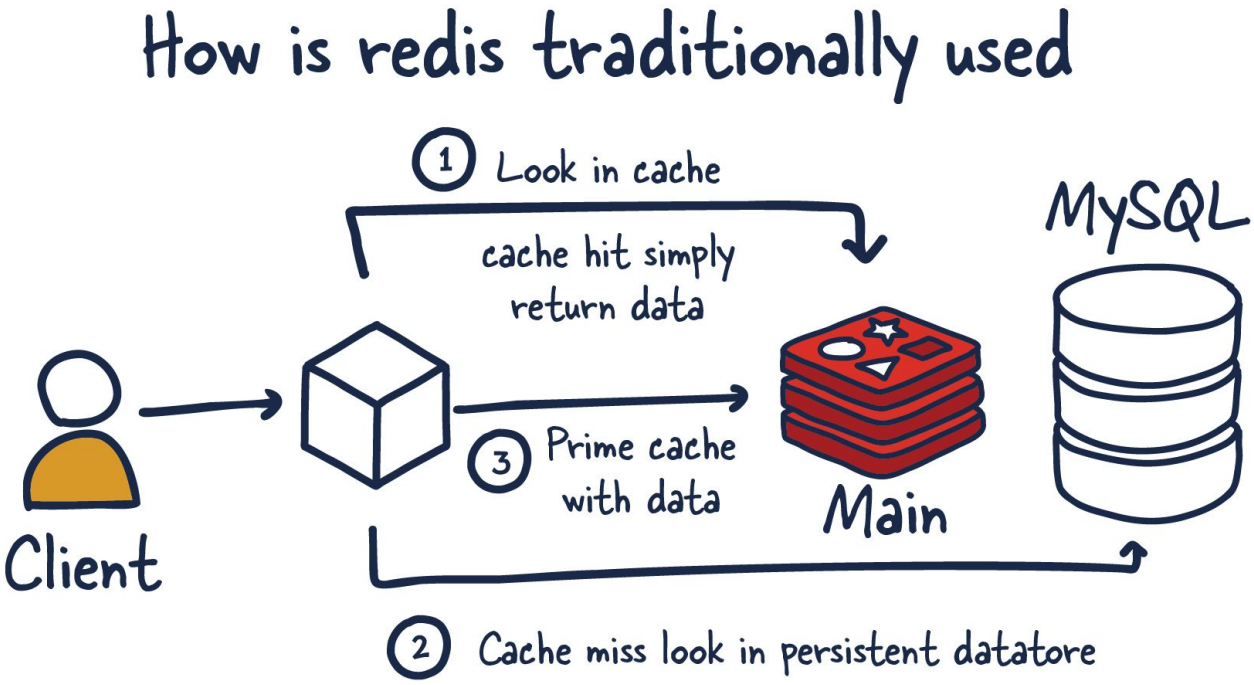

hello world
String
0110110101101111101101101
Bitmap
{23334}{6634728}{916}
Bitfield
{a: "hello", b: "world"}
Hash
[A>B>C>C]
List
{A<B<C}
Set
{A:1, B:2, C:3}
Sorted set
{A: (50.1, 0,5)}
Geospatial
01101101 01101111 01101101
Hyperlog
{id1=time1.seq((a: "foo", a: "bar")) }
Stream

Primarily, Redis is an in-memory database used as a cache in front of another "real" database like MySQL or PostgreSQL to help improve application performance.

It leverages the speed of memory and alleviates load off the central application database for:

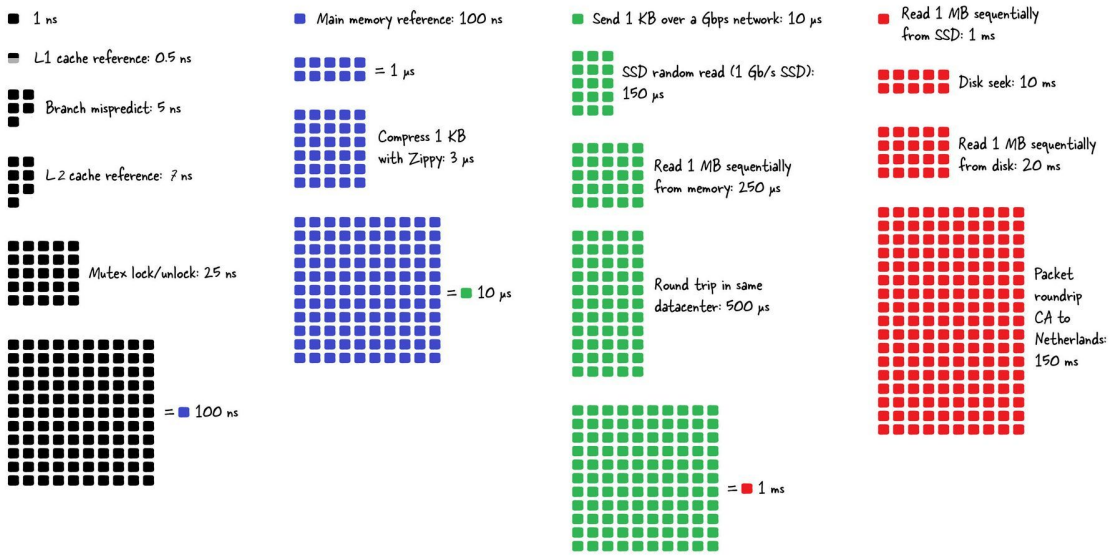
- Data that changes infrequently and is requested often
- Data that is less mission-critical and is frequently evolving.

Examples of above data can include session or data caches and leaderboard or roll-up analytics for dashboards.



Coupled with Redis plug-ins and its various High Availability (HA) setups, Redis as a database has become incredibly useful for certain scenarios and workloads.

Important note to understand here is that reading and manipulating data in memory is much faster than anything possible in traditional datastores using SSDs or HDDs.



	Memcached	Redis
Sub-millisecond latency	Yes	Yes
Developer ease of use	Yes	Yes
Data partitioning	Yes	Yes
Support for a broad set of programming languages	Yes	Yes
Advanced data structures	-	Yes
Multithreaded architecture	Yes	-
Snapshots	-	Yes
Replication	-	Yes
Transactions	-	Yes
Pub/Sub	-	Yes

Lua scripting	-	Yes
Geospatial support	-	Yes

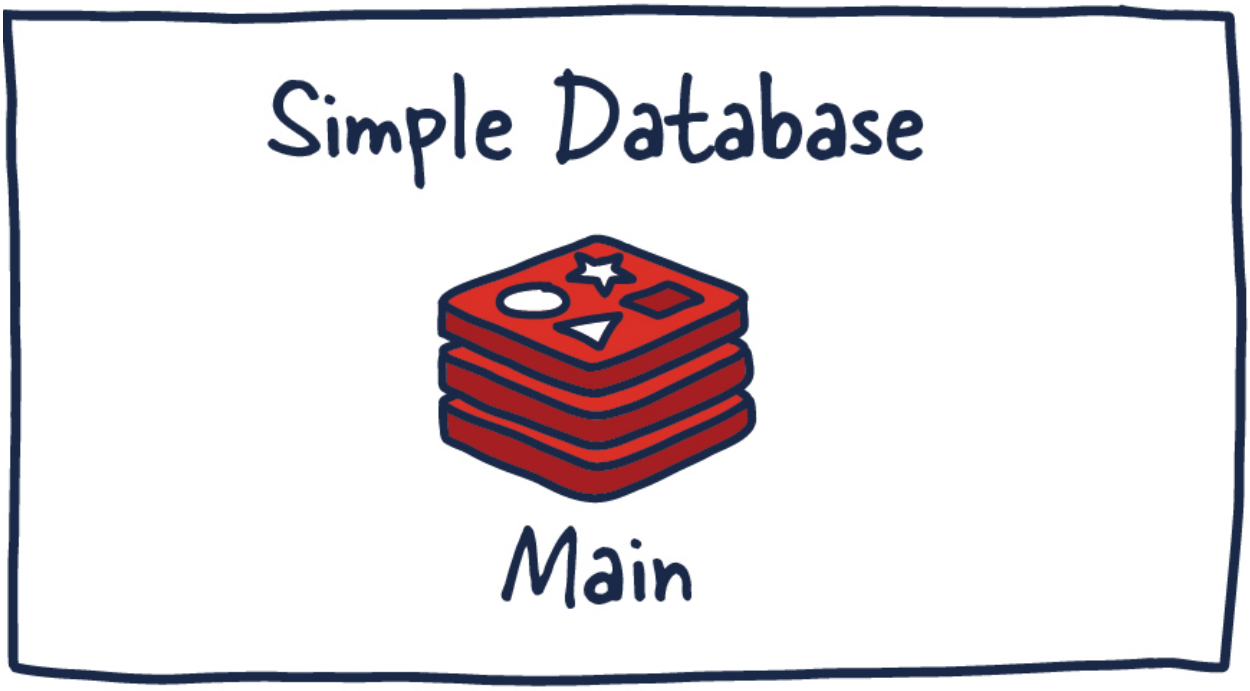
Redis Architectures

Before we start discussing Redis internals, let's discuss the various Redis deployments and their trade-offs.

1. Single Redis Instance
2. Redis HA
3. Redis Sentinel
4. Redis Cluster

Depending on your use case and scale, you can decide to use one setup or another.

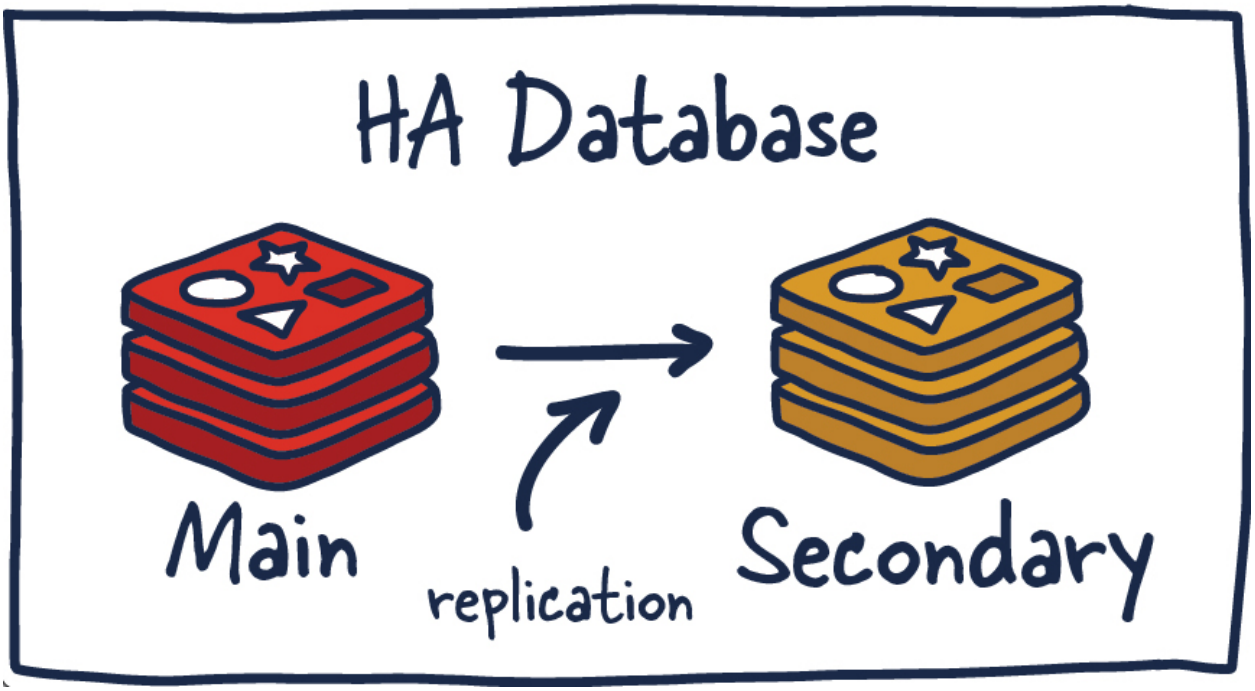
Single Redis Instance



- It allows users to set up and run small instances that can help them grow and speed up their services
- However, this deployment isn't without shortcomings. For example, if this instance fails or is unavailable, all client calls to Redis will fail and therefore degrade the system's overall performance and speed.
- Given enough memory and server resources, this instance can be powerful.
- This scenario primarily used for caching could result in a significant performance boost with minimal setup. Given enough system resources, you could deploy this Redis service on the same box the application is running.
- If persistence is set up on these instances, there is a forked process on some interval that facilitates data persistence RDB (very compact point-in-time representation of Redis data) snapshots or AOF (append-only files).
- These two flows allow Redis to have long-term storage, support various replication strategies, and enable more complicated topologies.
- If Redis isn't set up to persist data, data is lost in case of a restart or failover.

- If the persistence is enabled on a restart, it loads all of the data in the RDB snapshot or AOF back into memory, and then the instance can support new client requests.

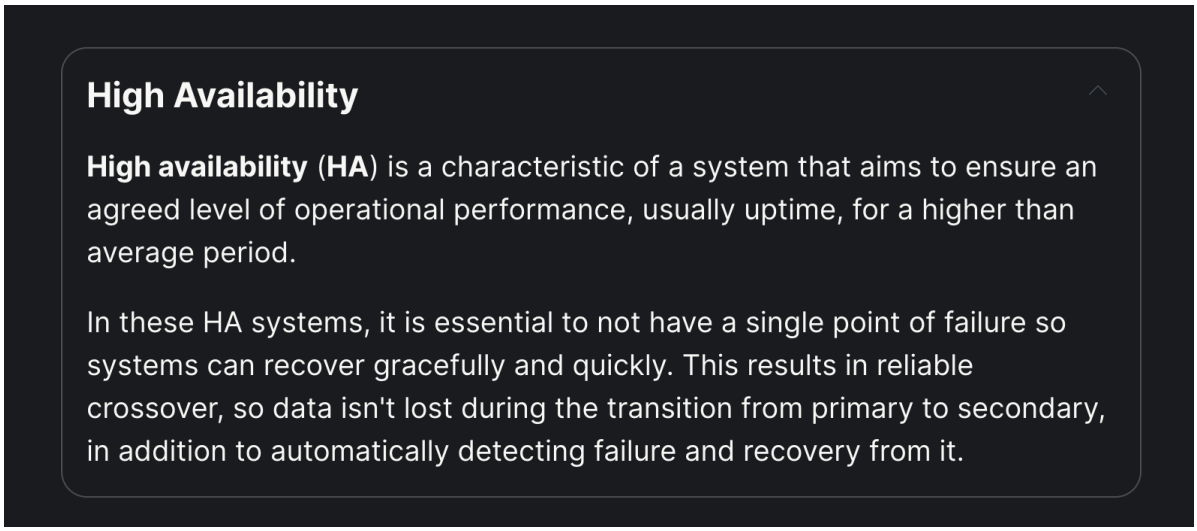
Redis HA



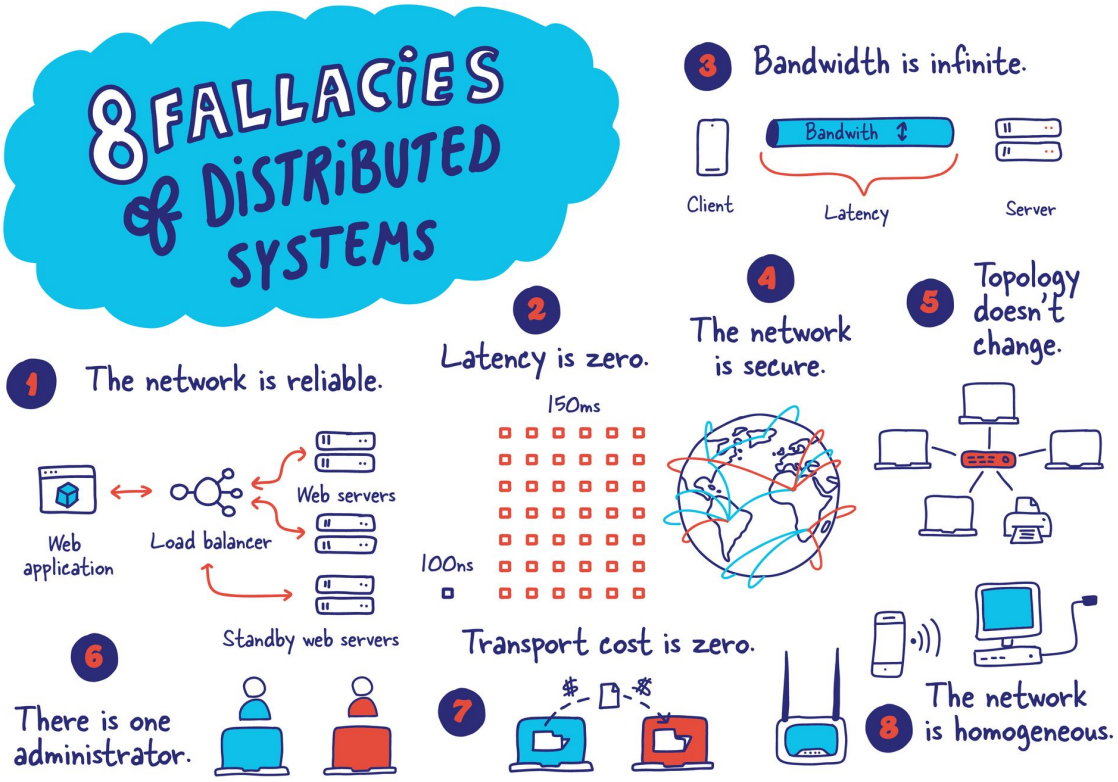
- Another popular setup with Redis is the main deployment with a secondary deployment that is kept in sync with replication.
- As data is written to the main instance it sends copies of those commands, to a replica client output buffer for secondary instances which facilitates replication.
- The secondary instances can be one or more instances in your deployment. These instances can help scale reads from Redis or provide failover in case the main is

lost.

▼


High Availability
High availability (**HA**) is a characteristic of a system that aims to ensure an agreed level of operational performance, usually uptime, for a higher than average period.
In these HA systems, it is essential to not have a single point of failure so systems can recover gracefully and quickly. This results in reliable crossover, so data isn't lost during the transition from primary to secondary, in addition to automatically detecting failure and recovery from it.

▼



Redis Replication

- Every main instance of Redis has a replication ID and an offset.
- These two pieces of data are critical to figure out a point in time where a replica can continue its replication process or to determine if it needs to do a complete sync.
- This offset is incremented for every action that happens on the main Redis deployment.
- More explicitly, when the Redis replica instance is just a few offsets behind the main instance, it receives the remaining commands from the primary, which is then replayed on its dataset until it is in sync.
- If the two instances cannot agree on a replication ID or the offset is unknown to the main instance, the replica will then request a full synchronization.

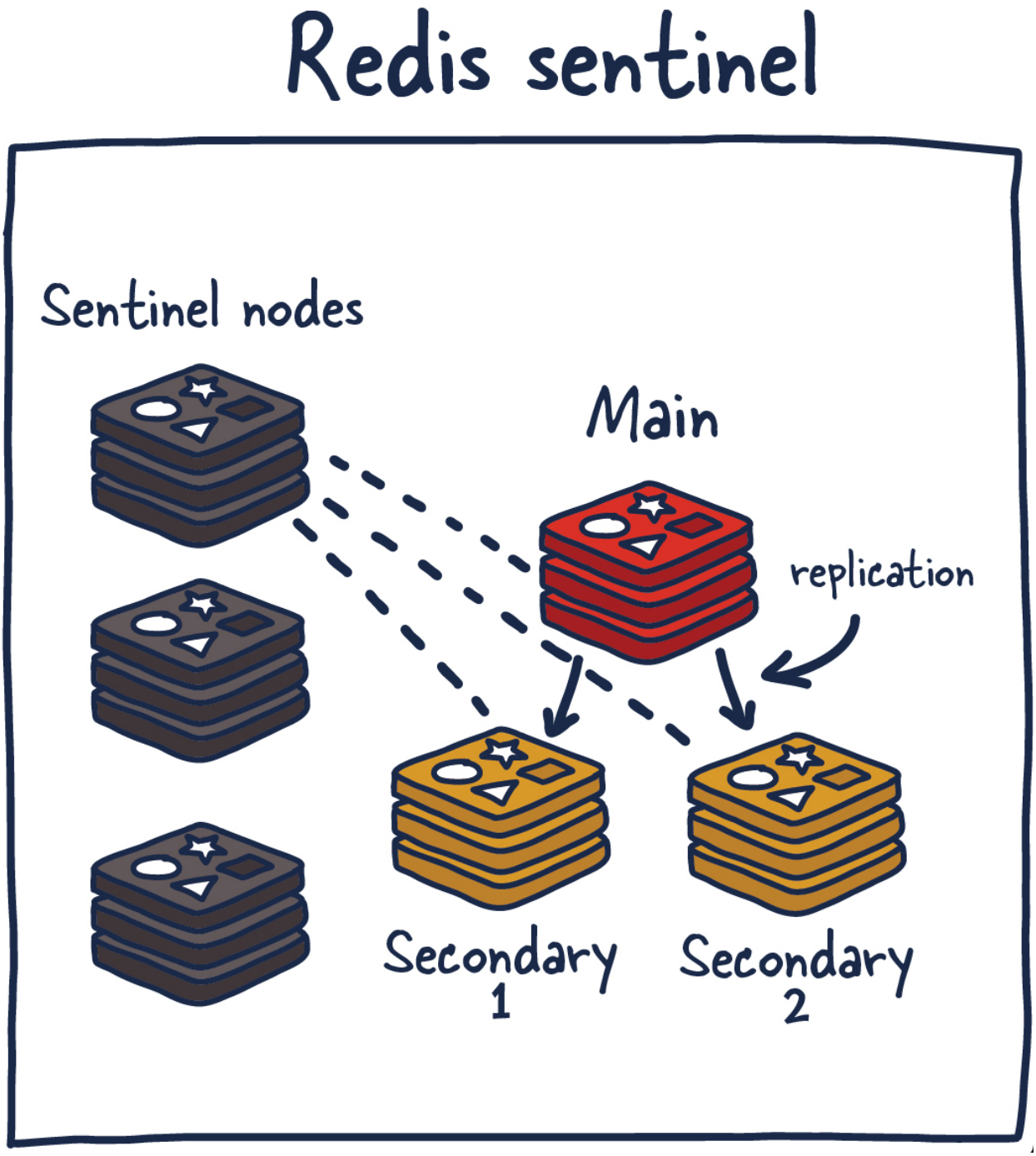
- This involves a primary instance creating a new RDB snapshot and sending it over to the replica.
- While this transfer is happening, the main instance is buffering all the intermediate updates between the snapshot cut-off and current offset to send to the secondary once it is in sync with the snapshot.
- Once complete, replication can continue as normal.
- If an instance has the same replication ID and offset, they have precisely the same data.

- Now you may be wondering why a replication ID is required. When a Redis instance is promoted to primary or restarts from scratch as a primary, it is given a new replication ID.
- This is used to infer the prior primary instance from which this newly promoted secondary was replicating.
- This allows for the ability to perform a partial synchronization (with other secondaries) since the new primary instance remembers its old replication ID.

- For example, two instances, primary and secondary, have the identical replication ID but offsets that differ by a few hundred commands, meaning that if those were replayed on the instance that is just behind in offset, they would have the same dataset.

- Now if the replication IDs differ entirely, and when we are unaware of the previous replication ID (no common ancestor) of the newly demoted (and rejoining) secondary. We will need to perform an expensive full sync.

Redis Sentinel



- Sentinel is a distributed system.
- As with all distributed systems, Sentinel comes with several advantages and disadvantages.
- Sentinel is designed in a way where there is a cluster of sentinel processes working together to coordinate state to provide high availability for Redis.

- Sentinel is responsible for a few things. First, it ensures that the current main and secondary instances are functional and responding.
- This is necessary because sentinel (with other sentinel processes) can alert and act on situations where the main and/or secondary nodes are lost.
- Second, it serves a role in service discovery much like Zookeeper and Consul in other systems.
- So when a new client attempts to write something to Redis, Sentinel will tell the client what current main instance is.

So sentinels are constantly monitoring availability and sending out that information to clients so they are able to react to them if they indeed do failover

Here are its responsibilities:

1. Monitoring **—** ensuring main and secondary instances are working as expected.
2. Notification **—** notify system admins about occurrences in the Redis instances.
3. Failover management — Sentinel nodes can start a failover process if the primary instance isn't available and enough (quorum of) nodes agree that is true.
4. Configuration management — Sentinel nodes also serve as a point of discovery of the current main Redis instance.

Using Redis Sentinel in this way allows for failure detection.

This detection involves multiple sentinel processes agreeing that current main instance is no longer available.

This agreement process is called Quorum. This allows for increased robustness and protection against one machine misbehaving and being unable to reach the main Redis node.

This setup isn't without its disadvantages so we are going to run through a few recommendations and best practices when using Redis Sentinel.

You can deploy Redis Sentinel in several ways.

- Sentinel node along aside each of your application servers (if possible) so you also don't need to factor in network reachability differences between sentinel nodes and clients who are actually using Redis.

- You can run Sentinel alongside the Redis instances or even on independent nodes

Number of Servers	Quorum	Number Of Tolerated Failures
1	1	0
2	2	0
3	2	2
4	3	1
5	3	2
6	4	2
7	4	3

Let's take a moment to think through what could go wrong in such a setup. If you run this system long enough, you will run into all of them.

1. What if the sentinel nodes fall out of quorum?

2. What if there is a network split which puts the old main instance in the minority group? What happens to those writes? (Spoiler: they are lost when the system recovers fully)
3. What happens if the network topologies of sentinel nodes and client nodes (application nodes) are misaligned? 😬

There are no durability guarantees, especially since persistence (see below) to disk is asynchronous

There is also the nagging problem of *when* clients find out about new primaries, how many writes did we lose to an unaware primary?

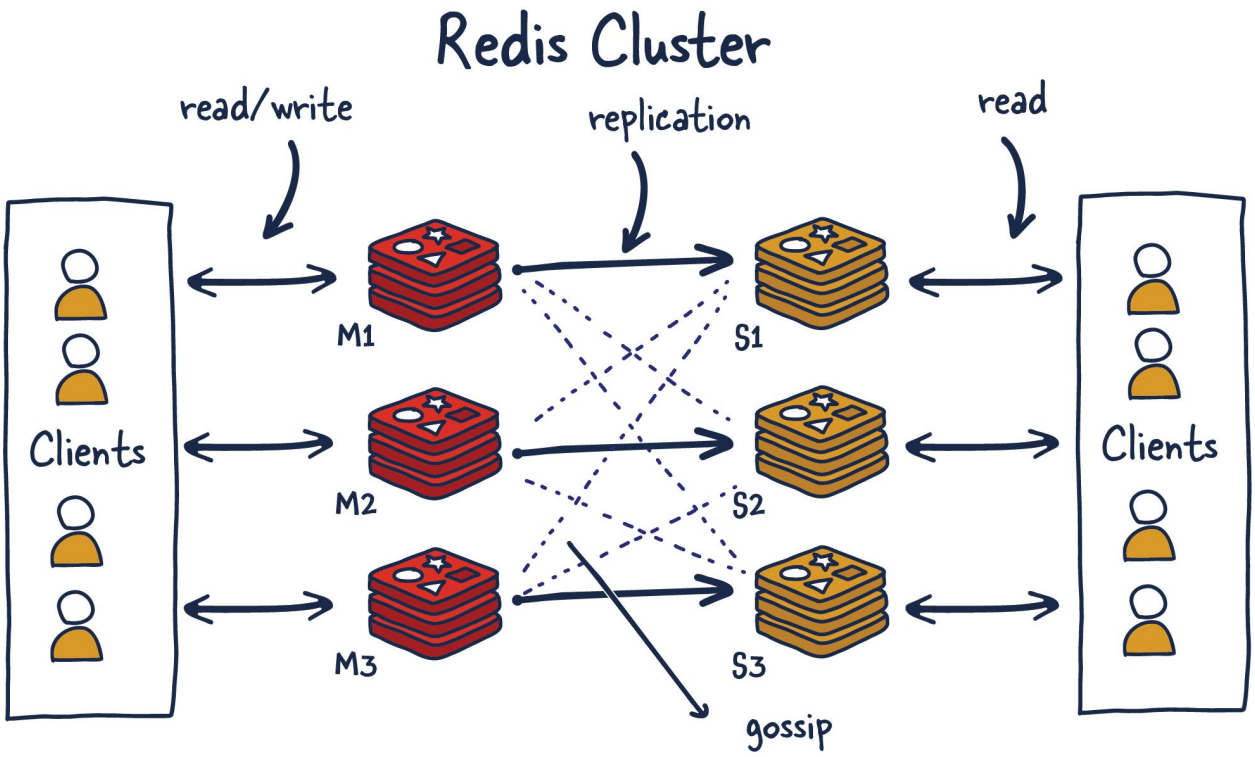
Redis recommends that when new connections are established that they should query for the new primary. Depending on the system configuration, that could mean a significant data loss.

There are a few ways to mitigate the level of losses if you force the main instance to replicate writes to a minimum of one secondary instance.

Remember, all Redis replication is asynchronous and has its trade-offs. So it will need to independently track acknowledgement and if they aren't confirmed by at least one secondary, the main instance will stop accepting writes.

💡 Redis Enterprise automated HA & Fault Tolerance behind

Redis Cluster



I am sure many have thought about what happens when you can't store all your data in memory on one machine.

Currently, the maximum RAM available in a single server is 24TIB, presently listed online at AWS. Granted, that's a lot, but for some systems, that isn't enough, even for a caching layer.

Redis Cluster allows for the horizontal scaling of Redis.

▼

Vertical and Horizontal Scaling

As your systems grow, you have three options.

1. Do less (No one does this entirely because we are insatiable monsters).
2. Scale up / vertical scaling
3. Scale out / Horizontal scaling

Taking the latter two seriously, scaling up and scaling out are known as vertical and horizontal scaling, respectively. Vertical scaling is a technique where you get bigger and better machines to do the work faster and hope all your problems scale well with your hardware. Even if this is possible, you will eventually be limited by the hardware you use.

Once you reach that point (more likely and hopefully way before), you will need to scale your system horizontally by spreading the workload across multiple smaller machines responsible for smaller parts of the whole.

So let's get some terminology out of the way; once we decide to use Redis Cluster, we have decided to spread the data we are storing across multiple machines, known as sharding. So each Redis instance in the cluster is considered a shard of the data as a whole.

This brings about a new problem.
If we push a key to the cluster, how do we know which Redis instance (shard) is holding that data? There are several ways to do this, but Redis Cluster uses **algorithmic sharding**.

To find the shard for a given key, we hash the key and mod the total result by the number of shards. Then, using a **deterministic hash function**, meaning that a given key will always map to the same shard, we can reason about where a particular key will be when we read it in the future.

What happens when we later want to add a new shard into the system? This process is called resharding.

Assuming the key 'foo' was mapped to shard zero after introducing a new shard, it may map to shard five. However, moving data around to reflect the new shard mapping would be slow and unrealistic if we need to grow the system quickly. It also has adverse effects on the availability of the Redis Cluster.

Redis Cluster has devised a solution to this problem called **Hashslot**, to which all data is mapped. There are 16K hashslot. This gives us a reasonable way to spread data across the cluster, and when we add new shards, we simply move hashslots across the systems. By doing this, we just need to move hashslots from shard to shard and simplify the process of adding new primary instances into the cluster.

This is possible without any downtime, and minimal performance hit. Let's talk through an example.

M1 contains hashslots from 0 to 8191.

M2 contains hashslots from 8192 to 16383.

So to map `foo', we take a deterministic hash of the key (foo) and mod it by the number of hash slots(16K), leading to a mapping of M2. Now let's say we add a new instance, M3. The new mappings would be

M1 contains hashslots from 0 to 5460.

M2 contains hashslots from 5461 to 10922.

M3 contains hashslots from 10923 to 16383.

All the keys that mapped the hashslots in M1 that are now mapped to M2 would need to move. But the hashing for the individual keys to hashslots wouldn't need to move because they have already been divided up across hashslots. So this one level of misdirection solves the resharding issue with algorithmic sharding.

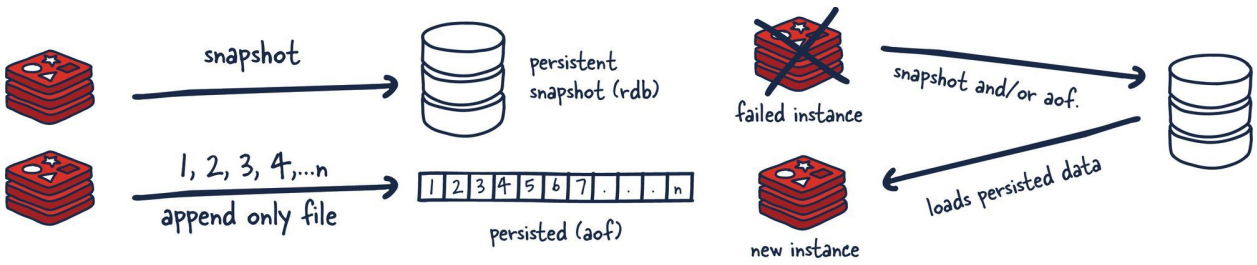
Gossiping

Redis Cluster uses gossiping to determine the entire cluster's health. In the illustration above, we have 3 M nodes and 3 S nodes. All these nodes constantly communicate to know which shards are available and ready to serve requests. If enough shards agree that M1 isn't responsive, they can decide to promote M1's secondary S1 into a primary to keep the cluster healthy. The number of nodes needed to trigger this is configurable, and it is essential to get this right. If you do it improperly, you can end up in situations where the cluster is split if it cannot break the tie when both sides of a partition are equal. This phenomenon is called split brain. As a general rule, it is essential to have an odd number of primary nodes and two replicas each for the most robust setup.

Redis Persistence Models

If we are going to use Redis to store any kind of data for safe keeping, it's important to understand how Redis is doing it. There are many use cases where if you were to lose the data Redis is storing is not the end of the world. Using it as a cache or in situations where its powering real-time analytics where if data loss occurs its no the end of the world.

In other scenarios, we want to have some guarantees around data persistence and recovery.



No persistence

No persistence: If you wish, you can disable persistence altogether. This is the fastest way to run Redis and has no durability guarantees.

RDB Files

RDB (Redis Database): The RDB persistence performs point-in-time snapshots of your dataset at specified intervals.

The main downside to this mechanism is that data between snapshots will be lost. In addition, this storage mechanism also relies on forking the main process, and in a larger dataset, this may lead to a momentary delay in serving requests. That being said, RDB files are much faster being loaded in memory than AOF.

AOF

AOF (Append Only File): The AOF persistence logs every write operation the server receives that will be played again at server startup, reconstructing the original dataset.

This way of ensuring persistence is much more durable than RDB snapshots since it is an append-only file. As operations happen, we buffer them to the log, but they aren't persisted yet. This log consists of the actual commands we ran in order for replay when needed.

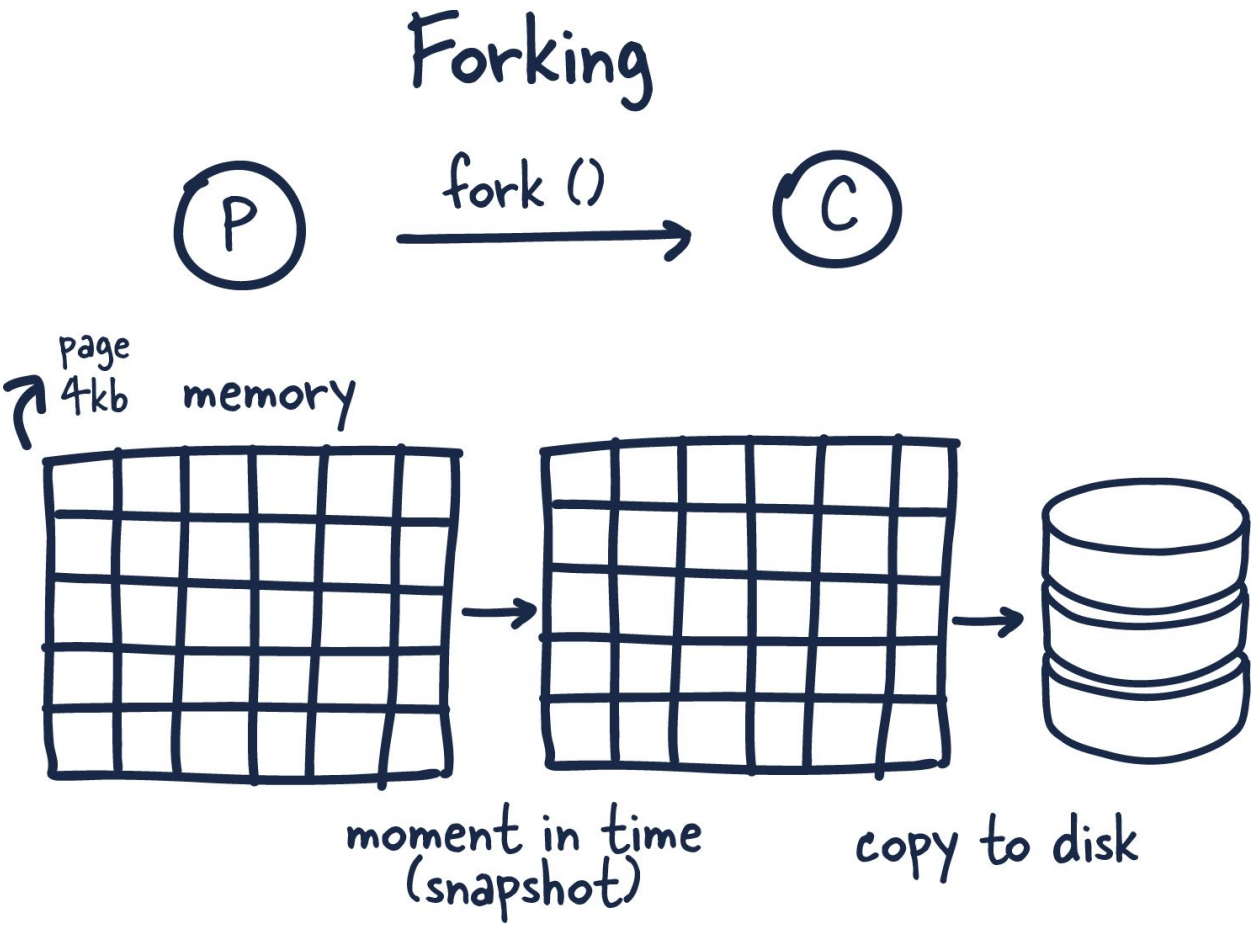
Then when possible, we flush it to disk with fsync (when this runs is configurable), it will be persisted. The downside is that the format isn't compact and uses more disk than RDB files.

Why not both?

RDB + AOF: It is possible to combine AOF and RDB in the same Redis instance. If durability in exchange for some speed is a tradeoff, you are willing to make it. I think this is an acceptable way to set up Redis. In the case of a restart, remember that if both are enabled, Redis will use AOF to reconstruct the data since it's the most complete.

Forking

Now that we understand the types of persistence, let’s discuss how we actually go about doing it in a single threaded application like Redis.



This coolest part of Redis in my opinion is how it leverages forking and copy-on-write to facilitate data persistence performantly.

Forking is a way for operating systems to create new processes by creating copies of themselves. With this, you get a new process ID and a few other bits of information and handles, so the newly forked process (child) can talk to the original process parent.

Now here is where things get interesting. Redis is a process with tons of memory allocated to it, so how does it make a copy without running out of memory?

When you fork a process, the parent and child share memory, and in that child process Redis begins the snapshotting (Redis) process. This is made possible by a memory

sharing technique called copy-on-write —which passes references to the memory at the time the fork was created. If no changes occur while the child process is persisting to disk, no new allocations are made.

In the case where there are changes, the kernel keeps track of references to each page, and if there are more than one to specific page the changes are written to new pages. The child process is fully unaware of the change and has consistent memory snapshot. Therefore only fraction of the memory is used and we are able to achieve a point in time snapshot of potentially gigabytes of memory extremely quickly and efficiently!

Possible Resis Arch

- Redis
- Redis + with persistence
- Redis Multiple Instances (master and replica(s)) fro High Availability
- Redis Multiple Instances (master and replica(s)) + Sentinal Instances / Proxy → Fault tolerance
- **Redis cluster with sharding** → for horizontal scaling
