

Grokking the Advanced System Design Interview

118 Lessons

16 Quizzes

2 Code Snippets

109 Illustrations

Course Overview

System design questions have increasingly become an integral part of software engineering interviews. For senior engineers, the discussion around system design is considered even more important than solving a coding question. In a system design interview, you can show your real design ...

See More



How You'll Learn

Faster than videos

Videos are holding you back. The average video tutorial is spoken at 150 words per minute, while you can read at 250. That's why our courses are text-based.

Progress you can show

Built in assessments let you test your skills. Completion certificates let you show them off.

Course Contents

Show All Lessons



1. Introduction



2. Dynamo: How to Design a Key-value Store?



3. Cassandra: How to Design a Wide-column NoSQL Database?



4. Kafka: How to Design a Distributed Messaging System?



5. Chubby: How to Design a Distributed Locking Service?



6. GFS: How to Design a Distributed File Storage System?



7. HDFS: How to Design a Distributed File Storage System?



8. BigTable: How to Design a Wide-column Storage System?



9. System Design Patterns



10. Final Assessment



11. Appendix

What Is This Course About?

Here is an overview of what to expect from this course.

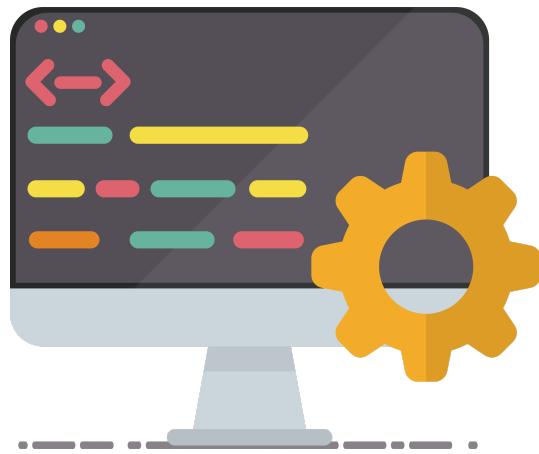
We'll cover the following



- What to expect
 - Part 1: System design case studies
 - Part 2: System design patterns

One common challenge senior engineers face is the lack of experience in designing scalable systems. The reason because it is not easy to get an opportunity to work on a large project, especially from the ground up. Most of the time, software engineers get to work on a small part of a bigger system. For these reasons, most engineers feel less prepared for system design interviews, as they lack the adequate knowledge required for designing large systems. This course is created to **help developers learn key system design skills** that will help them in interviews and in their professional careers.

One way to improve software designing skills is to understand the architecture of famous systems. This is equivalent to **learning from others** who have worked on designing large systems. In our experience, if a developer has a good understanding of the architecture of a complex system, it becomes pretty easy for them to gain knowledge from other systems even in a different domain. This is true because most system design techniques can easily be adapted and applied to other distributed systems.



One way to learn system design is to **read the technical papers of famous systems**. Unfortunately, reading a paper is generally believed hard, and keeping the system design interview in mind, we are not interested in a lot of details mentioned in the papers. This course extracts out the most relevant details about the system architecture that the creators had in mind while designing the system. Keeping system design interviews in mind, we will focus on the various tradeoffs that the original developers had considered and what prompted them to choose a certain design given their constraints.

Furthermore, systems grow over time; thus, original designs are revised, adapted, and enhanced to cater to emerging requirements. This means reading original papers is not enough. This course will cover **criticism on the original design** and the architectural changes that followed to overcome design limitations and satisfy growing needs.

What to expect

The course has two parts: **System Design Case Studies** and **System Design Patterns**.

Part 1: System design case studies

In the first part, we will go through the architecture of a carefully chosen set of distributed systems:

1. Key-value store: **Dynamo**
2. No-SQL wide column stores: **Cassandra** and **BigTable**
3. Distributed messaging and streaming system: **Kafka**
4. Distributed file storage systems: **GFS** and **HDFS**
5. Distributed coordination and locking service: **Chubby** (*similar to Zookeeper*)

Part 2: System design patterns

In the second part of this course, we will describe a set of design problems (*and their solutions*) that are common to distributed systems. We call these techniques '**System Design Patterns**', as they can be applied to all kinds of distributed systems and are very handy, especially in a system design interview. A few examples of such patterns are:

- **Write-ahead logging**
- **Bloom filters**
- **Heartbeat**
- **Quorum**
- **Checksum**
- **Lease**
- **Split Brain**

Happy learning!

Next →

Dynamo: How to Design a Key-value Store?

Dynamo: Introduction

Let's explore Dynamo and its use cases.

We'll cover the following



- Goal
- What is Dynamo?
- Background
- Design goals
- Dynamo's use cases
- System APIs

Goal

Design a **distributed key-value store** that is highly available (i.e., reliable), highly scalable, and completely decentralized.

What is Dynamo?

Dynamo is a **highly available key-value store** developed by Amazon for their internal use. Many Amazon services, such as shopping cart, bestseller lists, sales rank, product catalog, etc., need only primary-key access to data. A multi-table relational database system would be an overkill for such services and would also limit scalability and availability. Dynamo provides a flexible design to let applications choose their desired level of availability and consistency.

Background

Dynamo – not to be confused with DynamoDB, which was inspired by Dynamo's design – is a distributed key-value storage system that provides an **“always-on”** (or highly available) experience at a massive scale. In CAP theorem

(<https://www.educative.io/collection/page/5668639101419520/5559029852536832/5998984290631680>) terms, Dynamo falls within the category of **AP systems** (*i.e.*, available and partition tolerant) and is designed for **high availability and partition tolerance at the expense of strong consistency**.

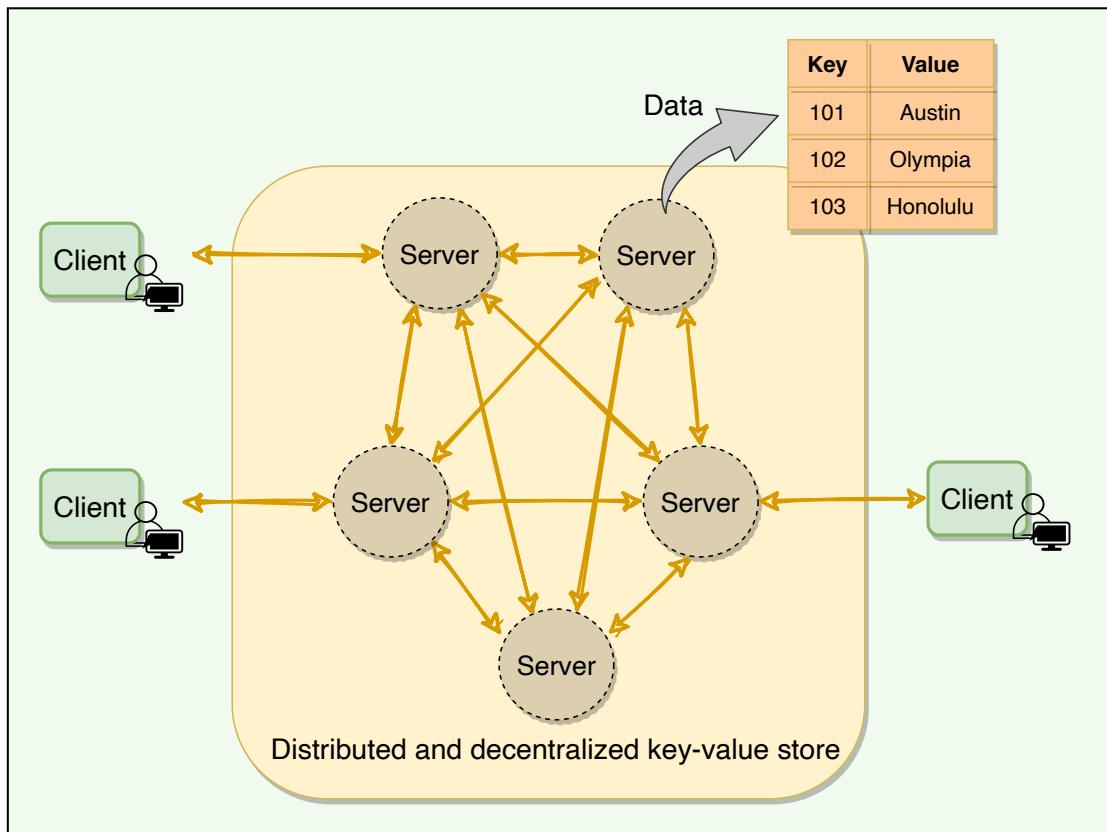
The primary motivation for designing Dynamo as a highly available system was the observation that the availability of a system directly correlates to the number of customers served. Therefore, the main goal is that the system, even when it is imperfect, should be available to the customer as it brings more customer satisfaction. On the other hand, inconsistencies can be resolved in the background, and most of the time they will not be noticeable by the customer. Derived from this core principle, Dynamo is aggressively optimized for availability.

The Dynamo design was highly influential as it inspired many NoSQL databases, like Cassandra (<https://cassandra.apache.org/>), Riak (<https://riak.com/>), and Voldemort (<http://www.project-voldemort.com/voldemort/>) – not to mention Amazon's own DynamoDB (<https://aws.amazon.com/dynamodb/>).

Design goals

As stated above, the main goal of Dynamo is to be **highly available**. Here is the summary of its other design goals:

- **Scalable:** The system should be *highly scalable*. We should be able to throw a machine into the system to see proportional improvement.
- **Decentralized:** To avoid single points of failure and performance bottlenecks, there should not be any central/leader process.
- **Eventually Consistent:** Data can be *optimistically replicated* to become eventually consistent. This means that instead of incurring write-time costs to ensure data correctness throughout the system (*i.e.*, *strong consistency*), inconsistencies can be resolved at some other time (*e.g.*, *during reads*). Eventual consistency is used to achieve high availability.



High-level view of a distributed key-value store

Dynamo's use cases

By default, **Dynamo** is an eventually consistent database. Therefore, any application where strong consistency is not a concern can utilize Dynamo. Though Dynamo can support strong consistency, it comes with a performance impact. Hence, if strong consistency is a requirement for an application, then Dynamo might not be a good option.

Dynamo is used at Amazon to manage services that have very high-reliability requirements and need tight control over the trade-offs between **availability, consistency, cost-effectiveness, and performance**. Amazon's platform has a very diverse set of applications with different storage requirements. Many applications chose Dynamo because of its flexibility for selecting the appropriate trade-offs to achieve high availability and guaranteed performance in the most cost-effective manner.

Many services on Amazon's platform require only primary-key access to a data store. For such services, the common pattern of using a relational database would lead to inefficiencies and limit scalability and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

System APIs

The Dynamo clients use `put()` and `get()` operations to write and read data corresponding to a specified key. This key uniquely identifies an object.

- **get(key)** : The `get` operation finds the nodes where the object associated with the given `key` is located and returns either a single object or a list of objects with conflicting versions along with a `context`. The `context` contains encoded metadata about the object that is meaningless to the caller and includes information such as the version of the object (more on this below).

- **put(key, context, object)**: The put operation finds the nodes where the object associated with the given key should be stored and writes the given object to the disk. The context is a value that is returned with a get operation and then sent back with the put operation. The context is always stored along with the object and is used like a cookie to verify the validity of the object supplied in the put request.

Dynamo treats both the object and the key as an arbitrary array of bytes (*typically less than 1 MB*). It applies the MD5 hashing algorithm on the key to generate a 128-bit identifier which is used to determine the storage nodes that are responsible for serving the key.

[!\[\]\(d219eb33a83c47f5c6c63c27bbe267cb_img.jpg\) Back](#)

[!\[\]\(6e934896f25e6ce1b0dbb50c23abc197_img.jpg\) Next](#)

What Is This Course About?

High-level Architecture

High-level Architecture

This lesson gives a brief overview of Dynamo's architecture.

We'll cover the following



- Introduction: Dynamo's architecture
 - Data distribution
 - Data replication and consistency
 - Handling temporary failures
 - Inter-node communication and failure detection
 - High availability
 - Conflict resolution and handling permanent failures

At a high level, Dynamo is a **Distributed Hash Table (DHT)** that is replicated across the cluster for high availability and fault tolerance.

Introduction: Dynamo's architecture

Dynamo's architecture can be summarized as follows (we will discuss all of these concepts in detail in the following lessons):

Data distribution

Dynamo uses **Consistent Hashing** to distribute its data among nodes. Consistent hashing also makes it easy to add or remove nodes from a Dynamo cluster.

Data replication and consistency

Data is replicated optimistically, i.e., Dynamo provides eventual consistency.

Handling temporary failures

To handle temporary failures, Dynamo replicates data to a **sloppy quorum** of other nodes in the system instead of a strict majority quorum.

Inter-node communication and failure detection

Dynamo's nodes use **gossip protocol** to keep track of the cluster state.

High availability

Dynamo makes the system “always writeable” (*or highly available*) by using **hinted handoff**.

Conflict resolution and handling permanent failures

Since there are no write-time guarantees that nodes agree on values, Dynamo resolves potential conflicts using other mechanisms:

- Use **vector clocks** to keep track of value history and reconcile divergent histories at read time.
- In the background, dynamo uses an anti-entropy mechanism like **Merkle trees** to handle permanent failures.

Let's discuss each of these concepts one by one.

← Back

Next →

Dynamo: Introduction

Data Partitioning

Data Partitioning

Let's learn how Dynamo distributes its data across a set of nodes.

We'll cover the following



- What is data partitioning?
- Consistent hashing: Dynamo's data distribution
- Virtual nodes
- Advantages of Vnodes

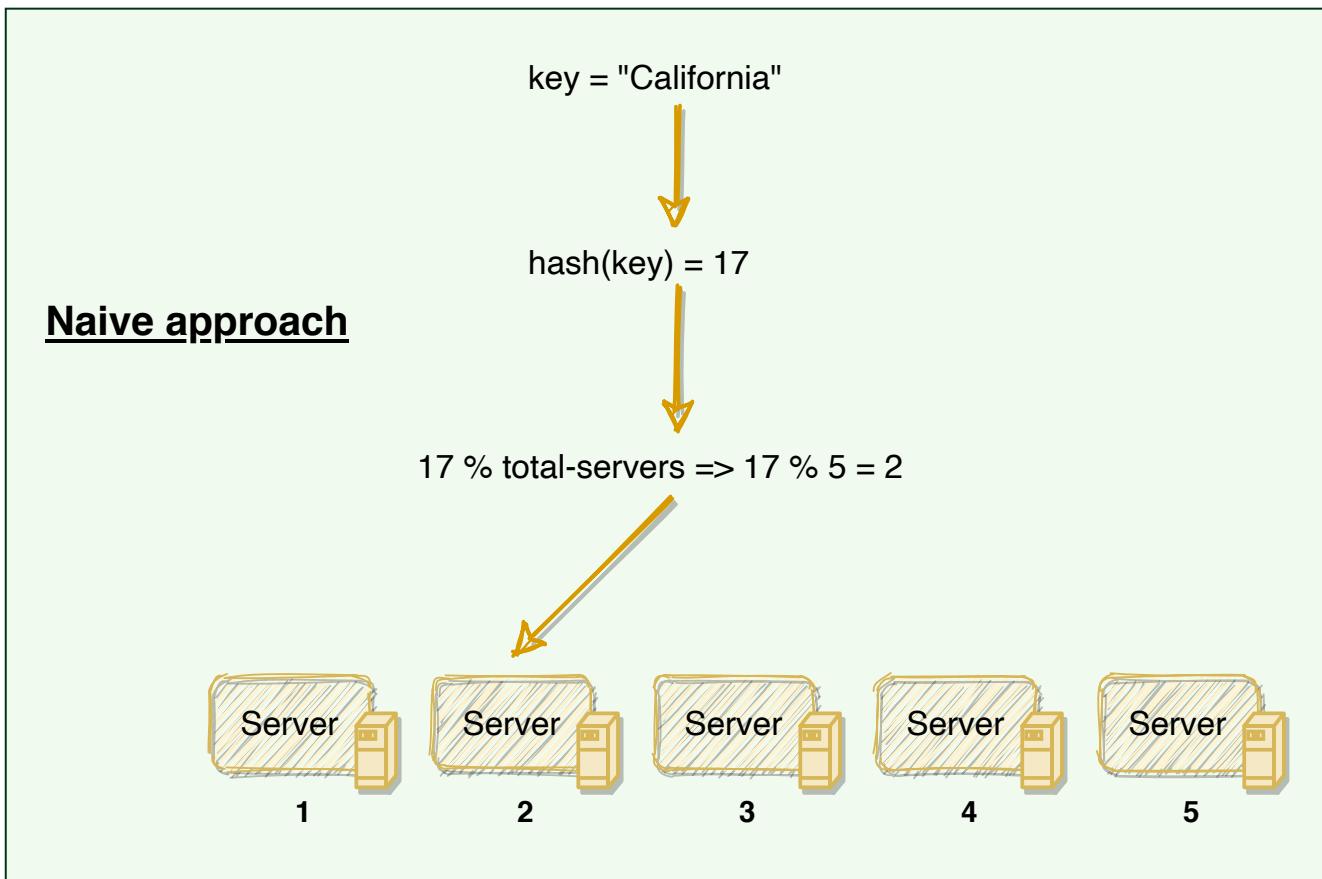
What is data partitioning?

The act of distributing data across a set of nodes is called data partitioning.

There are two challenges when we try to distribute data:

1. How do we know on which node a particular piece of data will be stored?
2. When we add or remove nodes, how do we know what data will be moved from existing nodes to the new nodes? Furthermore, how can we minimize data movement when nodes join or leave?

A naive approach will be to use a suitable hash function that maps the data key to a number. Then, find the server by applying modulo on this number and the total number of servers. For example:



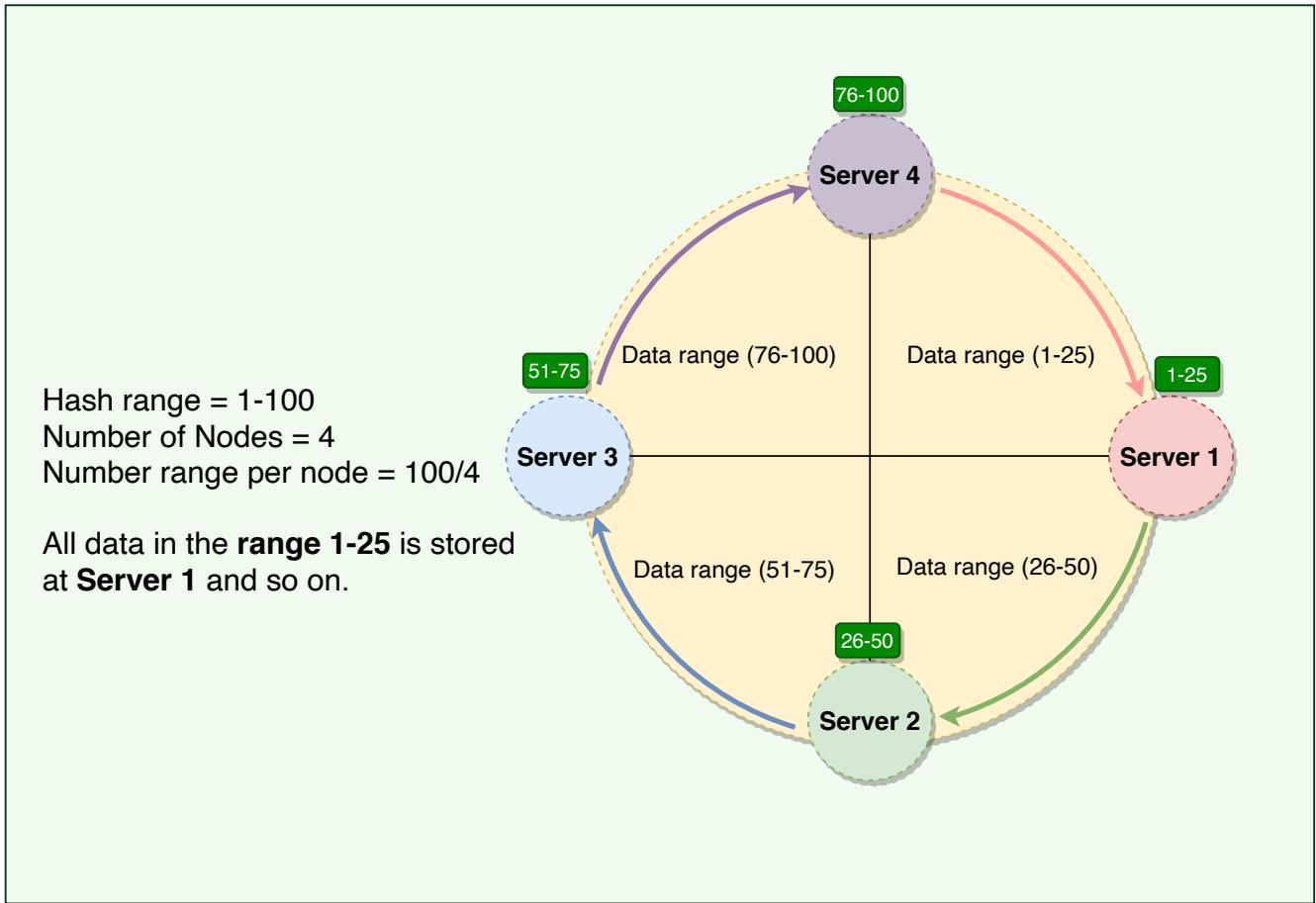
Data partitioning through simple hashing

The scheme described in the above diagram solves the problem of finding a server for storing/retrieving the data. But when we add or remove a server, we have to remap all the keys and move the data based on the new server count, which will be a complete mess!

Dynamo uses **consistent hashing** to solve these problems. The consistent hashing algorithm helps Dynamo map rows to physical nodes and also ensures that **only a small set of keys move when servers are added or removed**.

Consistent hashing: Dynamo's data distribution

Consistent hashing represents the data managed by a cluster as a ring. Each node in the ring is assigned a range of data. Dynamo uses the consistent hashing algorithm to determine what row is stored to what node. Here is an example of the consistent hashing ring:



Consistent Hashing ring

With consistent hashing, the ring is divided into smaller predefined ranges. Each node is assigned one of these ranges. In Dynamo's terminology, the start of the range is called a **token**. This means that each node will be assigned one token. The range assigned to each node is computed as follows:

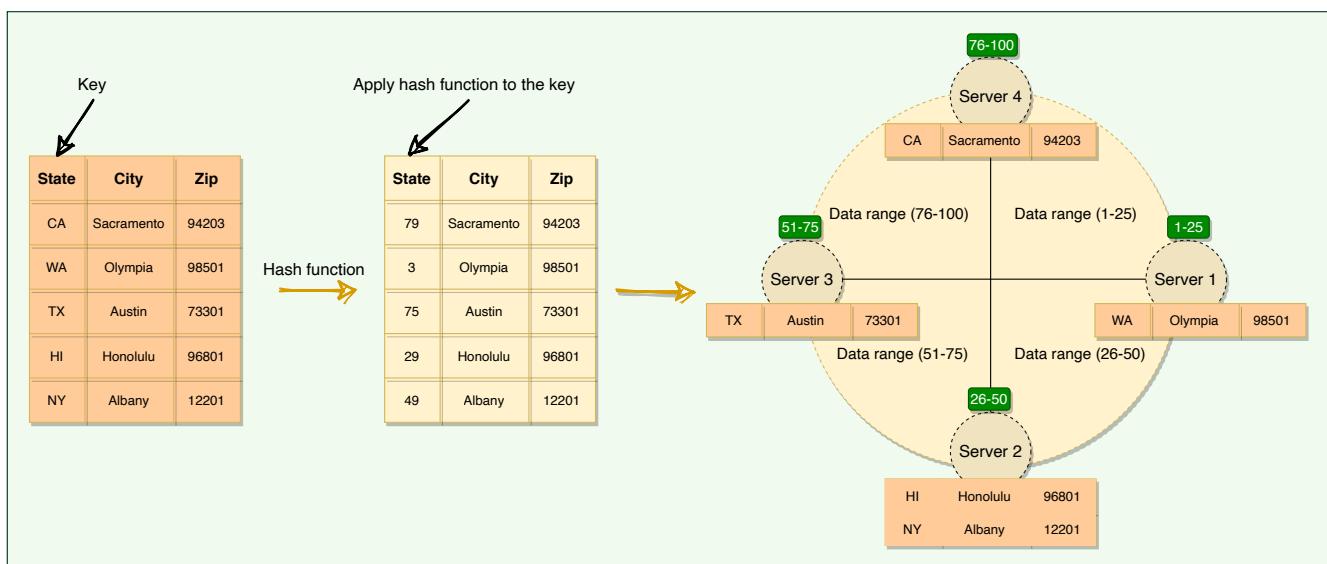
Range start: Token value

Range end: Next token value - 1

Here are the tokens and data ranges of the four nodes described in the above diagram:

Server	Token	Range Start	Range End
Server 1	1	1	25
Server 2	26	26	50
Server 3	51	51	75
Server 4	76	76	100

Whenever Dynamo is serving a `put()` or a `get()` request, the first step it performs is to apply the MD5 hashing algorithm to the key. The output of this hashing algorithm determines within which range the data lies and hence, on which node the data will be stored. As we saw above, each node in Dynamo is supposed to store data for a fixed range. Hence, the hash generated from the data key tells us the node where the data will be stored. Here is an example showing how data gets distributed across the Consistent Hashing ring:



Distributing data on the consistent hashing ring

The consistent hashing scheme described above works great when a node is added or removed from the ring; as only the next node is affected in these scenarios. For example, when a node is removed, the next node becomes responsible for all of the keys stored on the outgoing node. However, this scheme can result in non-uniform data and load distribution. Dynamo solves these issues with the help of Virtual nodes.

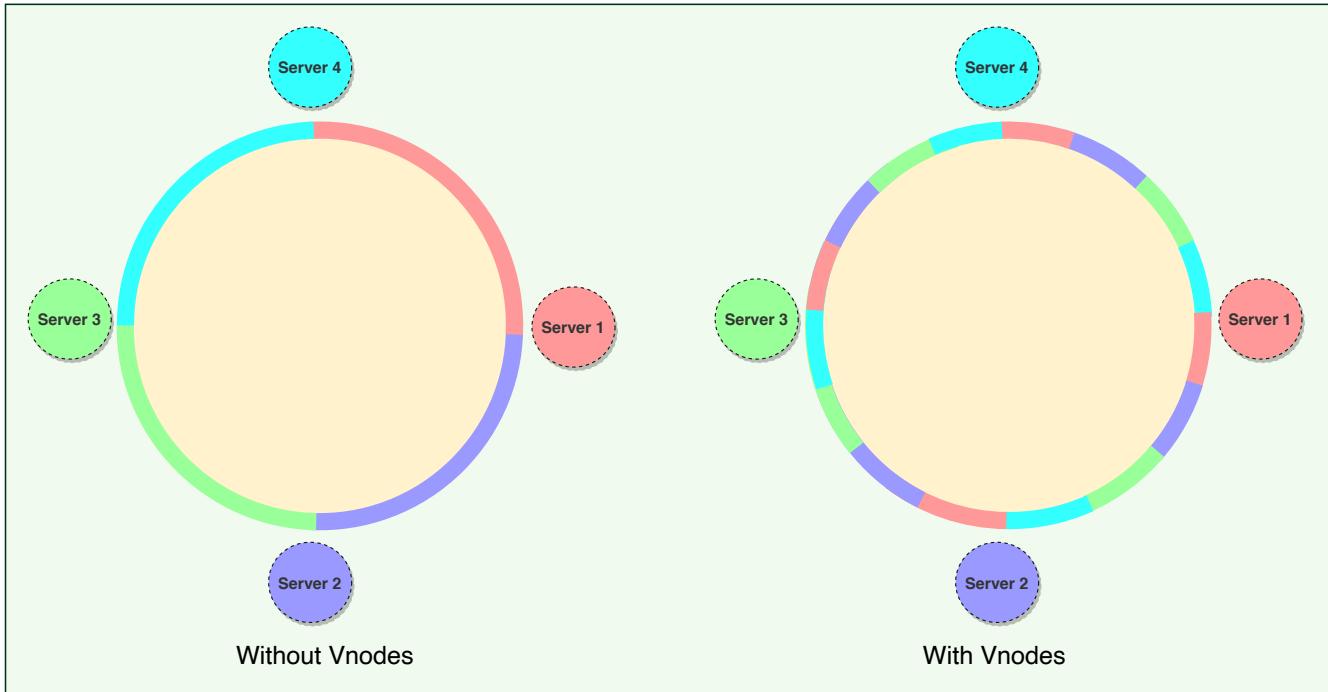
Virtual nodes

Adding and removing nodes in any distributed system is quite common. Existing nodes can die and may need to be decommissioned. Similarly, new nodes may be added to an existing cluster to meet growing demands. Dynamo efficiently handles these scenarios through the use of virtual nodes (or *Vnodes*).

As we saw above, the basic Consistent Hashing algorithm assigns a single token (or a consecutive hash range) to each physical node. This was a static division of ranges that requires calculating tokens based on a given number of nodes. This scheme made adding or replacing a node an expensive operation, as, in this case, we would like to rebalance and distribute the data to all other nodes, resulting in moving a lot of data. Here are a few potential issues associated with a manual and fixed division of the ranges:

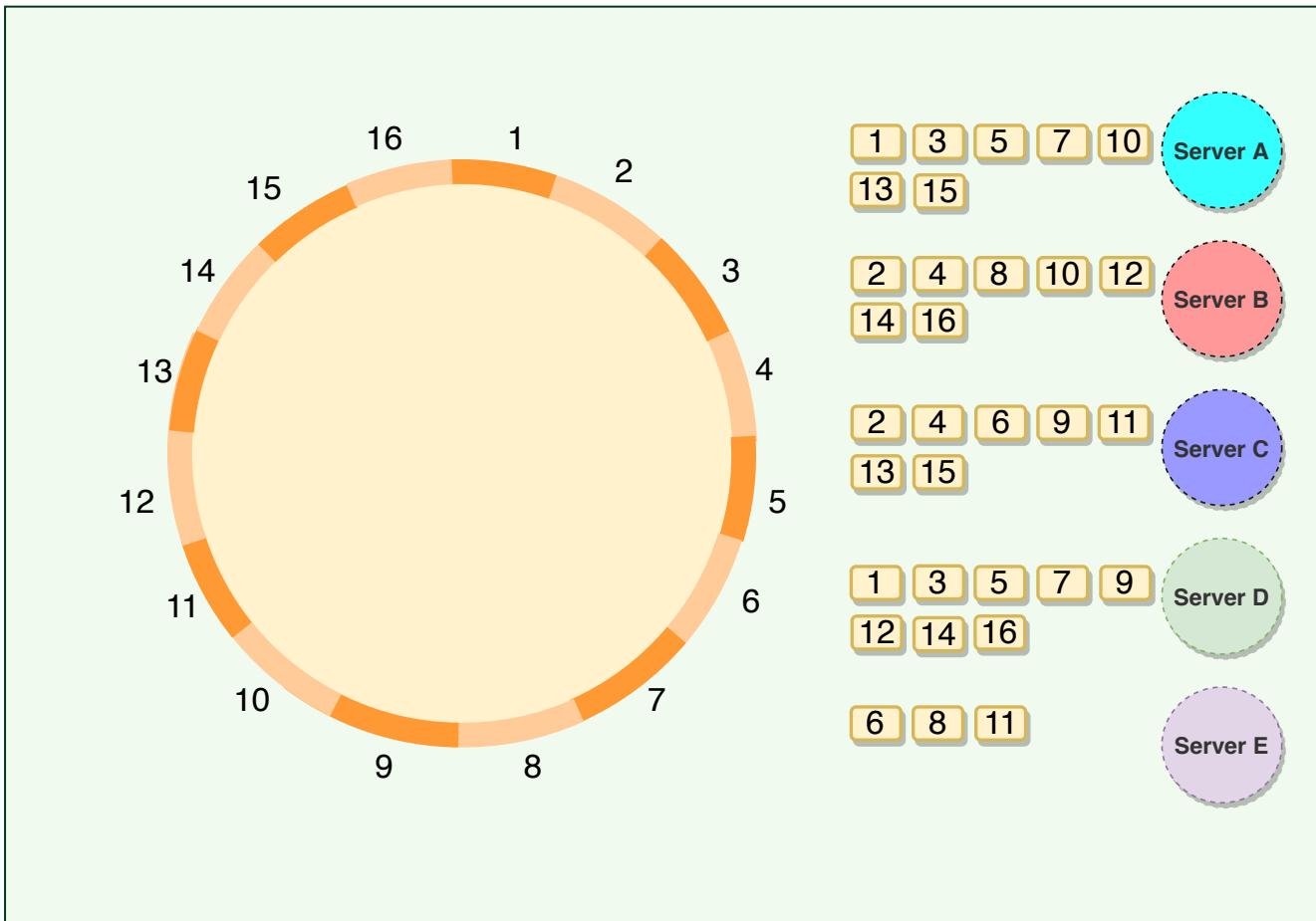
- **Adding or removing nodes:** Adding or removing nodes will result in recomputing the tokens causing a significant administrative overhead for a large cluster.
- **Hotspots:** Since each node is assigned one large range, if the data is not evenly distributed, some nodes can become hotspots.
- **Node rebuilding:** Since each node's data is replicated on a fixed number of nodes (*discussed later*), when we need to rebuild a node, only its replica nodes can provide the data. This puts a lot of pressure on the replica nodes and can lead to service degradation.

To handle these issues, Dynamo introduced a new scheme for distributing the tokens to physical nodes. Instead of assigning a single token to a node, the hash range is divided into multiple smaller ranges, and each physical node is assigned multiple of these smaller ranges. Each of these subranges is called a Vnode. With Vnodes, instead of a node being responsible for just one token, it is responsible for many tokens (or subranges).



Comparing Consistent Hashing ring with and without Vnodes

Practically, Vnodes are **randomly distributed** across the cluster and are generally **non-contiguous** so that no two neighboring Vnodes are assigned to the same physical node. Furthermore, nodes do carry replicas of other nodes for fault-tolerance. Also, since there can be heterogeneous machines in the clusters, some servers might hold more Vnodes than others. The figure below shows how physical nodes A, B, C, D, & E are using Vnodes of the Consistent Hash ring. Each physical node is assigned a set of Vnodes and each Vnode is replicated once.



Mapping Vnodes to physical nodes on a Consistent Hashing ring

Advantages of Vnodes

Vnodes give the following advantages:

1. Vnodes help **spread the load more evenly** across the physical nodes on the cluster by dividing the hash ranges into smaller subranges. This speeds up the rebalancing process after adding or removing nodes. When a new node is added, it receives many Vnodes from the existing nodes to maintain a balanced cluster. Similarly, when a node needs to be rebuilt, instead of getting data from a fixed number of replicas, many nodes participate in the rebuild process.
2. Vnodes make it easier to **maintain a cluster containing heterogeneous machines**. This means, with Vnodes, we can assign a

high number of ranges to a powerful server and a lower number of ranges to a less powerful server.

3. Since Vnodes help assign smaller ranges to each physical node, the **probability of hotspots is much less** than the basic Consistent Hashing scheme which uses one big range per node.

[!\[\]\(0a023d01ac3b7c728c29528b0758e35e_img.jpg\) Back](#)

High-level Architecture

[!\[\]\(004d352ca3e5c974252147a5c78e6fbb_img.jpg\) Next](#) 

Replication

Replication

Let's learn how Dynamo replicates its data and handles temporary failures through replication.

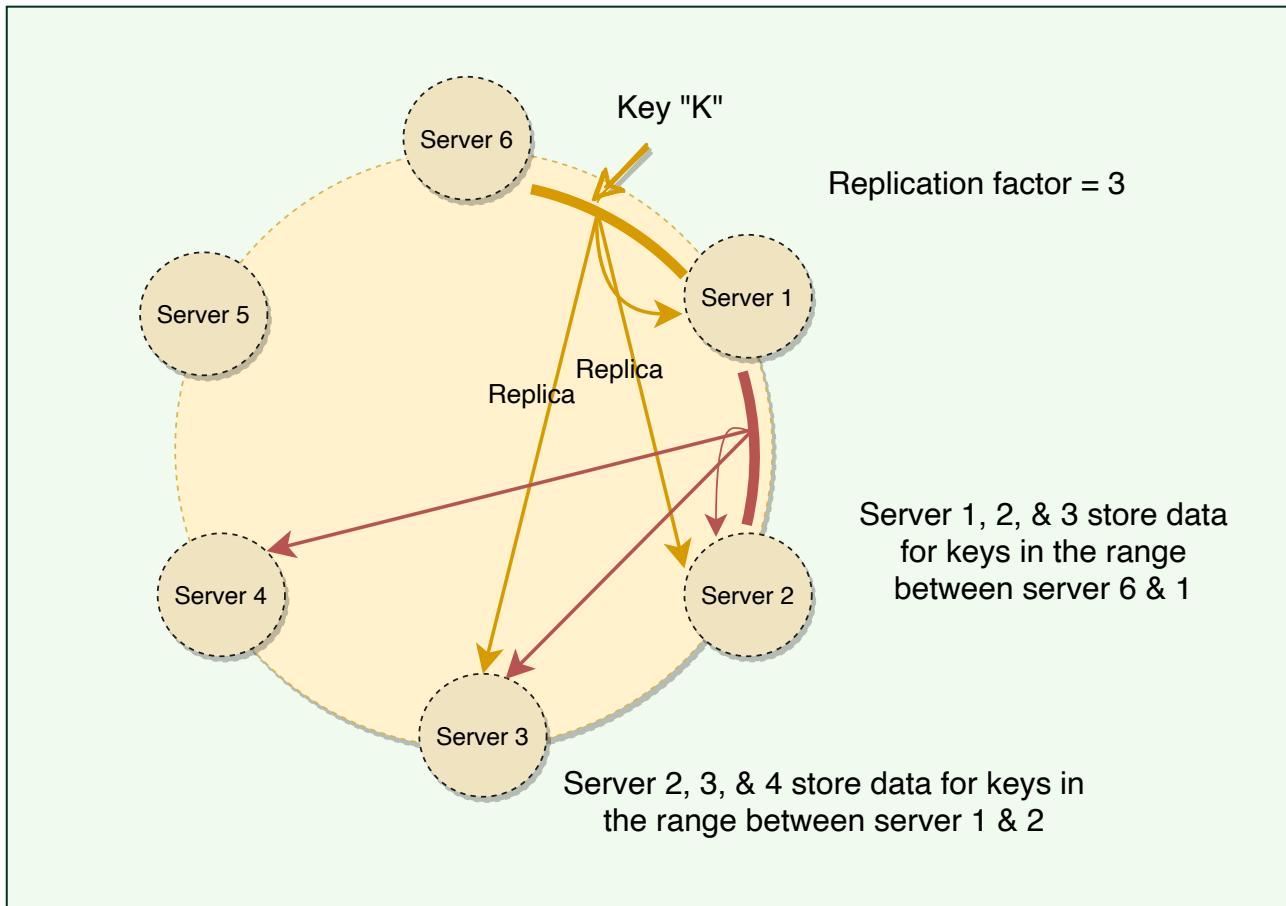
We'll cover the following



- What is optimistic replication?
- Preference List
- Sloppy quorum and handling of temporary failures
- Hinted handoff

What is optimistic replication?

To ensure high availability and durability, Dynamo replicates each data item on multiple N nodes in the system where the value N is equivalent to the replication factor and is configurable per instance of Dynamo. Each key is assigned to a **coordinator node** (*the node that falls first in the hash range*), which first stores the data locally and then replicates it to $N - 1$ clockwise successor nodes on the ring. This results in each node owning the region on the ring between it and its N th predecessor. This replication is done asynchronously (*in the background*), and Dynamo provides an **eventually consistent** model. This replication technique is called **optimistic replication**, which means that replicas are not guaranteed to be identical at all times.



Replication in consistent hashing

Each node in Dynamo serves as a replica for a different range of data. As Dynamo stores N copies of data spread across different nodes, if one node is down, other replicas can respond to queries for that range of data. If a client cannot contact the coordinator node, it sends the request to a node holding a replica.

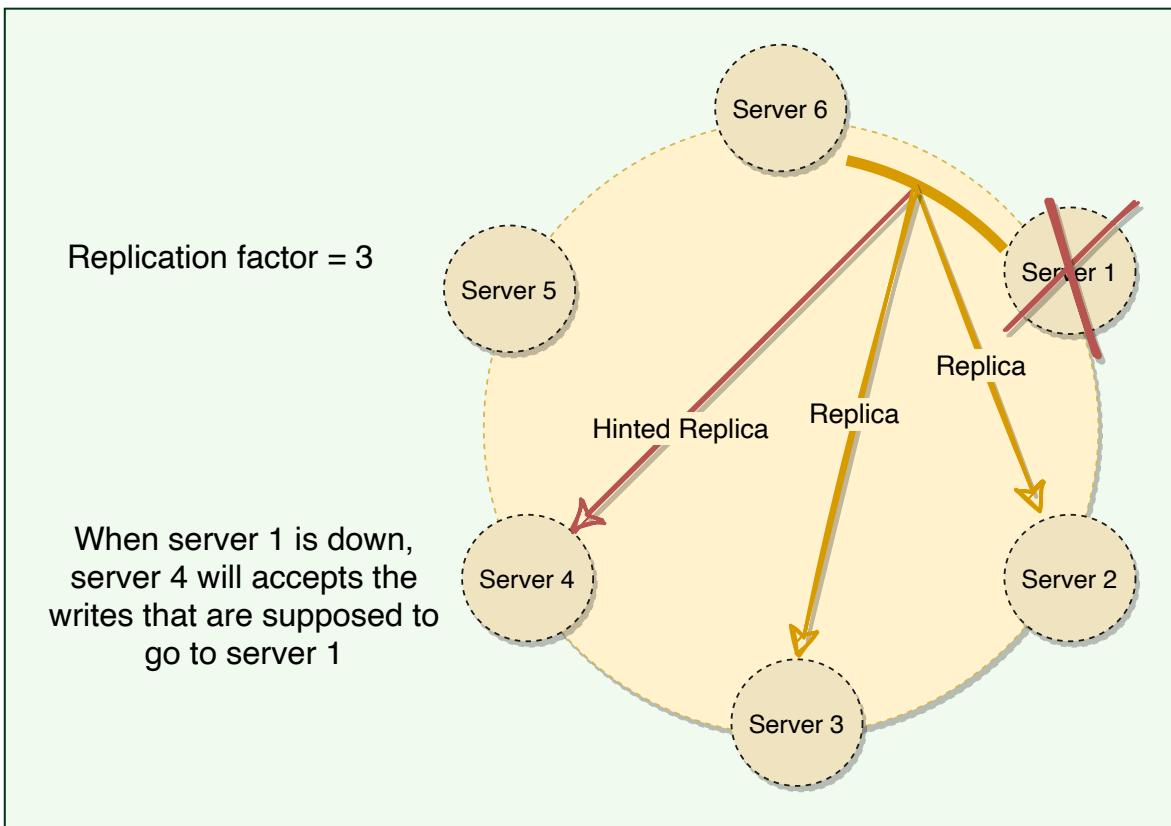
Preference List

The list of nodes responsible for storing a particular key is called the preference list. Dynamo is designed so that every node in the system can determine which nodes should be in this list for any specific key (discussed later). This list contains more than N nodes to account for failure and skip virtual nodes on the ring so that the list only contains distinct physical nodes.

Sloppy quorum and handling of temporary failures

Following traditional quorum approaches, any distributed system becomes unavailable during server failures or network partitions and would have reduced availability even under simple failure conditions. To increase the availability, Dynamo does not enforce strict quorum requirements, and instead uses something called **sloppy quorum**. With this approach, all read/write operations are performed on the first N healthy nodes from the preference list, which may not always be the first N nodes encountered while moving clockwise on the consistent hashing ring.

Consider the example of Dynamo configuration given in the figure below with $N = 3$. In this example, if Server 1 is temporarily down or unreachable during a write operation, its data will now be stored on Server 4. Thus, Dynamo transfers the replica stored on the failing node (*i.e.*, Server 1) to the next node of the consistent hash ring that does not have the replica (*i.e.*, Server 4). This is done to avoid unavailability caused by a short-term machine or network failure and to maintain desired availability and durability guarantees. The replica sent to Server 4 will have a hint in its metadata that suggests which node was the intended recipient of the replica (*in this case*, Server 1). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that Server 1 has recovered, Server 4 will attempt to deliver the replica to Server 1. Once the transfer succeeds, Server 4 may delete the object from its local store without decreasing the total number of replicas in the system.



Sloppy quorum

Hinted handoff

The interesting trick described above to increase availability is known as hinted handoff, i.e., **when a node is unreachable, another node can accept writes on its behalf**. The write is then kept in a local buffer and sent out once the destination node is reachable again. This makes Dynamo “**always writeable**.” Thus, even in the extreme case where only a single node is alive, write requests will still get accepted and eventually processed.

The main problem is that since a sloppy quorum is not a strict majority, the data can and will diverge, i.e., it is possible for two concurrent writes to the same key to be accepted by non-overlapping sets of nodes. This means that multiple conflicting values against the same key can exist in the system, and we can get stale or conflicting data while reading. Dynamo allows this and resolves these conflicts using Vector Clocks.

← Back

Next →

Data Partitioning

Vector Clocks and Conflicting Data

Vector Clocks and Conflicting Data

Let's learn how Dynamo uses vector clocks to keep track of data history and reconcile divergent histories at read time.

We'll cover the following



- What is clock skew?
- What is a vector clock?
- Conflict-free replicated data types (CRDTs)
- Last-write-wins (LWW)

As described in the previous lesson

(<https://www.educative.io/collection/page/5668639101419520/5559029852536832/4935298205614080>), sloppy quorum means multiple conflicting values against the same key can exist in the system and must be resolved somehow. Let's understand how this can happen.

What is clock skew?

On a single machine, all we need to know about is the absolute or **wall clock** time: suppose we perform a write to key k with timestamp t_1 and then perform another write to k with timestamp t_2 . Since $t_2 > t_1$, the second write must have been newer than the first write, and therefore the database can safely overwrite the original value.

In a distributed system, this assumption does not hold. The problem is **clock skew**, i.e., different clocks tend to run at different rates, so we cannot assume that time t on node a happened before time $t + 1$ on node b . The

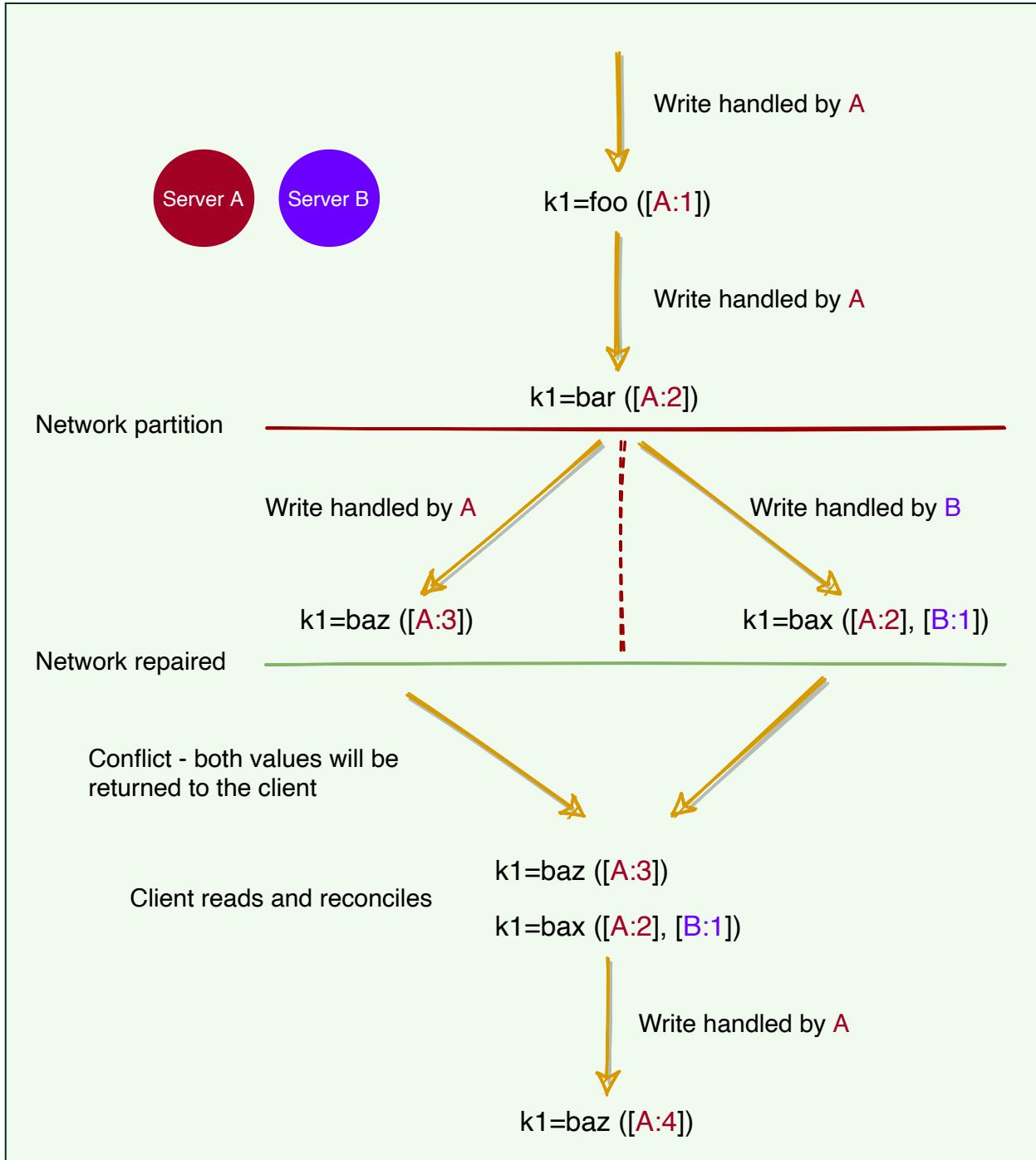
most practical techniques that help with synchronizing clocks, like NTP, still do not guarantee that every clock in a distributed system is synchronized at all times. So, without special hardware like GPS units and atomic clocks, just using wall clock timestamps is not enough.

What is a vector clock?

Instead of employing tight synchronization mechanics, Dynamo uses something called **vector clock** in order to **capture causality between different versions of the same object**. A vector clock is effectively a **(node, counter)** pair. One vector clock is associated with every version of every object stored in Dynamo. One can determine whether two versions of an object are on parallel branches or have a causal ordering by examining their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation. Dynamo resolves these conflicts at read-time. Let's understand this with an example:

1. Server A serves a write to key `k1`, with value `foo`. It assigns it a version of `[A:1]`. This write gets replicated to server B.
2. Server A serves a write to key `k1`, with value `bar`. It assigns it a version of `[A:2]`. This write also gets replicated to server B.
3. A network partition occurs. A and B cannot talk to each other.
4. Server A serves a write to key `k1`, with value `baz`. It assigns it a version of `[A:3]`. It cannot replicate it to server B, but it gets stored in a hinted handoff buffer on another server.
5. Server B sees a write to key `k1`, with value `bax`. It assigns it a version of `[B:1]`. It cannot replicate it to server A, but it gets stored in a hinted handoff buffer on another server.
6. The network heals. Server A and B can talk to each other again.

7. Either server gets a read request for key `k1`. It sees the same key with different versions `[A:3]` and `[A:2] [B:1]`, but it does not know which one is newer. It returns both and tells the client to figure out the version and write the newer version back into the system.



Conflict resolution using vector clocks

As we saw in the above example, most of the time, new versions subsume the previous version(s), and the system itself can determine the correct version (e.g., [A:2] is newer than [A:1]). However, version branching may happen in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object, and the client must perform the reconciliation to collapse multiple branches of data evolution back into one (*this process is called semantic reconciliation*). A typical example of a collapse operation is “merging” different versions of a customer’s shopping cart. Using this reconciliation mechanism, an add operation (i.e., adding an item to the cart) is never lost. However, deleted items can resurface.

Resolving conflicts is similar to how Git works. If Git can merge different versions into one, merging is done automatically. If not, the client (i.e., the developer) has to reconcile conflicts manually.

Dynamo truncates vector clocks (*oldest first*) when they grow too large. If Dynamo ends up deleting older vector clocks that are required to reconcile an object’s state, Dynamo would not be able to achieve eventual consistency. Dynamo’s authors note that this is a potential problem but do not specify how this may be addressed. They do mention that this problem has not yet surfaced in any of their production systems.

Conflict-free replicated data types (CRDTs)

A more straightforward way to handle conflicts is through the use of **CRDTs**. To make use of CRDTs, we need to model our data in such a way that concurrent changes can be applied to the data in any order and will produce the same end result. This way, the system does not need to worry about any ordering guarantees. Amazon's shopping cart is an excellent example of CRDT. When a user adds two items (A & B) to the cart, these two operations of adding A & B can be done on any node and with any order, as the end result is the two items are added to the cart. (*Removing from the shopping cart is modeled as a negative add.*) The idea that any two nodes that have received the same set of updates will see the same end result is called **strong eventual consistency**. Riak has a few built-in CRDTs (<https://docs.riak.com/riak/kv/2.2.0/developing/data-types/>).

Last-write-wins (LWW)

Unfortunately, it is not easy to model the data as CRDTs. In many cases, it involves too much effort. Therefore, vector clocks with client-side resolution are considered good enough.

Instead of vector clocks, Dynamo also offers ways to resolve the conflicts automatically on the server-side. Dynamo (*and Apache Cassandra*) often uses a simple conflict resolution policy: **last-write-wins (LWW)**, based on the wall-clock timestamp. LWW can easily end up losing data. For example, if two conflicting writes happen simultaneously, it is equivalent to flipping a coin on which write to throw away.

← Back

Next →

The Life of Dynamo's put() & get() Operations

Let's learn how Dynamo handles get() and put() requests.

We'll cover the following



- Strategies for choosing the coordinator node
- Consistency protocol
- 'put()' process
- 'get()' process
- Request handling through state machine

Strategies for choosing the coordinator node

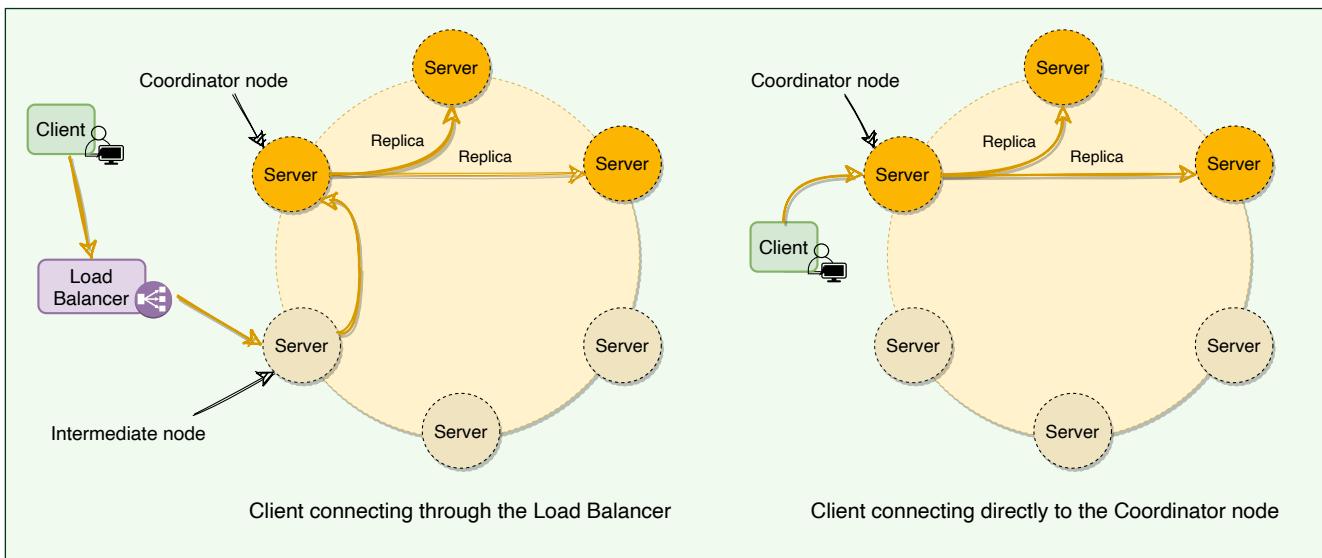
Dynamo clients can use one of the two strategies to choose a node for their `get()` and `put()` requests:

- Clients can route their requests through a generic load balancer.
- Clients can use a partition-aware client library that routes the requests to the appropriate coordinator nodes with lower latency.

In the first case, the load balancer decides which way the request would be routed, while in the second strategy, the client selects the node to contact. Both approaches are beneficial in their own ways.

In the first strategy, the client is unaware of the Dynamo ring, which helps scalability and makes Dynamo's architecture loosely coupled. However, in this case, since the load balancer can forward the request to any node in the ring, it is possible that the node it selects is not part of the preference list. This will result in an extra hop, as the request will then be forwarded to one of the nodes in the preference list by the intermediate node.

The second strategy helps in achieving lower latency, as in this case, the client maintains a copy of the ring and forwards the request to an appropriate node from the preference list. Because of this option, Dynamo is also called a **zero-hop DHT**, as the client can directly contact the node that holds the required data. However, in this case, Dynamo does not have much control over the load distribution and request handling.



How clients connect to Dynamo

Consistency protocol

Dynamo uses a consistency protocol similar to quorum systems. If R/W is the minimum number of nodes that must participate in a successful read/write operation respectively:

- Then $R + W > N$ yields a quorum-like system
- A Common (N, R, W) configuration used by Dynamo is $(3, 2, 2)$.
 - $(3, 3, 1)$: fast W , slow R , not very durable
 - $(3, 1, 3)$: fast R , slow W , durable
- In this model, the latency of a `get()` (or `put()`) operation depends upon the slowest of the replicas. For this reason, R and W are usually configured to be less than N to provide better latency.
- In general, low values of W and R increase the risk of inconsistency, as write requests are deemed successful and returned to the clients even if a majority of replicas have not processed them. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.
- For both Read and Write operations, the requests are forwarded to the first ' N ' healthy nodes.

‘put()’ process

Dynamo’s `put()` request will go through the following steps:

1. The coordinator generates a new data version and vector clock component.
2. Saves new data locally.
3. Sends the write request to $N - 1$ highest-ranked healthy nodes from the preference list.
4. The `put()` operation is considered successful after receiving $W - 1$ confirmation.

‘get()’ process

Dynamo's `get()` request will go through the following steps:

1. The coordinator requests the data version from $N - 1$ highest-ranked healthy nodes from the preference list.
2. Waits until $R - 1$ replies.
3. Coordinator handles causal data versions through a vector clock.
4. Returns all relevant data versions to the caller.

Request handling through state machine

Each client request results in creating a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies, and packaging the response for the client. Each state machine instance handles exactly one client request. For example, a read operation implements the following state machine:

1. Send read requests to the nodes.
2. Wait for the minimum number of required responses.
3. If too few replies were received within a given time limit, fail the request.
4. Otherwise, gather all the data versions and determine the ones to be returned.
5. If versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions.

After the read response has been returned to the caller, the state machine waits for a short period to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called **Read Repair** because it repairs replicas that have missed a recent update.

As stated above, `put()` requests are coordinated by one of the top N nodes in the preference list. Although it is always desirable to have the first node among the top N to coordinate the writes, thereby serializing all writes at a single location, this approach has led to uneven load distribution for Dynamo. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write operation usually follows a read operation, the coordinator for a write operation is chosen to be the node that replied fastest to the previous read operation, which is stored in the request's context information. This optimization enables Dynamo to pick the node that has the data that was read by the preceding read operation, thereby increasing the chances of getting “**read-your-writes**” consistency.

[← Back](#)

[Next →](#)

Vector Clocks and Conflicting Data

Anti-entropy Through Merkle Trees

Anti-entropy Through Merkle Trees

Let's understand how Dynamo uses Merkle trees for anti-entropy operations.

We'll cover the following

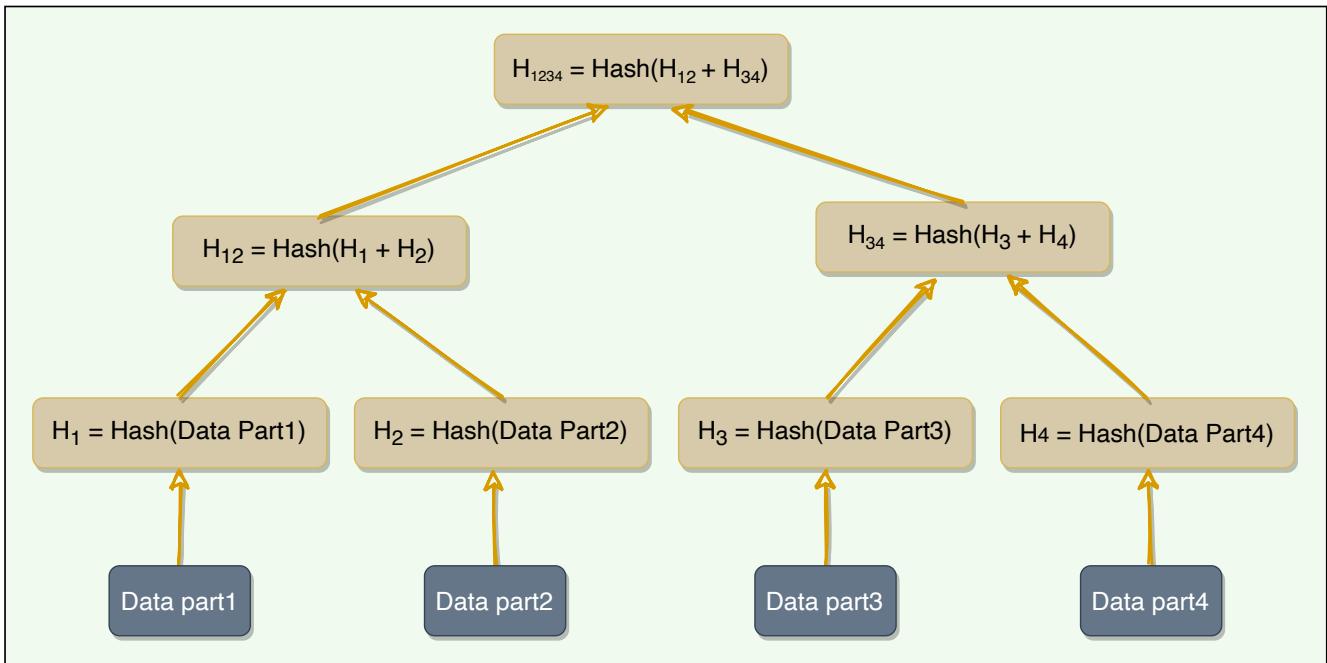


- What are Merkle trees?
- Merits and demerits of Merkle trees

As we know, Dynamo uses vector clocks to remove conflicts while serving read requests. Now, if a replica falls significantly behind others, it might take a very long time to resolve conflicts using just vector clocks. It would be nice to be able to automatically resolve some conflicts in the background. To do this, we need to quickly compare two copies of a range of data residing on different replicas and figure out exactly which parts are different.

What are Merkle trees?

A replica can contain a lot of data. Naively splitting up the entire data range for checksums is not very feasible; there is simply too much data to be transferred. Therefore, Dynamo uses **Merkle trees** to compare replicas of a range. A Merkle tree is a binary tree of hashes, where each internal node is the hash of its two children, and each leaf node is a hash of a portion of the original data.



Merkle tree

Comparing Merkle trees is conceptually simple:

1. Compare the root hashes of both trees.
2. If they are equal, stop.
3. Recurse on the left and right children.

Ultimately, this means that replicas know precisely which parts of the range are different, and the amount of data exchanged is minimized.

Merits and demerits of Merkle trees

The principal advantage of using a Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the whole data set. Hence, Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

The disadvantage of using Merkle trees is that many key ranges can change when a node joins or leaves, and as a result, the trees need to be recalculated.

 Back

Next 

The Life of Dynamo's put() & get() Op...

Gossip Protocol

Gossip Protocol

Let's explore how Dynamo uses gossip protocol to keep track of the cluster state.

We'll cover the following



- What is gossip protocol?
- External discovery through seed nodes

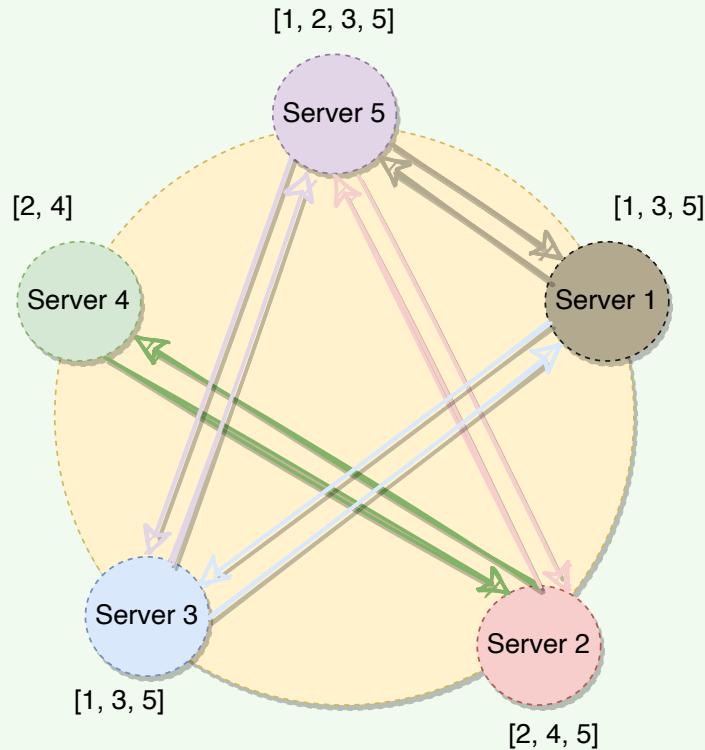
What is gossip protocol?

In a Dynamo cluster, since we do not have any central node that keeps track of all nodes to know if a node is down or not, how does a node know every other node's current state? The simplest way to do this is to have every node maintain heartbeats with every other node. When a node goes down, it will stop sending out heartbeats, and everyone else will find out immediately. But then $O(N^2)$ messages get sent every tick (*N being the number of nodes*), which is a ridiculously high amount and not feasible in any sizable cluster.

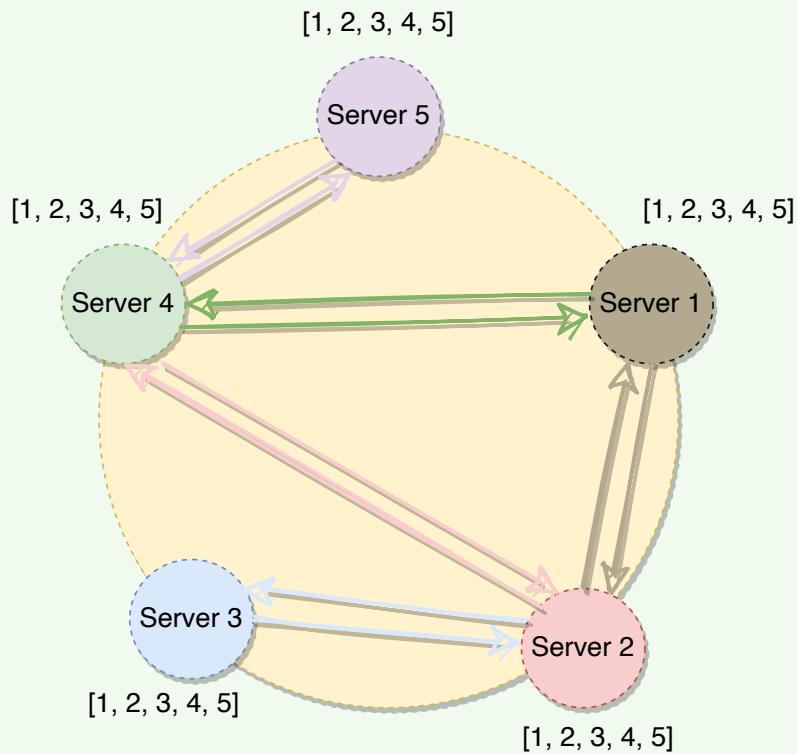
Dynamo uses **gossip protocol** that enables each node to keep track of state information about the other nodes in the cluster, like which nodes are reachable, what key ranges they are responsible for, and so on (*this is basically a copy of the hash ring*). Nodes share state information with each other to stay in sync. **Gossip protocol is a peer-to-peer communication mechanism** in which nodes periodically exchange state information about themselves and other nodes they know about. Each node initiates a gossip round every second to exchange state information about itself and other

nodes with one other random node. This means that any new event will eventually propagate through the system, and all nodes quickly learn about all other nodes in a cluster.

Every second each server exchanges information with one randomly selected server



Every second each server exchanges information about all the servers it knows about



Gossip protocol

External discovery through seed nodes

As we know, Dynamo nodes use gossip protocol to find the current state of the ring. This can result in a logical partition of the cluster in a particular scenario. Let's understand this with an example:

An administrator joins node A to the ring and then joins node B to the ring. Nodes A and B consider themselves part of the ring, yet neither would be immediately aware of each other. To prevent these logical partitions, Dynamo introduced the concept of **seed nodes**. Seed nodes are fully functional nodes and can be obtained either from a static configuration or a configuration service. This way, all nodes are aware of seed nodes. Each node communicates with seed nodes through gossip protocol to reconcile membership changes; therefore, logical partitions are highly unlikely.

[← Back](#)

[Next →](#)

Anti-entropy Through Merkle Trees

Dynamo Characteristics and Criticism

Dynamo Characteristics and Criticism

Let's explore the characteristics and the criticism of Dynamo's architecture.

We'll cover the following



- Responsibilities of a Dynamo's node
- Characteristics of Dynamo
- Criticism on Dynamo
- Datastores developed on the principles of Dynamo

Responsibilities of a Dynamo's node

Because Dynamo is completely decentralized and does not rely on a central/leader server (*unlike GFS, for example*), each node serves three functions:

1. **Managing `get()` and `put()` requests:** A node may act as a coordinator and manage all operations for a particular key or may forward the request to the appropriate node.
2. **Keeping track of membership and detecting failures:** Every node uses gossip protocol to keep track of other nodes in the system and their associated hash ranges.
3. **Local persistent storage:** Each node is responsible for being either the primary or replica store for keys that hash to a specific range of values. These (key, value) pairs are stored within that node using various

storage systems depending on application needs. A few examples of such storage systems are:

- BerkeleyDB Transactional Data Store
- MySQL (for large objects)
- An in-memory buffer (for best performance) backed by persistent storage

Characteristics of Dynamo

Here are a few reasons behind Dynamo's popularity:

- **Distributed:** Dynamo can run on a large number of machines.
- **Decentralized:** Dynamo is decentralized; there is no need for any central coordinator to oversee operations. All nodes are identical and can perform all functions of Dynamo.
- **Scalable:** By adding more nodes to the cluster, Dynamo can easily be scaled horizontally. No manual intervention or rebalancing is required. Additionally, Dynamo achieves linear scalability and proven fault-tolerance on commodity hardware.
- **Highly Available:** Dynamo is fault-tolerant, and the data remains available even if one or several nodes or data centers go down.
- **Fault-tolerant and reliable:** Since data is replicated to multiple nodes, fault-tolerance is pretty high.
- **Tunable consistency:** With Dynamo, applications can adjust the trade-off between availability and consistency of data, typically by configuring replication factor and consistency level settings.
- **Durable:** Dynamo stores data permanently.
- **Eventually Consistent:** Dynamo accepts the trade-off of strong consistency in favor of high availability.

Criticism on Dynamo

The following list contains criticism on Dynamo's design:

- Each Dynamo node contains the entire Dynamo routing table. This is likely to affect the scalability of the system as this routing table will grow larger and larger as nodes are added to the system.
- Dynamo seems to imply that it strives for symmetry, where every node in the system has the same set of roles and responsibilities, but later, it specifies some nodes as seeds. Seeds are special nodes that are externally discoverable. These are used to help prevent logical partitions in the Dynamo ring. This seems like it may violate Dynamo's symmetry principle.
- Although security was not a concern as Dynamo was built for internal use only, **DHTs can be susceptible to several different types of attacks**. While Amazon can assume a trusted environment, sometimes a buggy software can act in a manner quite similar to a malicious actor.
- Dynamo's design can be described as a “**leaky abstraction**,” where client applications are often asked to manage inconsistency, and the user experience is not 100% seamless. For example, inconsistencies in the shopping cart items may lead users to think that the website is buggy or unreliable.

Datastores developed on the principles of Dynamo

Dynamo is not open-source and was built for services running within Amazon. Two of the most famous datastores built on the principles of Dynamo are Riak (<https://riak.com/>) and Cassandra

(<https://cassandra.apache.org/>). Riak is a distributed NoSQL key-value data store that is highly available, scalable, fault-tolerant, and easy to operate. Cassandra is a distributed, decentralized, scalable, and highly available NoSQL wide-column database. Here is how they adopted different algorithms offered by Dynamo:

Technique	Apache Cassandra	Riak
Consistent Hashing with virtual nodes	✓	✓
Hinted Handoff	✓	✓
Anti-entropy with Merkle trees	✓ (manual repair)	✓
Vector Clocks	✗ (last-write-wins)	✓
Gossip-based protocol	✓	✓

← Back

Gossip Protocol

Next →

Summary: Dynamo

Summary: Dynamo

Here is a quick summary of Dynamo for you!

We'll cover the following

^

- Summary
- System design patterns
- References and further reading

Summary

1. Dynamo is a highly available **key-value store** developed by **Amazon** for their internal use.
2. Dynamo shows how business requirements can drive system designs. Amazon has chosen to sacrifice strong consistency for **higher availability** based on their business requirements.
3. Dynamo was designed with the understanding that system/hardware failures can and do occur.
4. Dynamo is a **peer-to-peer** distributed system, i.e., it does not have any leader or follower nodes. All nodes are equal and have the same set of roles and responsibilities. This also means that there is **no single point of failure**.
5. Dynamo uses the **Consistent Hashing** algorithm to distribute the data among nodes in the cluster automatically.
6. Data is replicated across nodes for fault tolerance and redundancy. Dynamo replicates writes to a **sloppy quorum** of other nodes in the system instead of a strict majority quorum.

7. For anti-entropy and to resolve conflicts, Dynamo uses **Merkle trees**.
8. Different storage engines can be plugged into Dynamo's local storage.
9. Dynamo uses the **gossip protocol** for inter-node communication.
10. Dynamo makes the system "always writeable" by using **hinted handoff**.
11. Dynamo's design philosophy is to ALWAYS allow writes. To support this, Dynamo allows concurrent writes. Writes can be performed by different servers concurrently, resulting in multiple versions of an object. Dynamo attempts to track and reconcile these changes using **vector clocks**. When Dynamo cannot reconcile an object's state from its vector clocks, it sends it to the client application for reconciliation (*the thought being that the clients have more semantic information on the object and may be able to reconcile it*).
12. Dynamo is able to successfully pull together several distributed techniques such as consistent hashing, p2p, gossip, vector clocks, and quorum, and combine them into a complex system.
13. Amazon built Dynamo for internal use only, so **no security** related issues were considered.

The following table presents a summary of the list of techniques Dynamo uses and their respective advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and Hinted Handoff	Provides high availability and durability guarantee when some of the replicas are not available

Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas on the background
Membership and failure detection	Gossip-based membership protocol and failure detection	Preserves symmetry and avoid centralized monitoring

System design patterns

Here is a summary of system design patterns used in Dynamo:

- **Consistent Hashing:** Dynamo uses Consistent Hashing to distribute its data across nodes.
- **Quorum:** To ensure data consistency, each Dynamo write operation can be configured to be successful only if the data has been written to at least a quorum of replica nodes.
- **Gossip protocol:** Dynamo uses gossip protocol that allows each node to keep track of state information about the other nodes in the cluster.
- **Hinted Handoff:** Dynamo nodes use Hinted Handoff to remember the write operation for failing nodes.
- **Read Repair:** Dynamo uses ‘Read Repair’ to push the latest version of the data to nodes with the older versions.
- **Vector clocks:** To reconcile concurrent updates on an object Dynamo uses Vector clocks.
- **Merkle trees:** For anti-entropy and to resolve conflicts in the background, Dynamo uses Merkle trees.

References and further reading

- Amazon's Dynamo
(https://www.allthingsdistributed.com/2007/10/amazons_dynamo.html)
- Eventually Consistent
(https://www.allthingsdistributed.com/2007/12/eventually_consistent.html)
- Bigtable (<https://research.google/pubs/pub27898/>)
- DynamoDB (<https://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html>)
- CRDT (https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type)
- A Decade of Dynamo (<https://www.allthingsdistributed.com/2017/10/a-decade-of-dynamo.html>)
- Riak (<https://docs.riak.com/riak/kv/2.2.0/learn/dynamo/>)
- Dynamo Architecture (<https://www.youtube.com/watch?v=w96lLsbI1q8>)
- Dynamo: A flawed architecture (<https://news.ycombinator.com/item?id=915212>)

← Back

Dynamo Characteristics and Criticism

Next →

Quiz: Dynamo

Cassandra: How to Design a Wide-column NoSQL Database?

Cassandra: Introduction

Let's explore Cassandra and its use cases.

We'll cover the following



- Goal
- Background
- What is Cassandra?
- Cassandra use cases

Goal

Design a distributed and scalable system that can store a huge amount of structured data, which is indexed by a row key where each row can have an unbounded number of columns.

Background

Cassandra is an open-source Apache project. It was originally developed at Facebook in 2007 for their inbox search feature. The Apache Cassandra architecture is designed to provide **scalability**, **availability**, and **reliability** to store large amounts of data. Cassandra combines the distributed nature of **Amazon's Dynamo** which is a key-value store and the data model of **Google's BigTable** which is a column-based data store. With Cassandra's **decentralized architecture**, there is **no single point of failure** in a cluster, and its performance can scale linearly with the addition of nodes.

What is Cassandra?

Cassandra is a **distributed, decentralized, scalable**, and **highly available** NoSQL database. In terms of CAP theorem (<https://www.educative.io/collection/page/5668639101419520/5559029852536832/5998984290631680>), Cassandra is typically classified as an AP (*i.e., available and partition tolerant*) system which means that availability and partition tolerance are generally considered more important than the consistency. Cassandra can be tuned with replication-factor and consistency levels to meet strong consistency requirements, but this comes with a performance cost. In other words, data can be highly available with low consistency guarantees, or it can be highly consistent with lower availability. Cassandra uses **peer-to-peer architecture**, with each node connected to all other nodes. Each Cassandra node performs all database operations and can serve client requests without the need for any leader node.

Disclaimer: All of the following lessons are Cassandra version agnostic and try to explore the general design and architectural layout of different Cassandra components and operations.

Cassandra use cases

By default, Cassandra is not a strongly consistent database (it is eventually consistent), hence, any application where consistency is not a concern can utilize Cassandra. Though Cassandra can support strong consistency, it comes with a performance impact. Cassandra is **optimized for high throughput** and **faster writes**, and can be used for collecting big data for performing real-time analysis. Here are some of its top use cases:

- **Storing key-value data with high availability** - Reddit and Digg use Cassandra as a persistent store for their data. Cassandra's ability to scale linearly without any downtime makes it very suitable for their growth needs.
- **Time series data model** - Due to its data model and log-structured storage engine, Cassandra benefits from high-performing write operations. This also makes Cassandra well suited for storing and analyzing sequentially captured metrics (i.e., measurements from sensors, application logs, etc.). Such usages take advantage of the fact that columns in a row are determined by the application, not a predefined schema. Each row in a table can contain a different number of columns, and there is no requirement for the column names to match.
- **Write-heavy applications** - Cassandra is especially suited for write-intensive applications such as time-series streaming services, sensor logs, and Internet of Things (IoT) applications.

[← Back](#)

Mock Interview: Dynamo

[Next →](#)

High-level Architecture

High-level Architecture

This lesson gives a brief overview of Cassandra's architecture.

We'll cover the following



- Cassandra common terms
- High-level architecture
 - Data partitioning
 - Cassandra keys
 - Clustering keys
 - Partitioner
 - Coordinator node

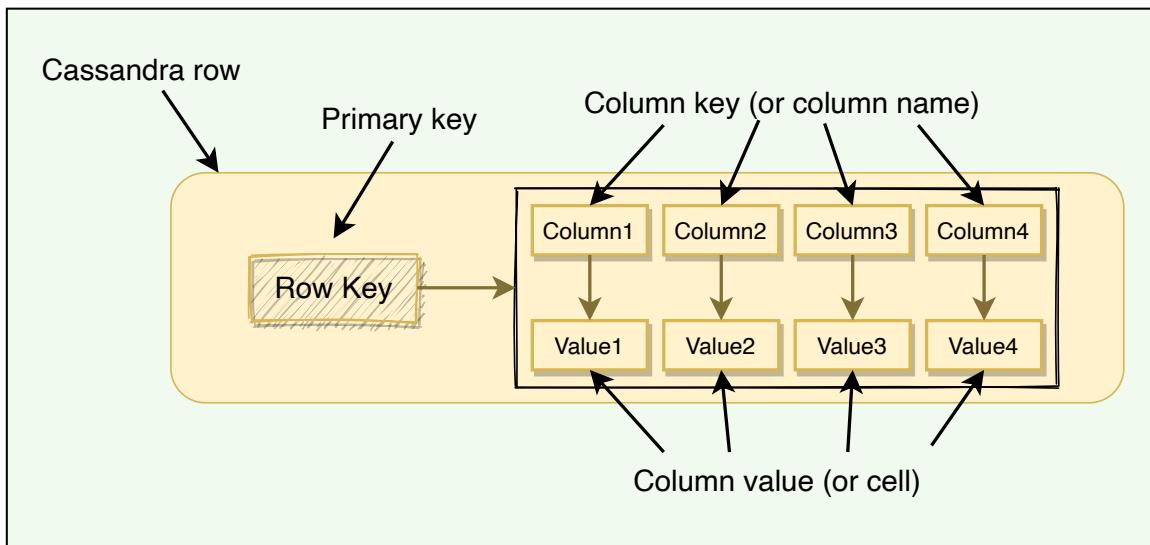
Cassandra common terms

Before digging deep into Cassandra's architecture, let's first go through some of its common terms:

Column: A column is a key-value pair and is the most basic unit of data structure.

- **Column key:** Uniquely identifies a column in a row.
- **Column value:** Stores one value or a collection of values.

Row: A row is a container for columns referenced by primary key. Cassandra does not store a column that has a null value; this saves a lot of space.



Components of a Cassandra row

Table: A table is a container of rows.

Keyspace: Keyspace is a container for tables that span over one or more Cassandra nodes.

Cluster: Container of Keyspaces is called a cluster.

Node: Node refers to a computer system running an instance of Cassandra. A node can be a physical host, a machine instance in the cloud, or even a Docker container.

NoSQL: Cassandra is a NoSQL database which means we cannot have `joins` between tables, there are no `foreign keys`, and while querying, we cannot add any column in the `where` clause other than the primary key. These constraints should be kept in mind before deciding to use Cassandra.

High-level architecture

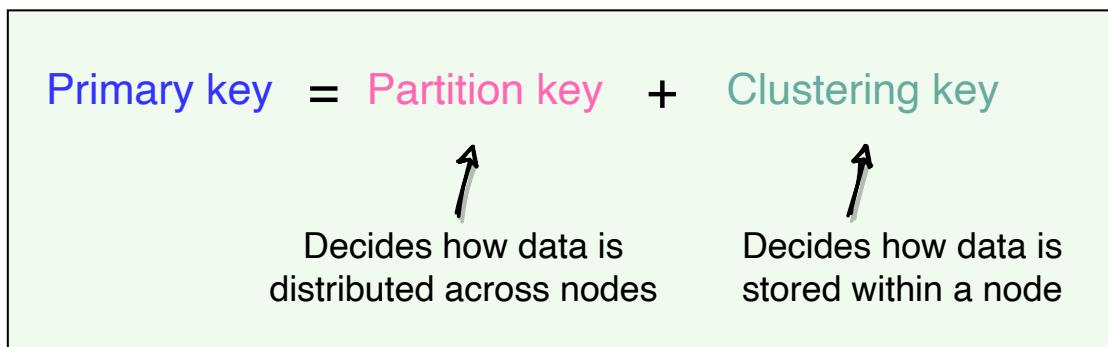
Data partitioning

Cassandra uses **consistent hashing** for data partitioning. Please take a look at Dynamo's data partitioning (<https://www.educative.io/collection/page/5668639101419520/5559029852536832/6501426287607808>); all consistent hashing details described in it applies to Cassandra too.

Let's look into mechanisms that Cassandra applies to uniquely identify rows.

Cassandra keys

The **Primary key** uniquely identifies each row of a table. In Cassandra primary key has two parts:



Parts of a primary key

The partition key decides which node stores the data, and the clustering key decides how the data is stored within a node. Let's take the example of a table with PRIMARY KEY (city_id, employee_id). This primary key has two parts represented by the two columns:

1. city_id is the partition key. This means that the data will be partitioned by the city_id field, that is, all rows with the same city_id will reside on the same node.
2. employee_id is the clustering key. This means that within each node, the data is stored in sorted order according to the employee_id column.

Clustering keys

As described above, clustering keys define how the data is stored within a node. We can have multiple clustering keys; all columns listed after the partition key are called clustering columns. Clustering columns specify the order that the data is arranged on a node.

The diagram illustrates the storage structure of data within a node. A table is shown with four columns: State, City, Zip, and Rest of columns. The 'State' column is highlighted in yellow and labeled 'Partition key'. The 'City' and 'Zip' columns are also highlighted in yellow and labeled 'Clustering key'. Arrows point from the text labels to their respective columns in the table. The table data is as follows:

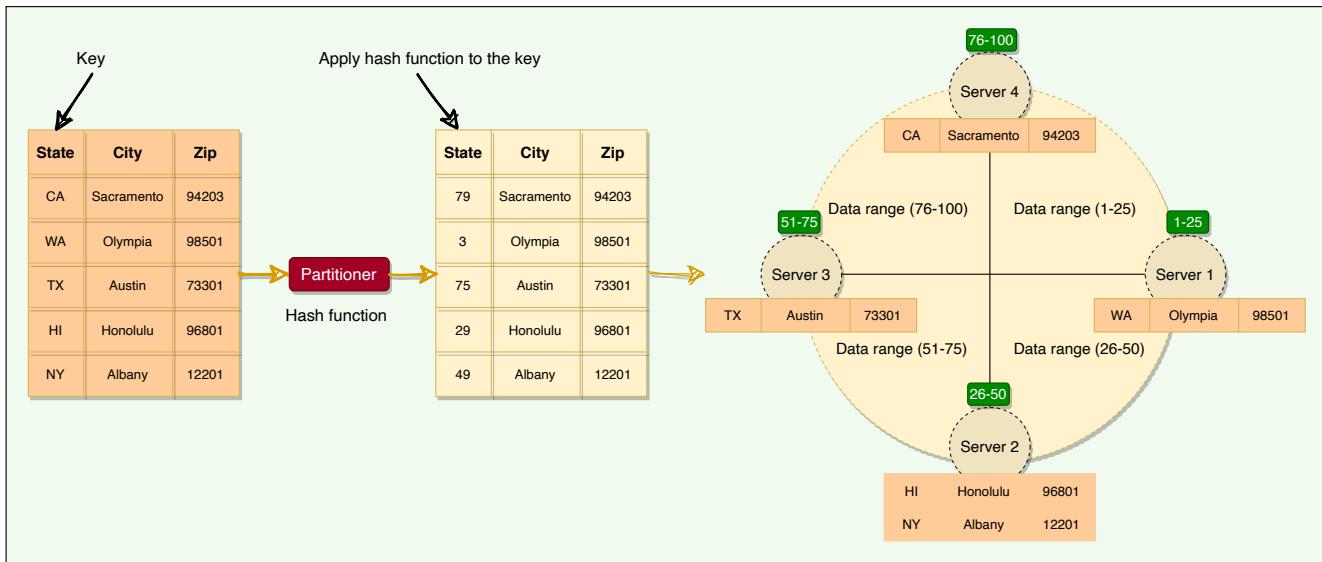
State	City	Zip	Rest of columns
CA	Sacramento	94203	x
	Sacramento	94250	x
	Los Angeles	90012	x
Los Angeles	Los Angeles	90040	x
	Los Angeles	90090	x
	Redmond	98052	x
WA	Seattle	98170	x
	Seattle	98191	x

On a node, data is ordered by the clustering key.

Clustering key

Partitioner

Partitioner is the component responsible for determining how data is distributed on the Consistent Hash ring. When Cassandra inserts some data into a cluster, the partitioner performs the first step, which is to apply a hashing algorithm to the partition key. The output of this hashing algorithm determines within which range the data lies and hence, on which node the data will be stored.



Distributing data on the Consistent Hashing ring

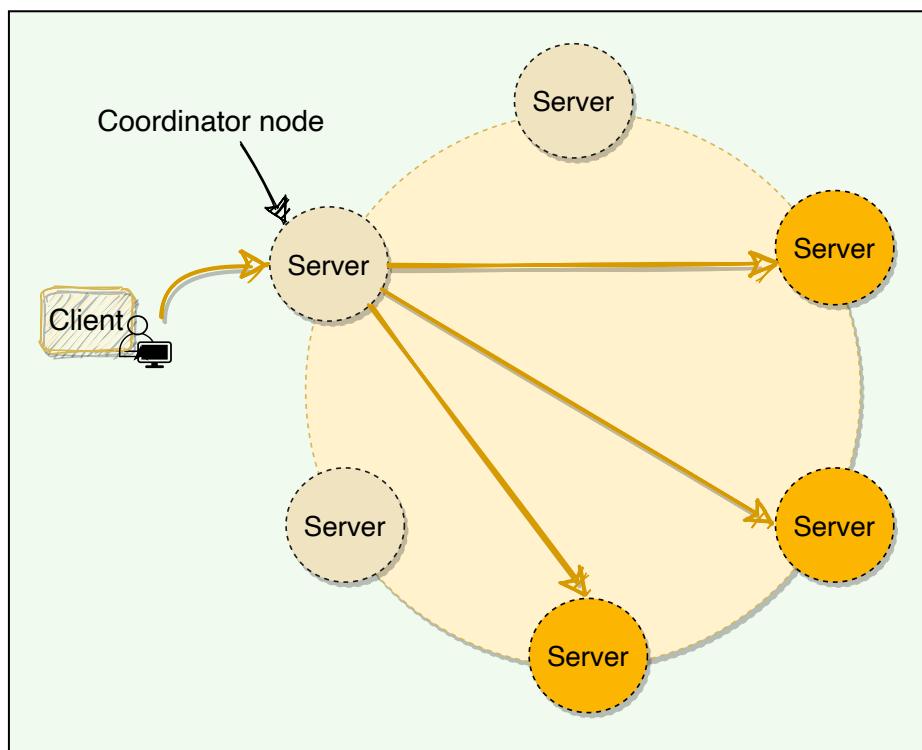
By default, Cassandra uses the Murmur3 hashing function. Murmur3 will always produce the same hash for a given partition key. This means that we can always find the node where a specific row is stored. Cassandra does allow custom hashing functions, however, once a cluster is initialized with a particular partitioner, it cannot be changed later. In Cassandra's default configuration, a token is a 64-bit integer. This gives a possible range for tokens from -2^{63} to $2^{63} - 1$.

All Cassandra nodes learn about the token assignments of other nodes through gossip (discussed later). This means any node can handle a request for any other node's range. The node receiving the request is called the

coordinator, and any node can act in this role. If a key does not belong to the coordinator's range, it forwards the request to the replicas responsible for that range.

Coordinator node

A client may connect to any node in the cluster to initiate a read or write query. This node is known as the coordinator node. The coordinator identifies the nodes responsible for the data that is being written or read and forwards the queries to them.



Client connecting to the coordinator node

As of now, we discussed the core concepts of Cassandra. Let's dig deeper into some of its advanced distributed concepts.

← Back

Next →

Replication

Let's explore Cassandra's replication strategy.

We'll cover the following



- Replication factor
- Replication strategy
 - Simple replication strategy
 - Network topology strategy

Each node in Cassandra serves as a replica for a different range of data. Cassandra stores multiple copies of data and spreads them across various replicas, so that if one node is down, other replicas can respond to queries for that range of data. This process of replicating the data on to different nodes depends upon two factors:

- Replication factor
- Replication strategy

Replication Factor + Replication Strategy

Decides how many replicas the system will have

Decides which nodes will be responsible for the replicas

Replication factor

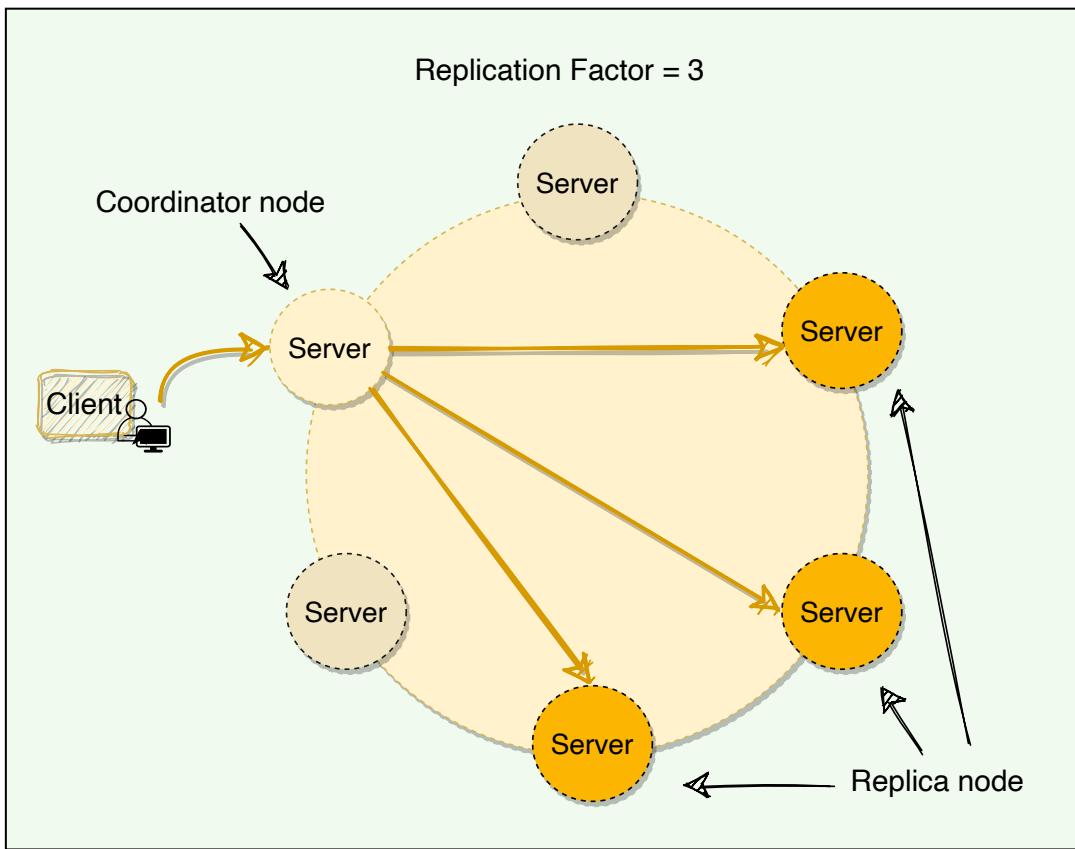
The replication factor is the number of nodes that will receive the copy of the same data. This means, if a cluster has a replication factor of 3, each row will be stored on three different nodes. Each keyspace in Cassandra can have a different replication factor.

Replication strategy

The node that owns the range in which the hash of the partition key falls will be the first replica; all the additional replicas are placed on the consecutive nodes. Cassandra places the subsequent replicas on the next node in a clockwise manner. There are two replication strategies in Cassandra:

Simple replication strategy

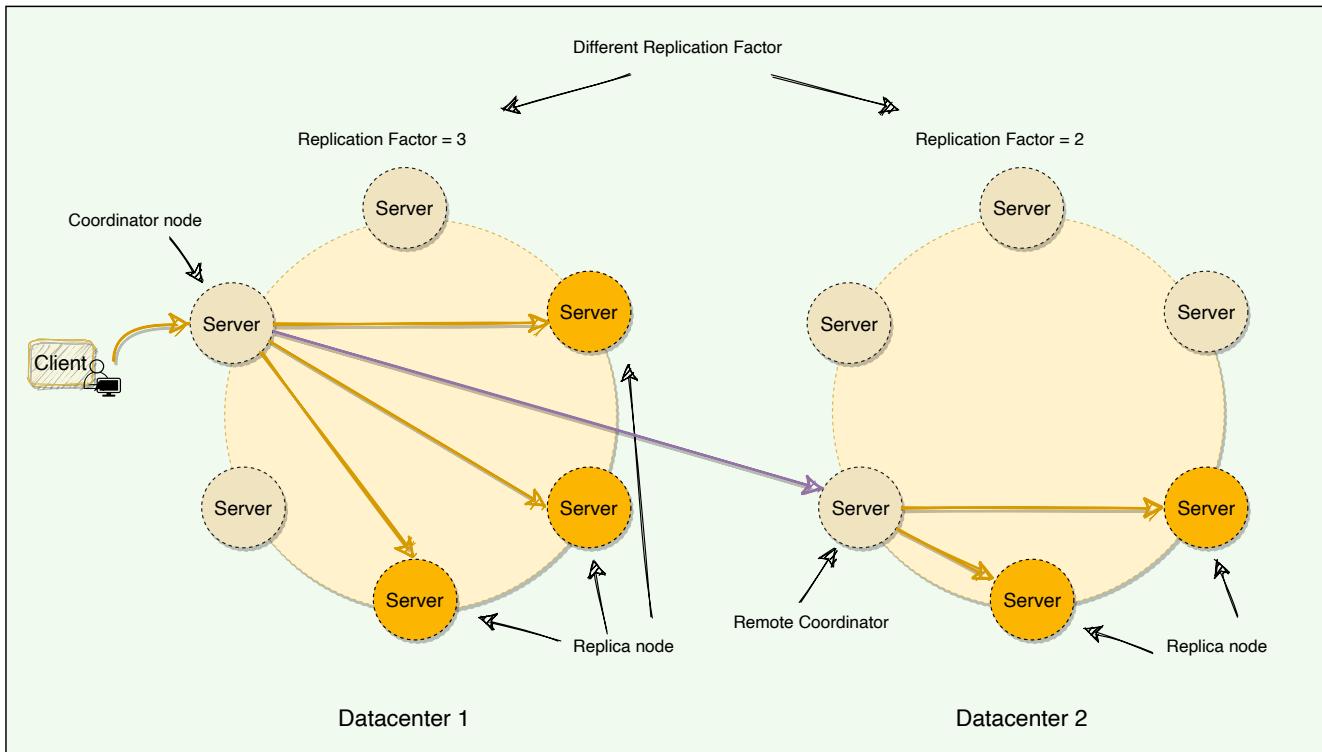
This strategy is used only for a single data center cluster. Under this strategy, Cassandra places the first replica on a node determined by the partitioner and the subsequent replicas on the next node in a clockwise manner.



Simple replication with 3 replicas

Network topology strategy

This strategy is used for multiple data-centers. Under this strategy, we can specify different replication factors for different data-centers. This enables us to specify how many replicas will be placed in each data center. Additional replicas are always placed on the next nodes in a clockwise manner.



Network topology strategy for replication

← Back

High-level Architecture

Next →

Cassandra Consistency Levels

Cassandra Consistency Levels

Let's explore how Cassandra manages data consistency.

We'll cover the following



- What are Cassandra's consistency levels?
- Write consistency levels
 - Hinted handoff
- Read consistency levels
- Snitch

What are Cassandra's consistency levels?

Cassandra's consistency level is defined as the minimum number of Cassandra nodes that must fulfill a read or write operation before the operation can be considered successful. Cassandra allows us to specify different consistency levels for read and write operations. Also, Cassandra has tunable consistency, i.e., we can increase or decrease the consistency levels for each request.

There is always a tradeoff between consistency and performance. A higher consistency level means that more nodes need to respond to a read or write query, giving the user more assurance that the values present on each replica are the same.

Write consistency levels

For write operations, the consistency level specifies how many replica nodes must respond for the write to be reported as successful to the client. The consistency level is specified per query by the client. Because Cassandra is eventually consistent, updates to other replica nodes may continue in the background. Here are different write consistency levels that Cassandra offers:

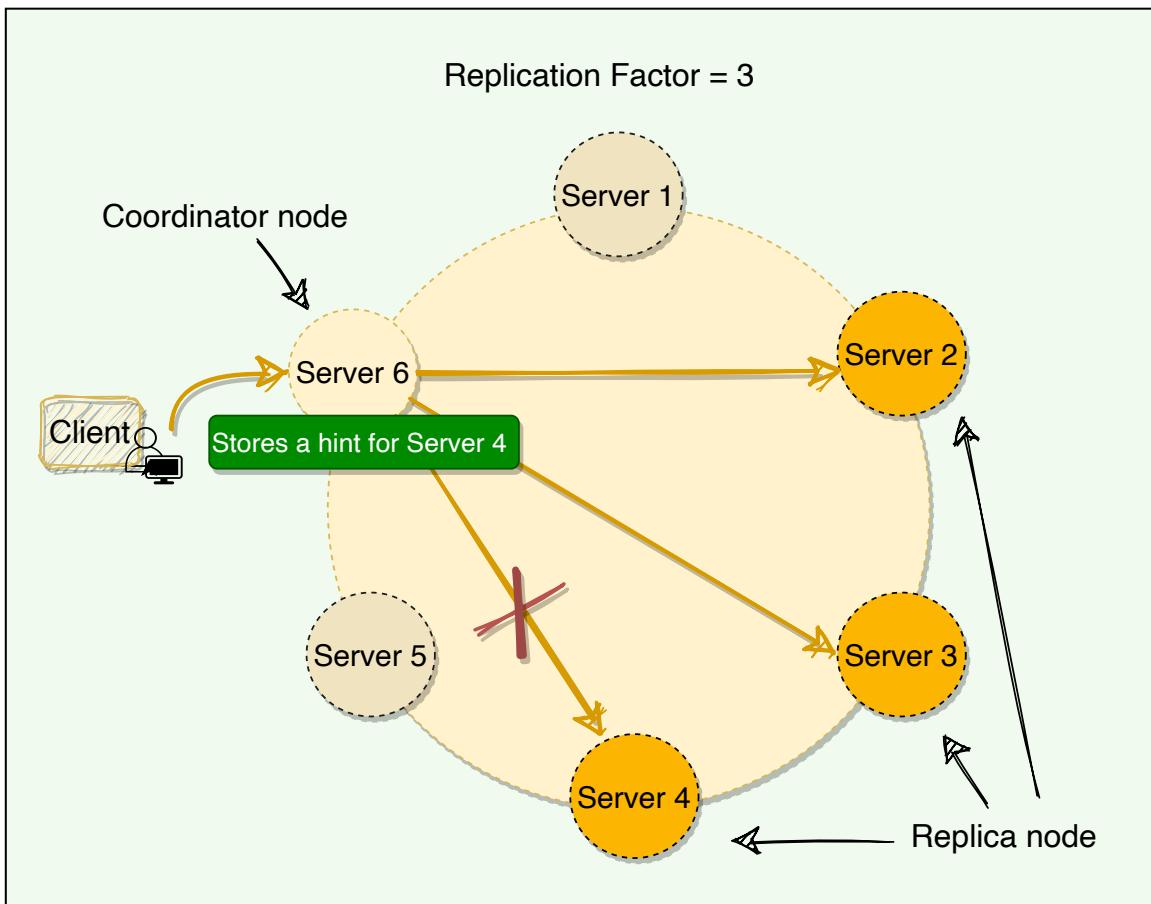
- **One or Two or Three:** The data must be written to at least the specified number of replica nodes before a write is considered successful.
- **Quorum:** The data must be written to at least a quorum (or majority) of replica nodes. Quorum is defined as $\text{floor}(RF/2 + 1)$, where RF represents the replication factor. For example, in a cluster with a replication factor of five, if three nodes return success, the write is considered successful.
- **All:** Ensures that the data is written to all replica nodes. This consistency level provides the highest consistency but lowest availability as writes will fail if any replica is down.
- **Local_Quorum:** Ensures that the data is written to a quorum of nodes in the same datacenter as the coordinator. It does not wait for the response from the other data-centers.
- **Each_Quorum:** Ensures that the data is written to a quorum of nodes in each datacenter.
- **Any:** The data must be written to at least one node. In the extreme case, when all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff (discussed below) has been written. ‘Any’ consistency level provides the lowest latency and highest availability, however, it comes with the lowest consistency. If all replica nodes are down at write time, an ‘Any’ write is not readable until the

replica nodes for that partition have recovered and the latest data is written on them.

How does Cassandra perform a write operation? For a write, the coordinator node contacts all replicas, as determined by the replication factor, and considers the write successful when a number of replicas equal to the consistency level acknowledge the write.

Hinted handoff

Depending upon the consistency level, Cassandra can still serve write requests even when nodes are down. For example, if we have the replication factor of three and the client is writing with a quorum consistency level. This means that if one of the nodes is down, Cassandra can still write on the remaining two nodes to fulfill the consistency level, hence, making the write successful.



Hinted handoff

Now when the node which was down comes online again, how should we write data to it? Cassandra accomplishes this through hinted handoff.

When a node is down or does not respond to a write request, the coordinator node writes a hint in a text file on the local disk. This hint contains the data itself along with information about which node the data belongs to. When the coordinator node discovers from the Gossiper (will be discussed later) that a node for which it holds hints has recovered, it forwards the write requests for each hint to the target. Furthermore, each node every ten minutes checks to see if the failing node, for which it is holding any hints, has recovered.

With consistency level ‘Any,’ if all the replica nodes are down, the coordinator node will write the hints for all the nodes and report success to the client. However, this data will not reappear in any subsequent reads.

until one of the replica nodes comes back online, and the coordinator node successfully forwards the write requests to it. This is assuming that the coordinator node is up when the replica node comes back. This also means that we can lose our data if the coordinator node dies and never comes back. For this reason, we should avoid using the ‘Any’ consistency level.

If a node is offline for some time, the hints can build up considerably on other nodes. Now, when the failed node comes back online, other nodes tend to flood that node with write requests. This can cause issues on the node, as it is already trying to come back after a failure. To address this problem, Cassandra limits the storage of hints to a configurable time window. It is also possible to disable hinted handoff entirely.

Cassandra, by default, stores hints for three hours. After three hours, older hints will be removed, which means, if now the failed node recovers, it will have stale data. Cassandra can fix this stale data while serving a read request. Cassandra can issue a **Read Repair** when it sees stale data; we will go through this while discussing the read path.

One thing to remember: When the cluster cannot meet the consistency level specified by the client, Cassandra fails the write request and does not store a hint.

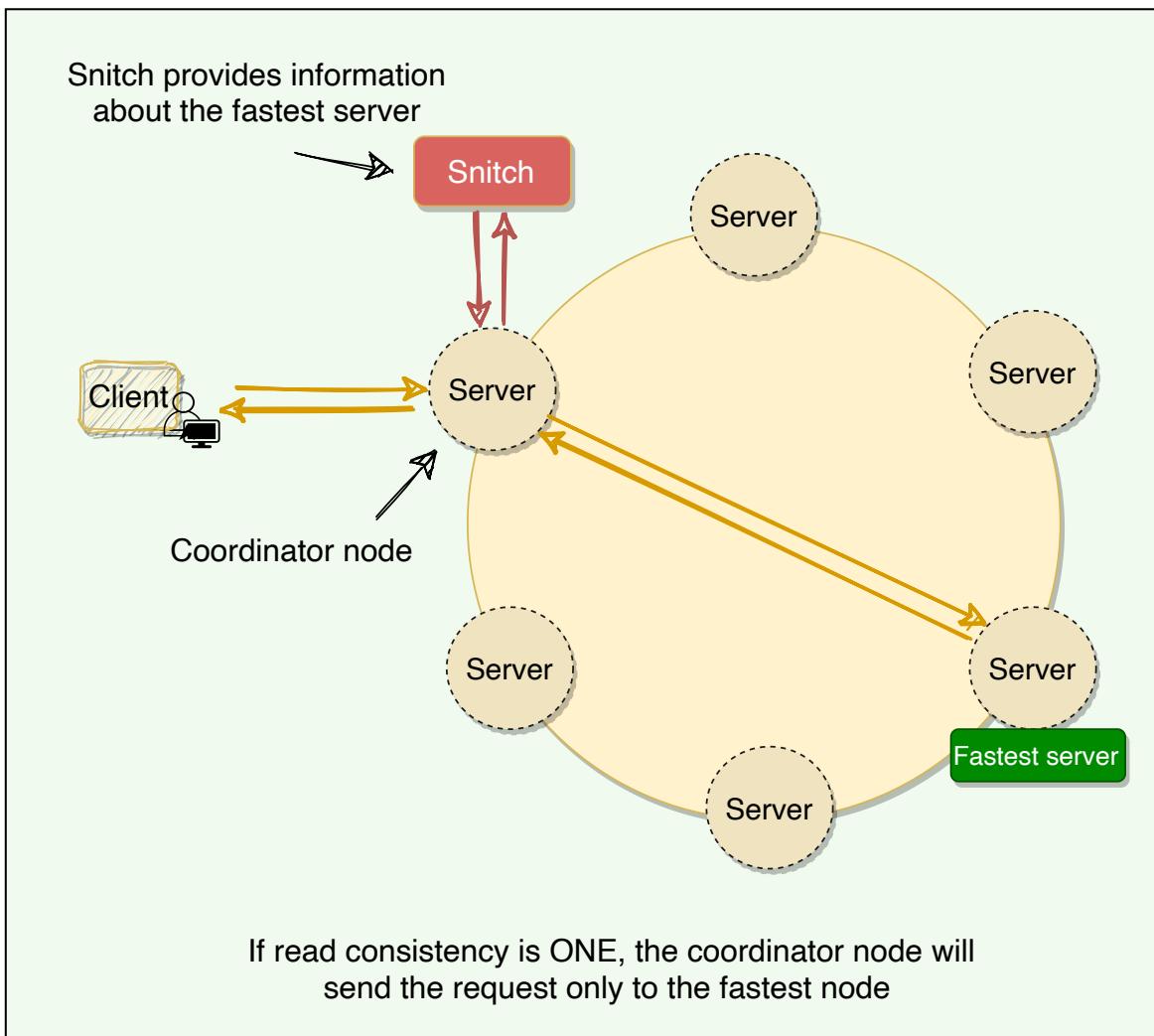
Read consistency levels

The consistency level for read queries specifies how many replica nodes must respond to a read request before returning the data. For example, for a read request with a consistency level of quorum and replication factor of three, the coordinator waits for successful replies from at least two nodes.

Cassandra has the same consistency levels for read requests as that of write operations except Each_Quorum (because it is very expensive).

To achieve strong consistency in Cassandra: $R + W > RF$ gives us strong consistency. In this equation, R , W , and RF are the read replica count, the write replica count, and the replication factor, respectively. All client reads will see the most recent write in this scenario, and we will have strong consistency.

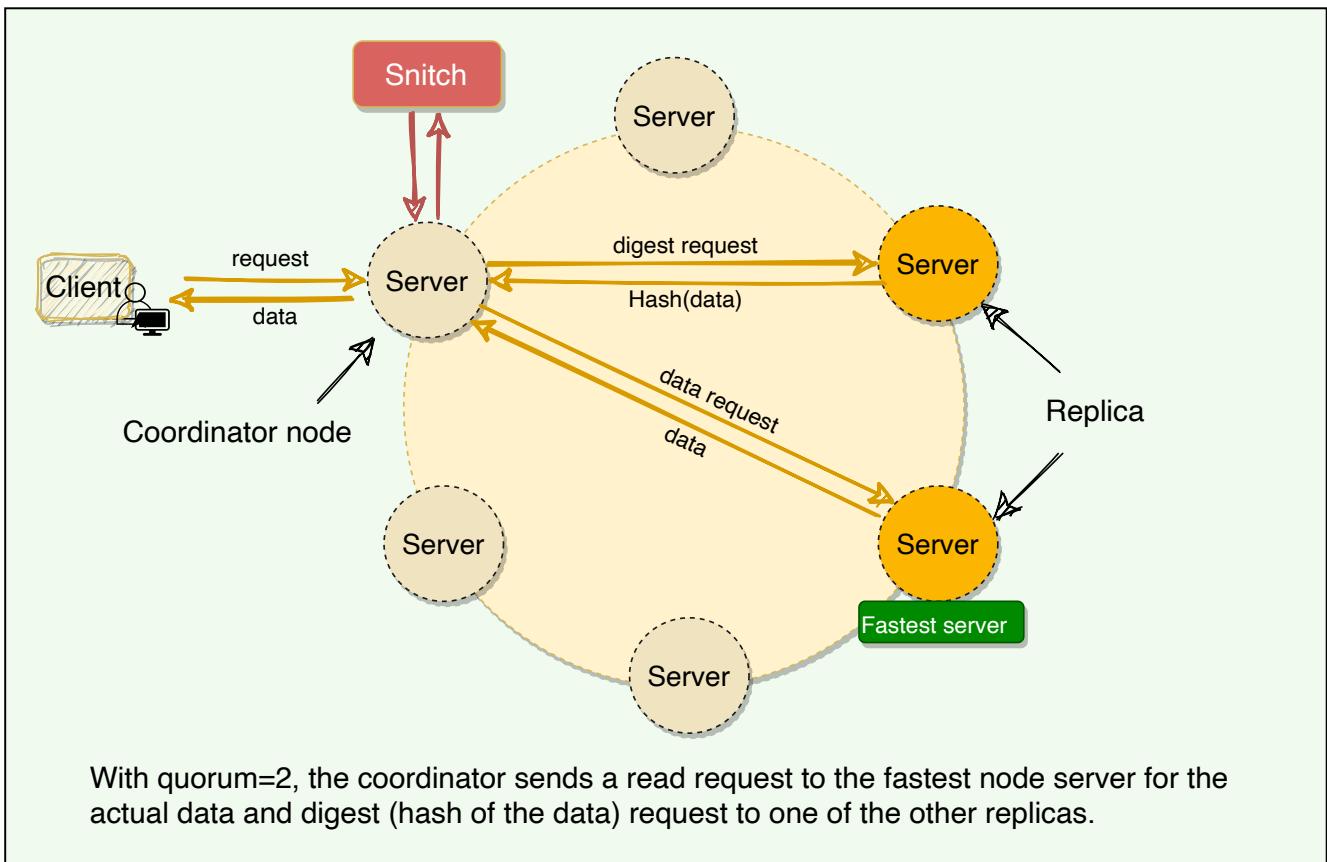
Snitch: The Snitch is an application that determines the proximity of nodes within the ring and also tells which nodes are faster. Cassandra nodes use this information to route read/write requests efficiently. We will discuss this in detail later.



Coordinator node forwards the read request to the fastest server

How does Cassandra perform a read operation? The coordinator always sends the read request to the fastest node. For example, for Quorum=2, the coordinator sends the request to the fastest node and the digest of the data from the second-fastest node. The digest is a checksum of the data and is used to save network bandwidth.

If the digest does not match, it means some replicas do not have the latest version of the data. In this case, the coordinator reads the data from all the replicas to determine the latest data. The coordinator then returns the latest data to the client and initiates a **read repair** request. The read repair operation pushes the newer version of data to nodes with the older version.



Read repair

While discussing Cassandra's write path, we saw that the nodes could become out of sync due to network issues, node failures, corrupted disks, etc. The read repair operation helps nodes to resync with the latest data. Read

operation is used as an opportunity to repair inconsistent data across replicas. The latest write-timestamp is used as a marker for the correct version of data. The read repair operation is performed only in a portion of the total reads to avoid performance degradation. Read repairs are opportunistic operations and not a primary operation for anti-entropy.

Read Repair Chance: When the read consistency level is less than ‘All,’ Cassandra performs a read repair probabilistically. By default, Cassandra tries to read repair 10% of all requests with DC local read repair. In this case, Cassandra immediately sends a response when the consistency level is met and performs the read repair asynchronously in the background.

Snitch

Snitch keeps track of the network topology of Cassandra nodes. It determines which data-centers and racks nodes belong to. Cassandra uses this information to route requests efficiently. Here are the two main functions of a snitch in Cassandra:

- Snitch determines the proximity of nodes within the ring and also monitors the read latencies to avoid reading from nodes that have slowed down. Each node in Cassandra uses this information to route requests efficiently.
- Cassandra’s replication strategy uses the information provided by the Snitch to spread the replicas across the cluster intelligently. Cassandra will do its best by not having more than one replica on the same “rack”.

To understand Snitch’s role, let’s take the example of Cassandra’s read operation. Let’s assume that the client is performing a read with a quorum consistency level, and the data is replicated on five nodes. To support maximum read speed, Cassandra selects a single replica to query for the full

object and asks for the digest of the data from two additional nodes in order to ensure that the latest version of the data is returned. The Snitch helps to identify the fastest replica, and Cassandra asks this replica for the full object.

← Back

Replication

Next →

Gossiper

Gossiper

Let's explore how Cassandra uses gossip protocol to keep track of the state of the system.

We'll cover the following



- How does Cassandra use gossip protocol?
- Node failure detection

How does Cassandra use gossip protocol?

Cassandra uses **gossip protocol** that allows each node to keep track of state information about the other nodes in the cluster. Nodes share state information with each other to stay in sync. Gossip protocol is a peer-to-peer communication mechanism in which nodes periodically exchange state information about themselves and other nodes they know about. Each node initiates a gossip round every second to exchange state information about themselves (and other nodes) with one to three other random nodes. This way, all nodes quickly learn about all other nodes in a cluster.

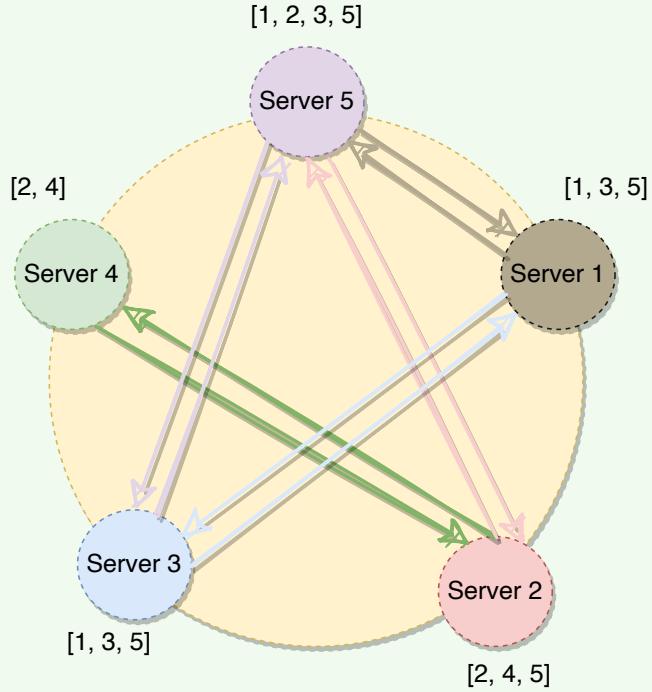
Each gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

Generation number: In Cassandra, each node stores a generation number which is incremented every time a node restarts. This generation number is included in each gossip message exchanged between nodes and is used to

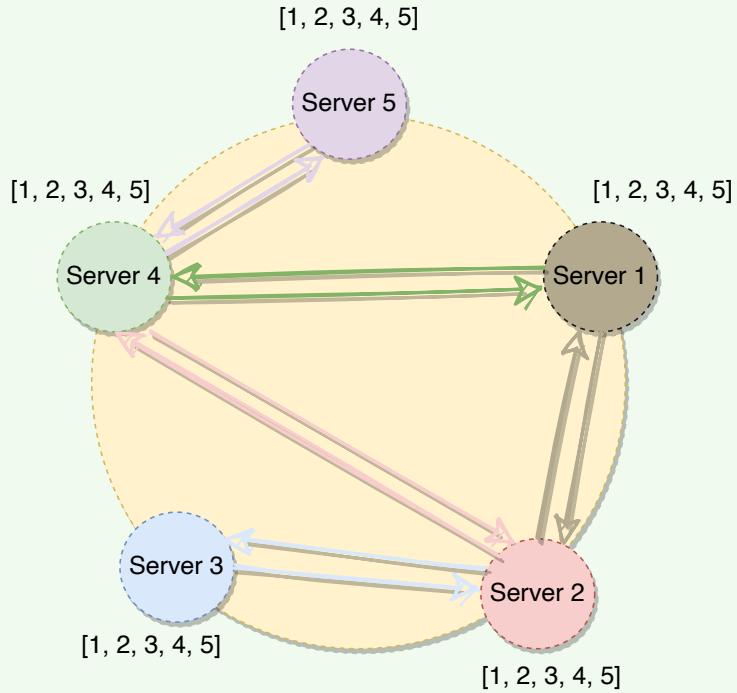
distinguish the current state of a node from its state before a restart. The generation number remains the same while the node is alive and is incremented each time the node restarts. The node receiving the gossip message can compare the generation number it knows and the gossip message's generation number. If the generation number in the gossip message is higher, it knows that the node was restarted.

Seed nodes: To prevent problems in gossip communications, Cassandra designates a list of nodes as the seeds in a cluster. This is critical for a node starting up for the first time. By default, a node remembers other nodes it has gossiped with between subsequent restarts. The seed node designation has no purpose other than bootstrapping the gossip process for new nodes joining the cluster. Thus, seed nodes are not a single point of failure, nor do they have any other special purpose in cluster operations other than the bootstrapping of nodes.

Every second each server exchanges information with one randomly selected server



Every second each server exchanges information about all the servers it knows about



Gossip protocol

Node failure detection

Accurately detecting failures is a hard problem to solve as we cannot say with 100% surety that if a system is genuinely down or is just very slow in responding due to heavy load, network congestion, etc. Mechanisms like Heartbeating outputs a boolean value telling us if the system is alive or not; there is no middle ground. Heartbeating uses a fixed timeout, and if there is no heartbeat from a server, the system, after the timeout, assumes that the server has crashed. Here the value of the timeout is critical. If we keep the timeout short, the system will be able to detect failures quickly but with many false positives due to slow machines or faulty networks. On the other hand, if we keep the timeout long, the false positives will be reduced, but the system will not perform efficiently for being slow in detecting failures.

Cassandra uses an adaptive failure detection mechanism as described by **Phi Accrual Failure Detector**. This algorithm uses historical heartbeat information to make the threshold adaptive. A generic Accrual Failure Detector, instead of telling that the server is alive or not, outputs the suspicion level about a server; a higher suspicion level means there are higher chances that the server is down. Using Phi Accrual Failure Detector, if a node does not respond, its suspicion level is increased and could be declared dead later. As a node's suspicion level increases, the system can gradually decide to stop sending new requests to it. Phi Accrual Failure Detector makes a distributed system efficient as it takes into account fluctuations in the network environment and other intermittent server issues before declaring a system completely dead.

Now that we have discussed Cassandra's major components, let's see how Cassandra performs its read and write operations.

[← Back](#)

[Next →](#)

Anatomy of Cassandra's Write Operation

Let's dig deeper into the components involved in Cassandra's write path.

We'll cover the following



- Commit log
- MemTable
- SSTable

Cassandra stores data both in memory and on disk to provide both high performance and durability. Every write includes a timestamp. Write path involves a lot of components, here is the summary of Cassandra's write path:

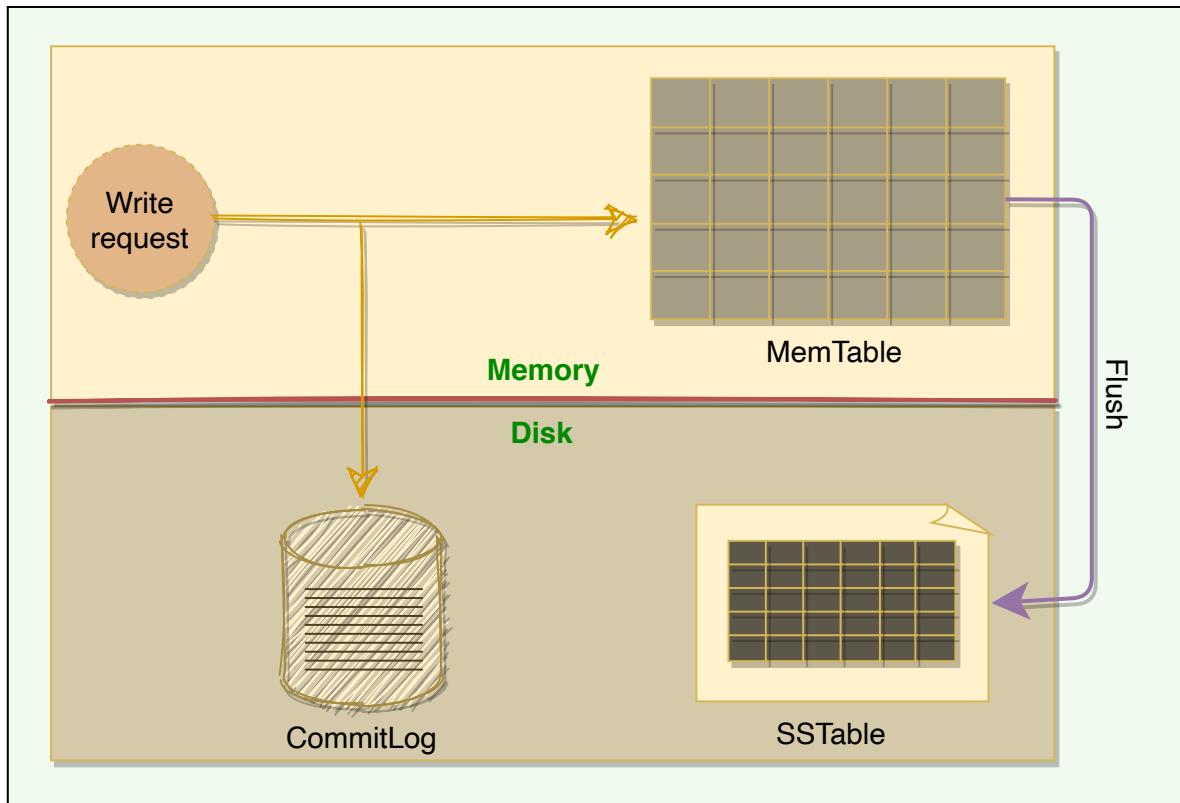
1. Each write is appended to a **commit log**, which is stored on disk.
2. Then it is written to **MemTable** in memory.
3. Periodically, MemTables are flushed to **SSTables** on the disk.
4. Periodically, compaction runs to merge SSTables.

Let's dig deeper into these parts.

Commit log

When a node receives a write request, it immediately writes the data to a commit log. The commit log is a write-ahead log and is stored on disk. It is used as a crash-recovery mechanism to support Cassandra's durability goals. A write will not be considered successful on the node until it's written to the commit log; this ensures that if a write operation does not make it to the in-

memory store (*the MemTable*, discussed in a moment), it will still be possible to recover the data. If we shut down the node or it crashes unexpectedly, the commit log can ensure that data is not lost. That's because if the node restarts, the commit log gets replayed.



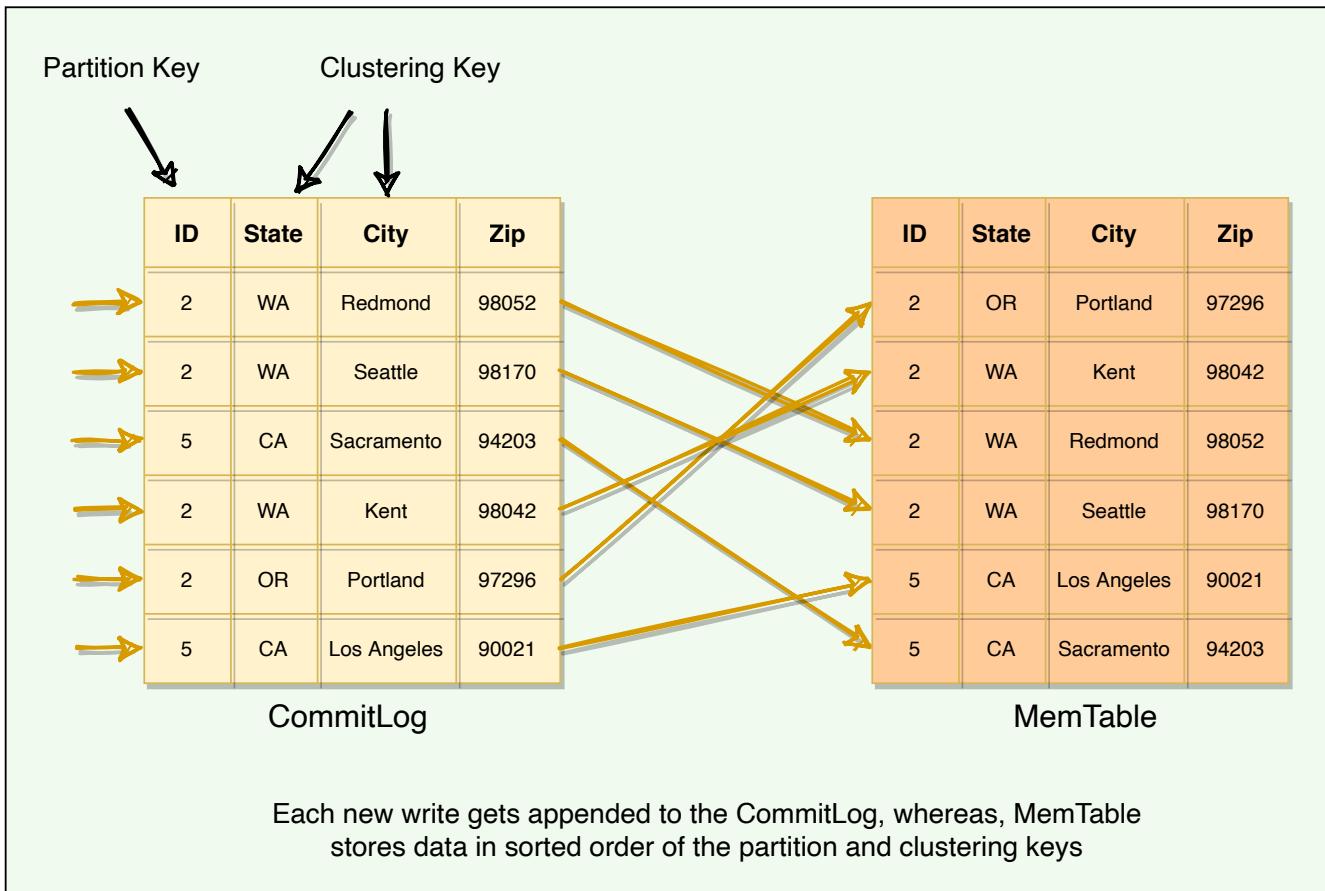
Cassandra's write path

MemTable

After it is written to the commit log, the data is written to a memory-resident data structure called the MemTable.

- Each node has a MemTable in memory for each Cassandra table.
- Each MemTable contains data for a specific Cassandra table, and it resembles that table in memory.
- Each MemTable accrues writes and provides reads for data not yet flushed to disk.

- Commit log stores all the writes in sequential order, with each new write appended to the end, whereas MemTable stores data in the sorted order of partition key and clustering columns.
- After writing data to the Commit Log and MemTable, the node sends an acknowledgment to the coordinator that the data has been successfully written.



Storing data to commit log and MemTable

SSTable

When the number of objects stored in the MemTable reaches a threshold, the contents of the MemTable are flushed to disk in a file called SSTable. At this point, a new MemTable is created to store subsequent data. This flushing is a

non-blocking operation; multiple MemTables may exist for a single table, one current, and the rest waiting to be flushed. Each SSTable contains data for a specific table.

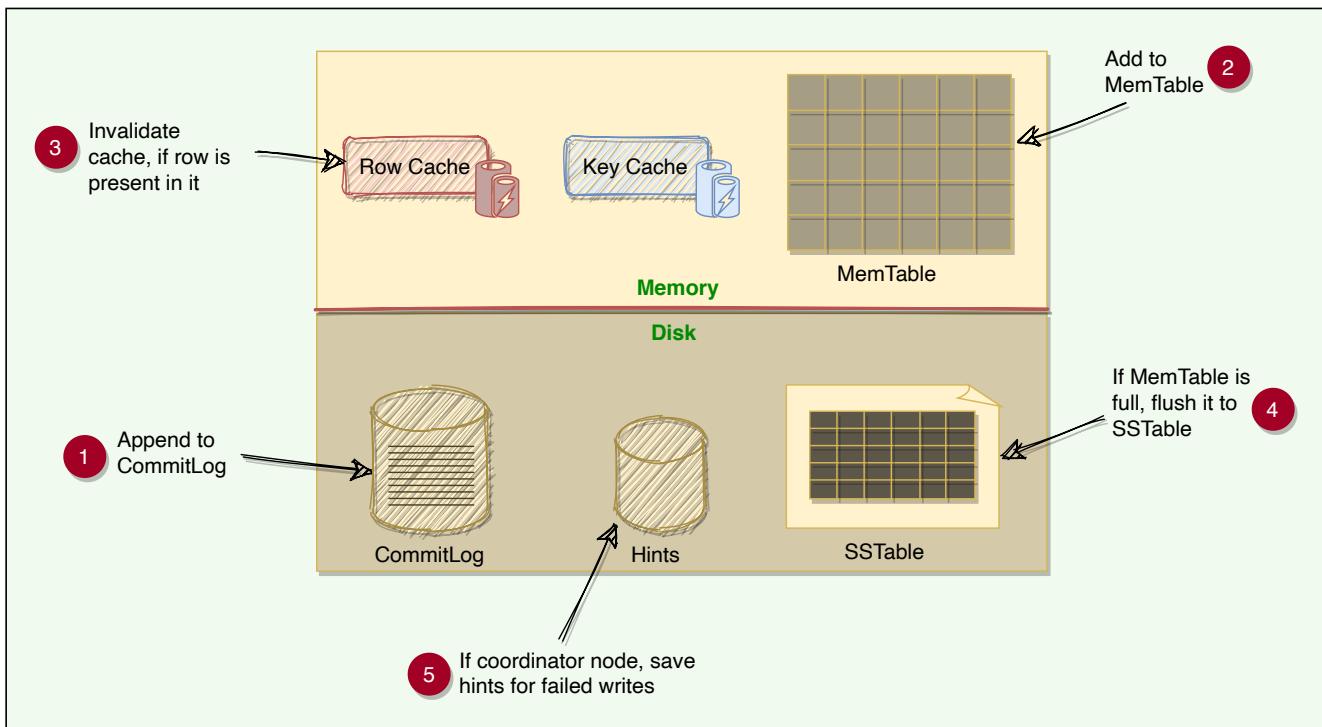
When the MemTable is flushed to SSTables, corresponding entries in the Commit Log are removed.

Why are they called ‘SSTables’? The term ‘SSTables’ is short for ‘Sorted String Table’ and first appeared in Google’s Bigtable which is also a storage system. Cassandra borrowed this term even though it does not store data as strings on the disk.

Once a MemTable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application. If we are not allowed to update SSTables, how do we delete or update a column? In Cassandra, each delete or update is considered a new write operation. We will look into this in detail while discussing Tombstones.

The current data state of a Cassandra table consists of its MemTables in memory and SSTables on the disk. Therefore, on reads, Cassandra will read both SSTables and MemTables to find data values, as the MemTable may contain values that have not yet been flushed to the disk. The MemTable works like a write-back cache that Cassandra looks up by key.

Generation number is an index number that is incremented every time a new SSTable is created for a table and is used to uniquely identify SSTables. Here is the summary of Cassandra’s write path:



Anatomy of Cassandra's write path

← Back

Gossiper

Next →

Anatomy of Cassandra's Read Operati...

Anatomy of Cassandra's Read Operation

Let's explore Cassandra's read path.

We'll cover the following



- Caching
- Reading from MemTable
- Reading from SSTable
 - Bloom filters
 - How are SSTables stored on the disk?
 - Partition index summary file
 - Reading SSTable through key cache

Let's dig deeper into the components involved in Cassandra's read path.

Caching

To boost read performance, Cassandra provides three optional forms of caching:

1. **Row cache:** The row cache, caches frequently read (or hot) rows. It stores a complete data row, which can be returned directly to the client if requested by a read operation. This can significantly speed up read access for frequently accessed rows, at the cost of more memory usage.
2. **Key cache:** Key cache stores a map of recently read partition keys to their SSTable offsets. This facilitates faster read access into SSTables

stored on disk and improves the read performance but could slow down the writes, as we have to update the Key cache for every write.

3. **Chunk cache:** Chunk cache is used to store uncompressed chunks of data read from SSTable files that are accessed frequently.

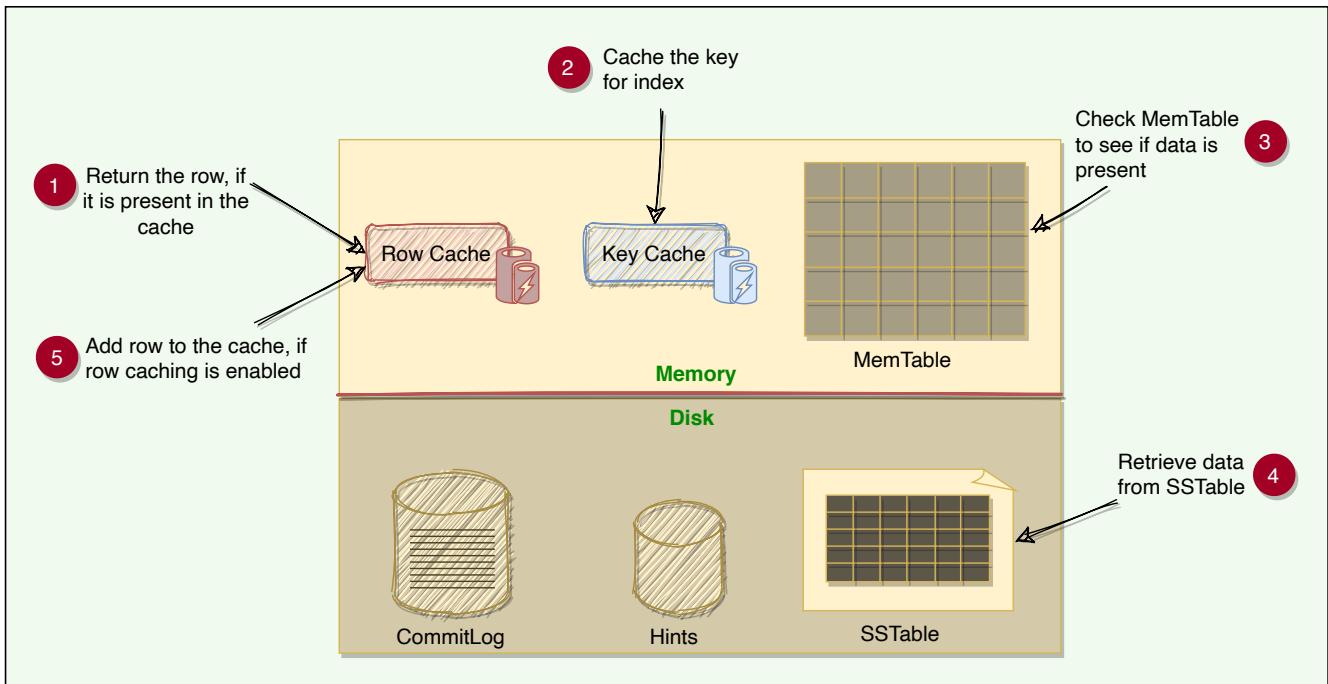
Reading from MemTable

As we know, data is sorted by the partition key and the clustering columns. Let's take an example. Here we have two partitions of a table with partition keys '2' and '5'. The clustering columns are the state and city names. When a read request comes in, the node performs a binary search on the partition key to find the required partition and then return the row.

ID	State	City	Zip
2	OR	Portland	97296
2	WA	Kent	98042
2	WA	Redmond	98052
2	WA	Seattle	98170
5	CA	Los Angeles	90021
5	CA	Sacramento	94203

Reading data from MemTable

Here is the summary of Cassandra's read path:



Anatomy of Cassandra's read path

Reading from SSTable

Bloom filters

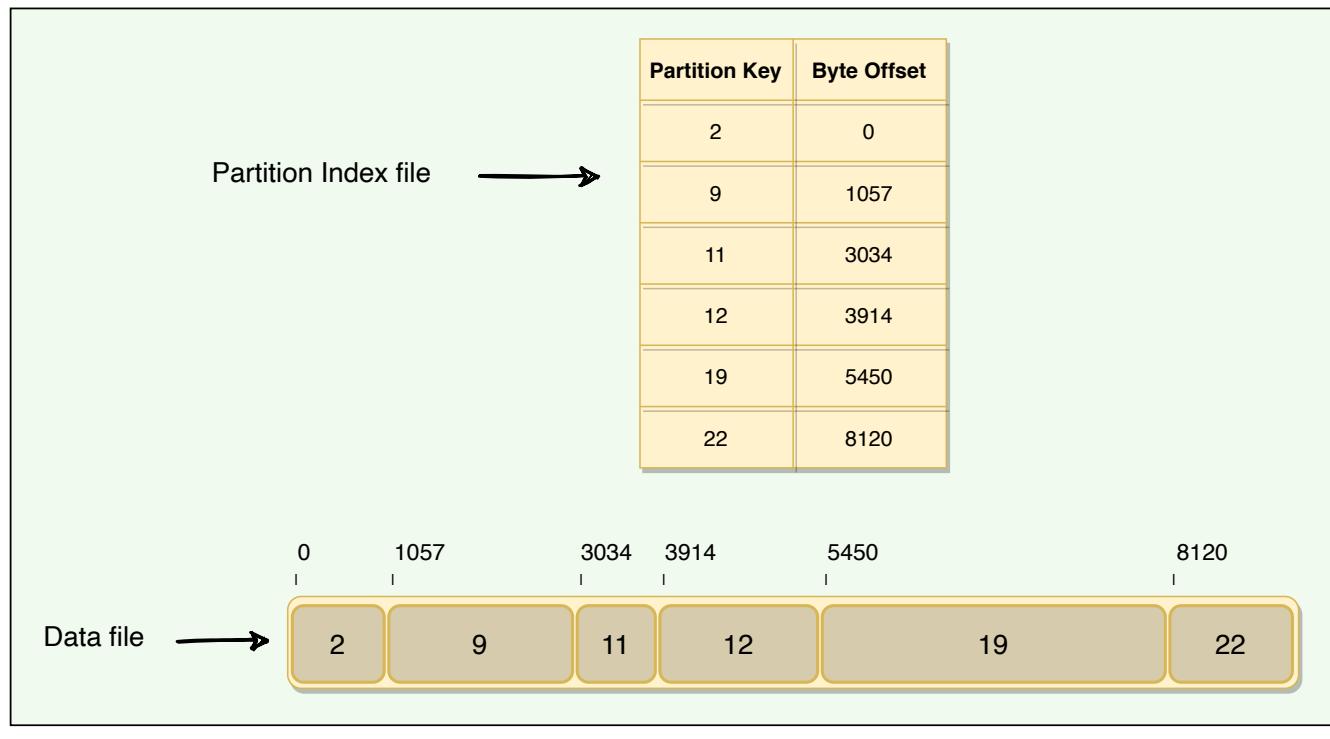
Each SSTable has a Bloom filter associated with it, which tells if a particular key is present in it or not. Bloom filters are used to boost the performance of read operations. Bloom filters are very fast, non-deterministic algorithms for testing whether an element is a member of a set. They are non-deterministic because it is possible to get a false-positive read from a Bloom filter, but false-negative is not possible. Bloom filters work by mapping the values in a data set into a bit array and condensing a larger data set into a digest string using a hash function. The digest, by definition, uses a much smaller amount of memory than the original data would. The filters are stored in memory and are used to improve performance by reducing the need for disk access on key lookups. Disk access is typically much slower than memory access. So, in a way, a Bloom filter is a special kind of key cache.

Cassandra maintains a Bloom filter for each SSTable. When a query is performed, the Bloom filter is checked first before accessing the disk. Because false negatives are not possible, if the filter indicates that the element does not exist in the set, it certainly does not; but if the filter thinks that the element is in the set, the disk is accessed to make sure.

How are SSTables stored on the disk?

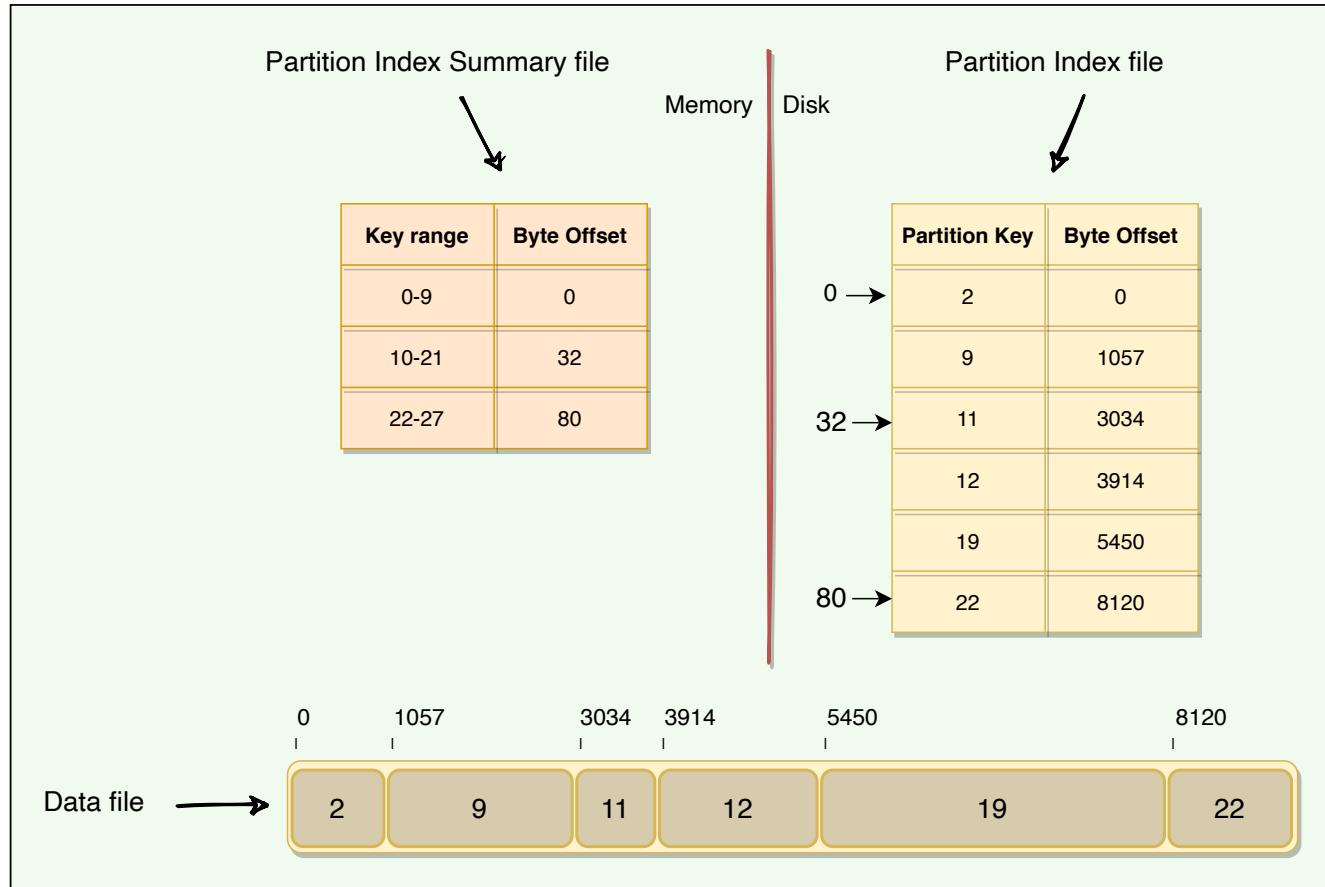
Each SSTable consists of two files:

- 1. Data File:** Actual data is stored in a data file. It has partitions and rows associated with those partitions. The partitions are in sorted order.
- 2. Partition Index file:** Stored on disk, partition index file stores the sorted partition keys mapped to their SSTable offsets. It enables locating a partition exactly in an SSTable rather than scanning data.



Partition index summary file

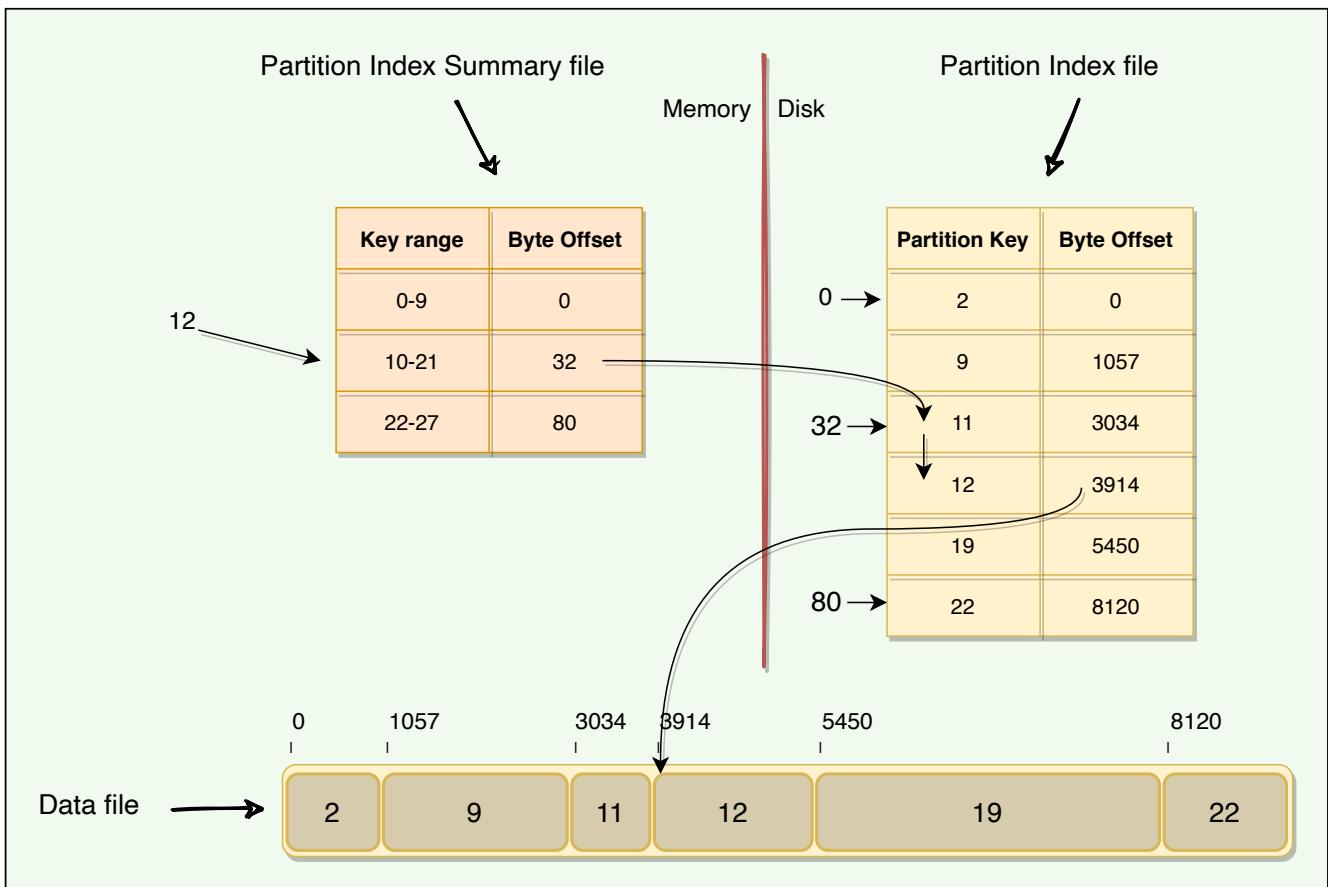
Stored in memory, the Partition Index Summary file stores the summary of the Partition Index file. This is done for performance improvement.



Reading from partition index summary file

If we want to read data for `key=12`, here are the steps we need to follow (also shown in the figure below):

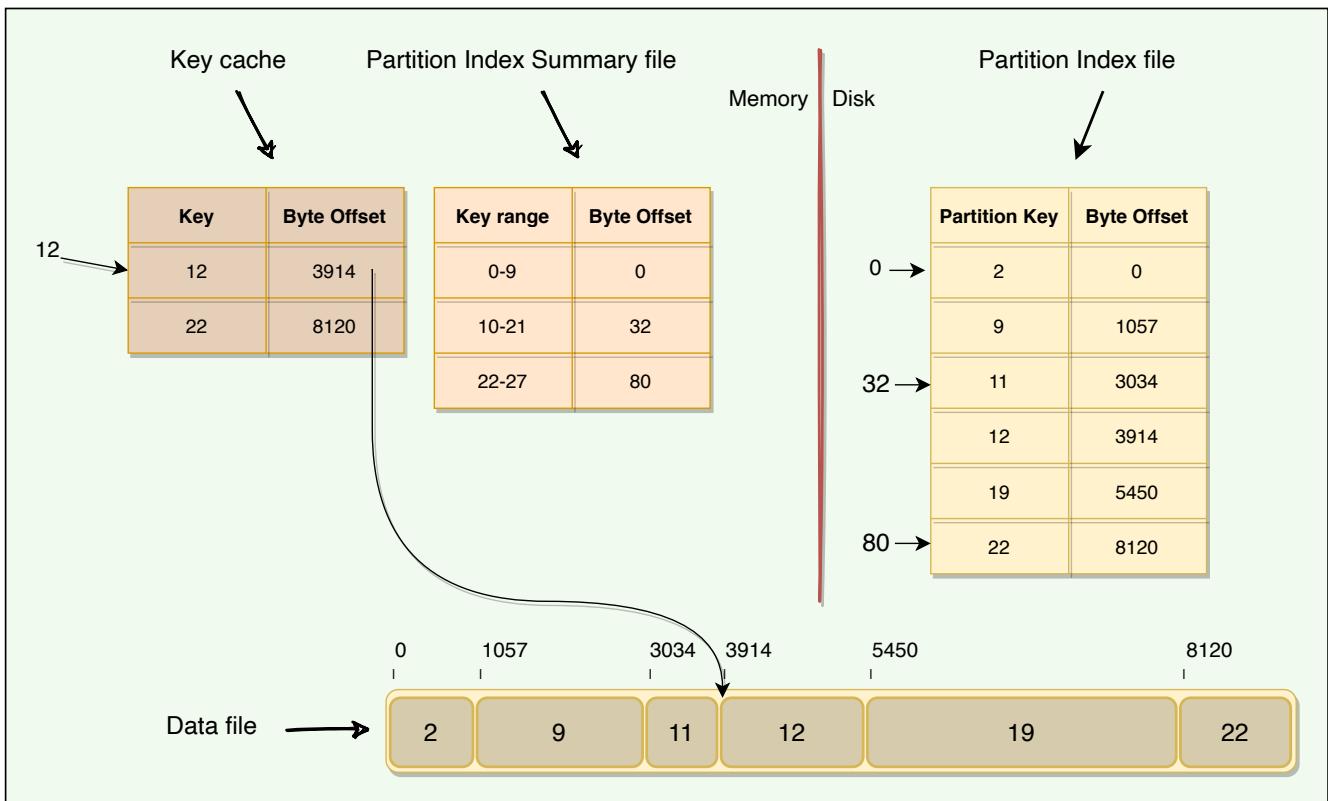
1. In the Partition Index Summary file, find the key range in which the `key=12` lies. This will give us offset (=32) into the Partition Index file.
2. Jump to offset 32 in the Partition Index file to search for the offset of `key=12`. This will give us offset (=3914) into the SSTable file.
3. Jump to SSTable at offset 3914 to read the data for `key=12`



Reading from partition index summary file

Reading SSTable through key cache

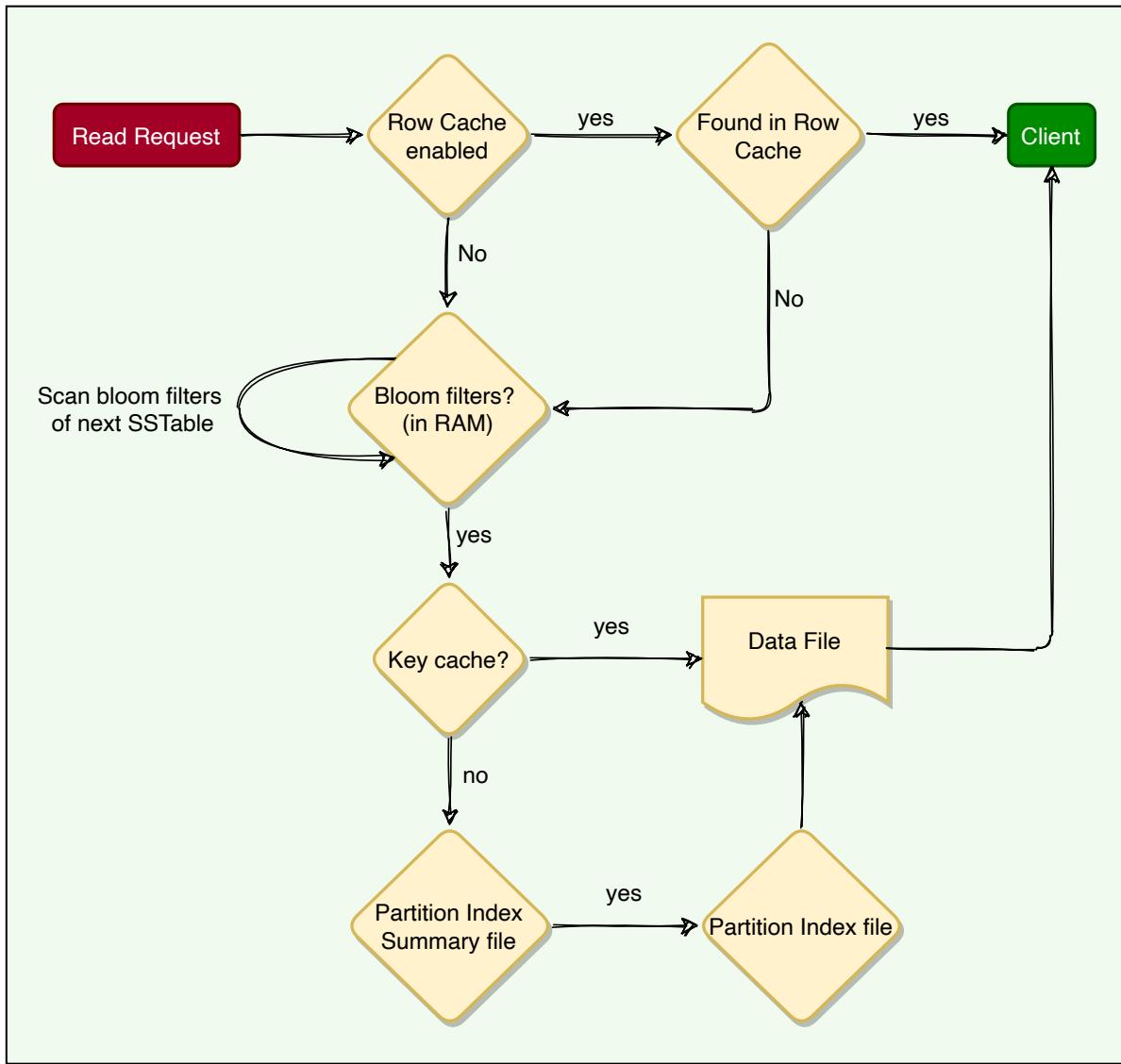
As the Key Cache stores a map of recently read partition keys to their SSTable offsets, it is the fastest way to find the required row in the SSTable.



Reading SSTable through key cache

If data is not present in MemTable, we have to look it up in SSTables or other data structures like partition index, etc. Here is the summary of Cassandra's read operation:

1. First, Cassandra checks if the row is present in the Row Cache. If present, the data is returned, and the request ends.
2. If the row is not present in the Row Cache, bloom filters are checked. If a bloom filter indicates that the data is present in an SSTable, Cassandra looks for the required partition in that SSTable.
3. The key cache is checked for the partition key presence. A cache hit provides an offset for the partition in SSTable. This offset is then used to retrieve the partition, and the request completes.
4. Cassandra continues to seek the partition in the partition summary and partition index. These structures also provide the partition offset in an SSTable which is then used to retrieve the partition and return. The caches are updated if present with the latest data read.



Cassandra's read operation workflow

[← Back](#)

Anatomy of Cassandra's Write Operat...

[Next →](#)

Compaction

Compaction

Let's explore how Cassandra handles compaction.

We'll cover the following

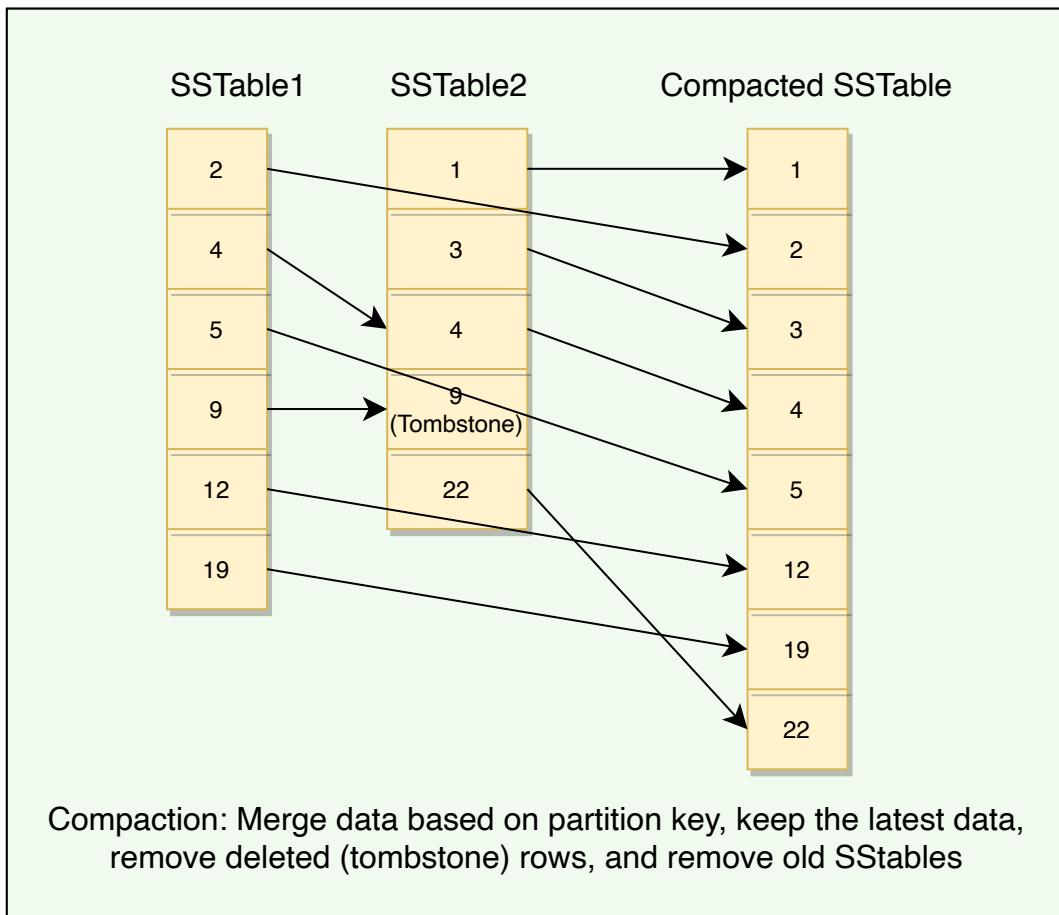


- How does compaction work in Cassandra?
- Compaction strategies
- Sequential writes

How does compaction work in Cassandra?

As we have already discussed, SSTables are immutable, which helps Cassandra achieve such high write speeds. Flushing of MemTable to SSTable is a continuous process. This means we can have a large number of SSTables lying on the disk. While reading, it is tedious to scan all these SSTables. So, to improve the read performance, we need compaction. Compaction in Cassandra refers to the operation of merging multiple related SSTables into a single new one. During compaction, the data in SSTables is merged: the keys are merged, columns are combined, obsolete values are discarded, and a new index is created.

On compaction, the merged data is sorted, a new index is created over the sorted data, and this freshly merged, sorted, and indexed data is written to a single new SSTable.



Compacting two SSTables into one

- Compaction will reduce the number of SSTables to consult and therefore improve read performance.
- Compaction will also reclaim space taken by obsolete data in SSTable.

Compaction strategies

SizeTiered Compaction Strategy: This compaction strategy is suitable for insert-heavy and general workloads. This is the default compaction strategy and is triggered when multiple SSTables of a similar size are present.

Leveled Compaction Strategy: This strategy is used to optimize read performance. This strategy groups SSTables into levels, each of which has a fixed size limit which is ten times larger than the previous level.

Time Window Compaction Strategy: The Time Window Compaction Strategy is designed to work on time series data. It compacts SSTables within a configured time window. This strategy is ideal for time series data which is immutable after a fixed time interval.

Sequential writes

Sequential writes are the primary reason that writes perform so well in Cassandra. No reads or seeks of any kind are required for writing a value to Cassandra because all writes are ‘append’ operations. This makes the speed of the disk one key limitation on performance. Compaction is intended to amortize the reorganization of data, but it uses sequential I/O to do so, which makes it efficient. If Cassandra naively inserted values where they ultimately belonged, writing clients would pay for seeks upfront.

← Back

Anatomy of Cassandra's Read Operati...

Next →

Tombstones

Tombstones

Let's explore how Tombstones work in Cassandra.

We'll cover the following



- What are Tombstones?
- Common problems associated with Tombstones

What are Tombstones?

An interesting case with Cassandra can be when we delete some data for a node that is down or unreachable, that node could miss a delete. When that node comes back online later and a repair occurs, the node could “resurrect” the data that had been previously deleted by re-sharing it with other nodes. To prevent deleted data from being reintroduced, Cassandra uses a concept called a tombstone. A tombstone is similar to the idea of a “**soft delete**” from the relational database world. When we delete data, Cassandra does not delete it right away, instead associates a tombstone with it, with a time to expiry. In other words, a tombstone is a marker that is kept to indicate data that has been deleted. When we execute a delete operation, the data is not immediately deleted. Instead, it's treated as an update operation that places a tombstone on the value.

Each tombstone has an expiry time associated with it, representing the amount of time that nodes will wait before removing the data permanently. By default, each tombstone has an expiry of ten days. The purpose of this delay is to give a node that is unavailable time to recover. If a node is down

longer than this value, then it should be treated as failed and replaced.

Tombstones are removed as part of compaction. During compaction, any row with an expired tombstone will not be propagated further.

Common problems associated with Tombstones

Tombstones make Cassandra writes actions efficient because the data is not removed right away when deleted. Instead, it is removed later during compaction. Having said that, Tombstones cause the following problems:

- As a tombstone itself is a record, **it takes storage space**. Hence, it should be kept in mind that upon deletion, the application will end up increasing the data size instead of shrinking it. Furthermore, if there are a lot of tombstones, the available storage for the application could be substantially reduced.
- When a table accumulates many tombstones, read queries on that table could become slow and can cause serious performance problems like timeouts. This is because we have to read much more data until the actual compaction happens and removes the tombstones.

[← Back](#)

Compaction

[Next →](#)

Summary: Cassandra

Summary: Cassandra

Here is a quick summary of Cassandra for you!

We'll cover the following



- Summary
- System design patterns
- Cassandra characteristics
- References and further reading

Summary

1. Cassandra is a **distributed, decentralized, scalable, and highly available** NoSQL database.
2. Cassandra was designed with the understanding that software/hardware **failures can and do occur**.
3. Cassandra is a **peer-to-peer** distributed system, i.e., it does not have any leader or follower nodes. All nodes are equal, and there is no single point of failure.
4. Data, in Cassandra, is automatically distributed across nodes.
5. Data is replicated across the nodes for fault tolerance and redundancy.
6. Cassandra uses the **Consistent Hashing** algorithm to distribute the data among nodes in the cluster. Cassandra cluster has a ring-type architecture, where its nodes are logically distributed like a ring.
7. Cassandra utilizes the data model of Google's Bigtable, i.e., **SSTables** and **MemTables**.

8. Cassandra utilizes distributed features of Amazon's Dynamo, i.e., consistent hashing, partitioning, and replication.
9. Cassandra offers **Tunable consistency** for both read and write operations to adjust the tradeoff between availability and consistency of data.
10. Cassandra uses the **gossip protocol** for inter-node communication.

System design patterns

Here is a summary of system design patterns used in Cassandra:

- **Consistent Hashing:** Cassandra uses Consistent Hashing to distribute its data across nodes.
- **Quorum:** To ensure data consistency, each Cassandra write operation can be configured to be successful only if the data has been written to at least a quorum of replica nodes.
- **Write-Ahead Log:** To ensure durability, whenever a node receives a write request, it immediately writes the data to a commit log which is a write-ahead log.
- **Segmented Log:** Cassandra uses the segmented log strategy to split its commit log into multiple smaller files instead of a single large file for easier operations. As we know, when a node receives a write operation, it immediately writes the data to a commit log. As the commit log grows and reaches its threshold in size, a new commit log is created. Hence, over time several commit logs could be present, each of which is called a segment. Commit log segments reduce the number of seeks needed to write to disk. Commit log segments are truncated when Cassandra has flushed corresponding data to SSTables. Commit log segments can be archived, deleted, or recycled once all its data has been flushed to SSTables.

- **Gossip protocol:** Cassandra uses gossip protocol that allows each node to keep track of state information about the other nodes in the cluster.
- **Generation number:** In Cassandra, each node keeps a generation number which is incremented whenever a node restarts. This generation number is included in gossip messages exchanged between nodes and is used to distinguish the node's current state from its state before a restart.
- **Phi Accrual Failure Detector:** Cassandra uses an adaptive failure detection mechanism as described by the Phi Accrual Failure Detector algorithm. This algorithm, instead of providing a binary output telling if the system is up or down, uses historical heartbeat information to output the suspicion level about a node. A higher suspicion level means there are high chances that the node is down.
- **Bloom filters:** In Cassandra, each SSTable has a Bloom filter associated with it, which tells if a particular key is present in it or not.
- **Hinted Handoff:** Cassandra nodes use Hinted Handoff to remember the write operation for failing nodes.
- **Read Repair:** Cassandra uses 'Read Repair' to push the latest version of the data to nodes with the older versions.

Cassandra characteristics

Here are a few reasons behind Cassandra's performance and popularity:

- **Distributed** means it can run on a large number of machines.
- **Decentralized** means there's no leader-follower paradigm. All nodes are identical and can perform all functions of Cassandra.

- **Scalable** means that Cassandra can be easily scaled horizontally by adding more nodes to the cluster without any performance impact. No manual intervention or rebalancing is required. Cassandra achieves linear scalability and proven fault-tolerance on commodity hardware.
- **Highly Available** means Cassandra is fault-tolerant, and the data remains available even if one or several nodes or data centers go down.
- **Fault-Tolerant and reliable**, as data is replicated to multiple nodes, fault-tolerance is pretty high.
- **Tunable consistency** means that it is possible to adjust the tradeoff between availability and consistency of data on Cassandra nodes, typically by configuring replication factor and consistency level settings.
- **Durable** means Cassandra stores data permanently.
- **Eventually Consistent** as Cassandra favors high availability at the cost of strong consistency.
- **Geographic distribution** means Cassandra supports geographical distribution and efficient data replication across multiple clouds and data centers.

References and further reading

- Bigtable (<https://research.google/pubs/pub27898/>)
- Dynamo (<http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf>)
- Datastax docs (https://docs.datastax.com/en/dse/6.8/dse-arch/datastax_enterprise/dbArch/archTOC.html)
- Tombstones common problems (<https://opencredo.com/blogs/cassandra-tombstones-common-issues/>)
- The Phi Accrual Failure Detector
(<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.7427&rep=rep1&type=pdf>)

- Cassandra introduction video (<https://www.coursera.org/lecture/cloud-applications-part2/2-3-1-cassandra-introduction-olmpu>)

← Back

Tombstones

Next →

Quiz: Cassandra

Kafka: How to Design a Distributed Messaging System?

Messaging Systems: Introduction

This lesson gives a brief overview of messaging systems.

We'll cover the following



- Goal
- Background
- What is a messaging system?
 - Queue
 - Publish-subscribe messaging system

Goal

Design a distributed messaging system that can reliably transfer a high throughput of messages between different entities.

Background

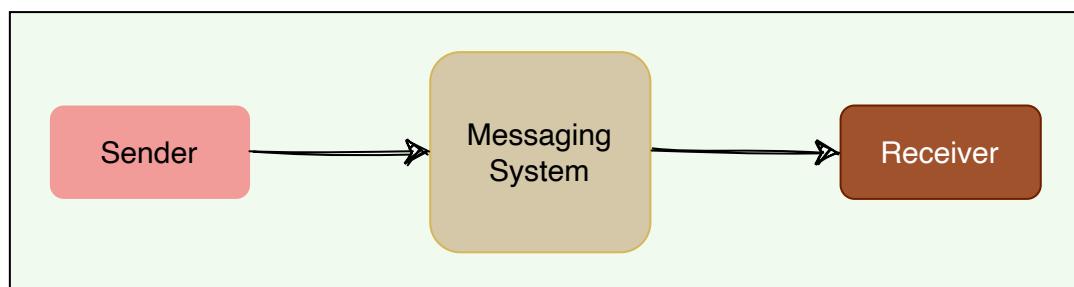
One of the common challenges among distributed systems is handling a continuous influx of data from multiple sources. Imagine a log aggregation service that is receiving hundreds of log entries per second from different sources. The function of this log aggregation service is to store these logs on disk at a shared server and also build an index so that the logs can be searched later. A few challenges of this service are:

1. How will the log aggregation service handle a spike of messages? If the service can handle (or buffer) 500 messages per second, what will happen if it starts receiving a higher number of messages per second? If we decide to have multiple instances of the log aggregation service, how do we divide the work among these instances?
2. How can we receive messages from different types of sources? The sources producing (or consuming) these logs need to decide upon a common protocol and data format to send log messages to the log aggregation service. This leads us to a strongly coupled architecture between the producer and consumer of the log messages.
3. What will happen to the log messages if the log aggregation service is down or unresponsive for some time?

To efficiently manage such scenarios, distributed systems depend upon a messaging system.

What is a messaging system?

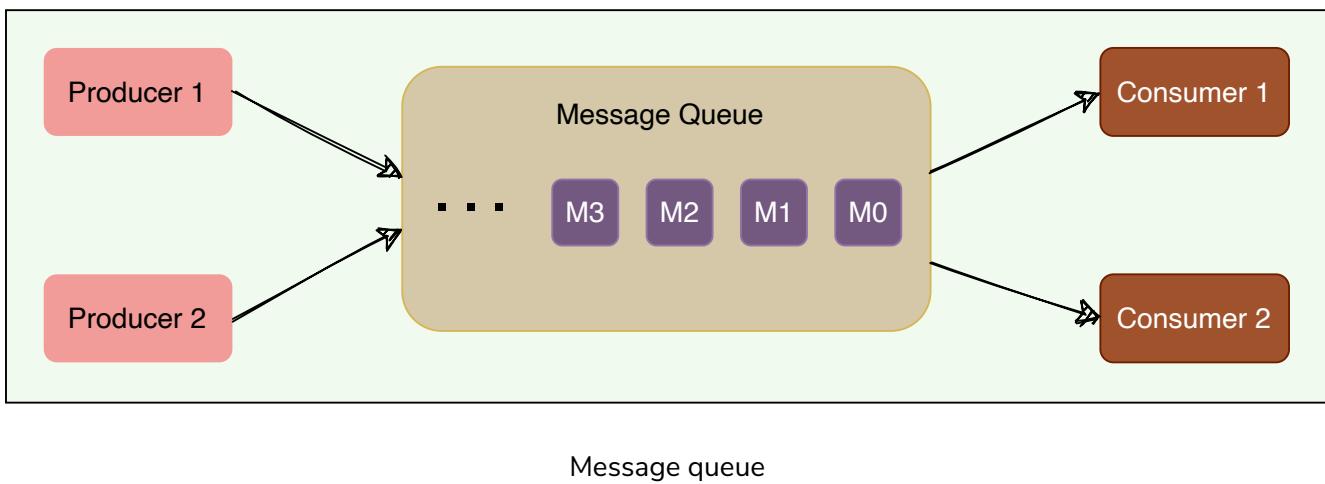
A **messaging system** is responsible for transferring data among services, applications, processes, or servers. Such a system helps **decouple** different parts of a distributed system by providing an **asynchronous** way of transferring messaging between the sender and the receiver. Hence, all senders (or producers) and receivers (or consumers) focus on the data/message without worrying about the mechanism used to share the data.



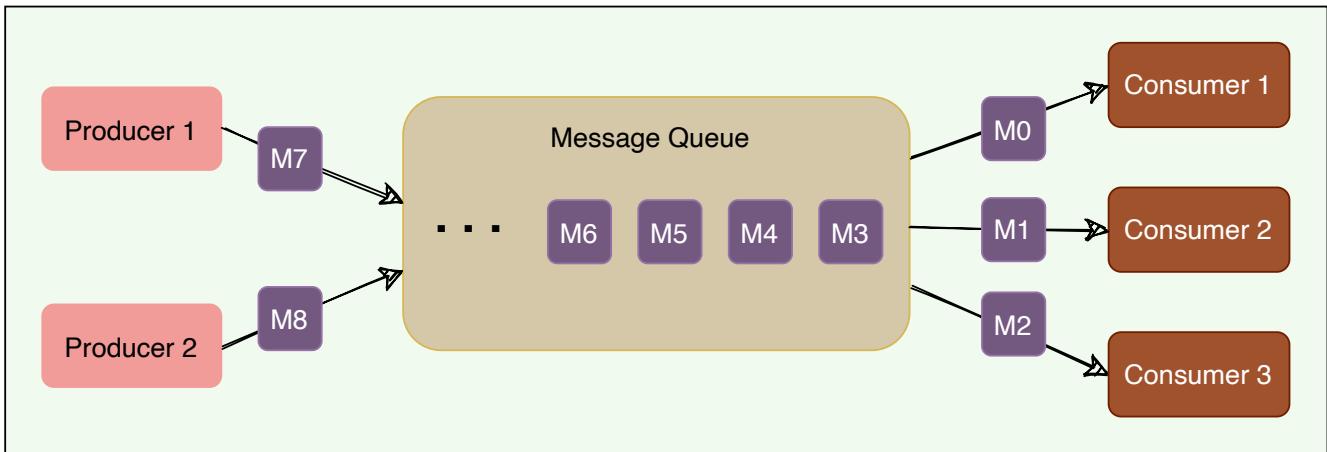
There are two common ways to handle messages: **Queuing** and **Publish-Subscribe**.

Queue

In the queuing model, messages are stored sequentially in a queue. Producers push messages to the rear of the queue, and consumers extract the messages from the front of the queue.



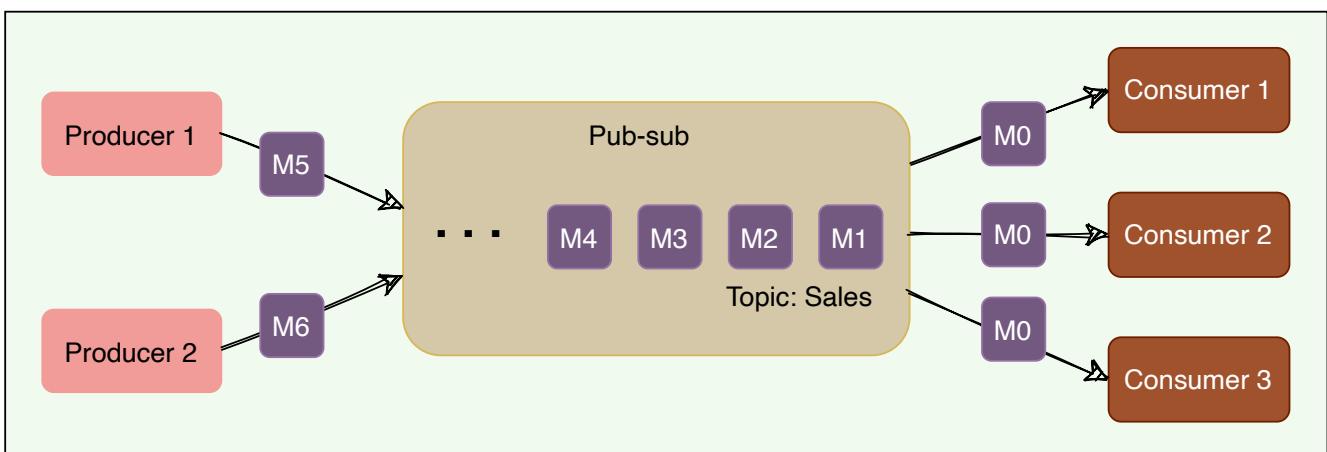
A particular message can be consumed by a maximum of one consumer only. Once a consumer grabs a message, it is removed from the queue such that the next consumer will get the next message. This is a great model for distributing message-processing among multiple consumers. But this also limits the system as multiple consumers cannot read the same message from the queue.



Message consumption in a message queue

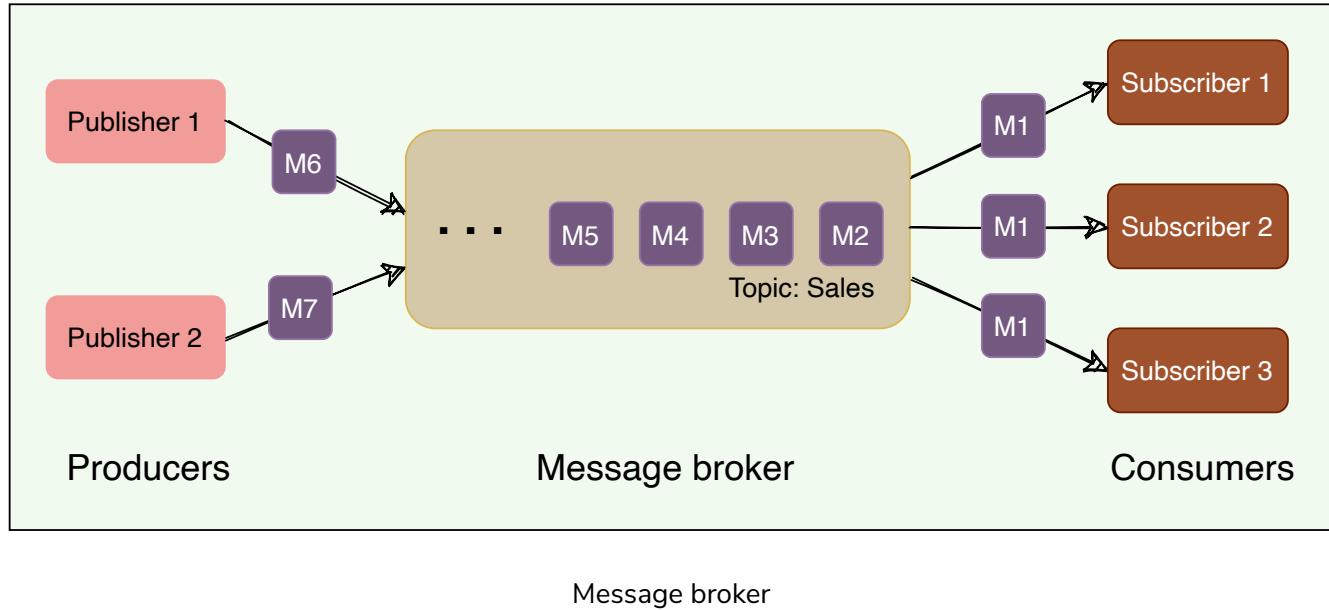
Publish-subscribe messaging system

In the pub-sub (short for publish-subscribe) model, messages are divided into topics. A publisher (or a producer) sends a message to a topic that gets stored in the messaging system under that topic. Subscribers (or the consumer) subscribe to a topic to receive every message published to that topic. Unlike the Queuing model, the pub-sub model allows multiple consumers to get the same message; if two consumers subscribe to the same topic, they will receive all messages published to that topic.



Pub-sub messaging system

The messaging system that stores and maintains the messages is commonly known as the message **broker**. It provides a loose coupling between publishers and subscribers, or producers and consumers of data.



The message broker stores published messages in a queue, and subscribers read them from the queue. Hence, subscribers and publishers do not have to be synchronized. This **loose coupling** enables subscribers and publishers to read and write messages at different rates.

The messaging system's ability to store messages provides **fault-tolerance**, so messages do not get lost between the time they are produced and the time they are consumed.

To summarize, a message system is deployed in an application stack for the following reasons:

1. **Messaging buffering.** To provide a buffering mechanism in front of processing (i.e., to deal with temporary incoming message spikes that are greater than what the processing app can deal with). This enables the system to safely deal with spikes in workloads by temporarily storing data until it is ready for processing.

2. **Guarantee of message delivery.** Allows producers to publish messages with assurance that the message will eventually be delivered if the consuming application is unable to receive the message when it is published.
3. **Providing abstraction.** A messaging system provides an architectural separation between the consumers of messages and the applications producing the messages.
4. **Enabling scale.** Provides a flexible and highly configurable architecture that enables many producers to deliver messages to multiple consumers.

[!\[\]\(aa575729278afb3b4588fb8f2546baa4_img.jpg\) Back](#)

[!\[\]\(53e5eaf6c60a36c126588b975a7f59a2_img.jpg\) Next](#)

Mock Interview: Cassandra

Kafka: Introduction

Kafka: Introduction

This lesson presents a brief introduction and common use cases of Kafka.

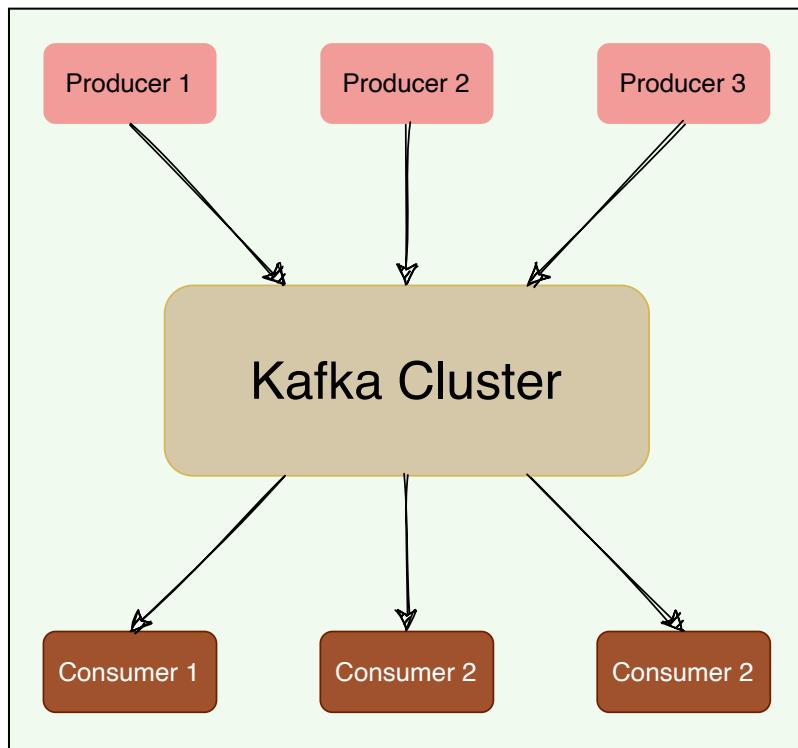
We'll cover the following



- What is Kafka?
- Background
- Kafka use cases

What is Kafka?

Apache Kafka is an open-source **publish-subscribe**-based messaging system (*Kafka can work as a message queue too, more on this later*). It is **distributed**, **durable**, **fault-tolerant**, and **highly scalable** by design. Fundamentally, it is a system that takes streams of messages from applications known as producers, stores them reliably on a central cluster (containing a set of brokers), and allows those messages to be received by applications (known as consumers) that process the messages.



A high-level view of Kafka

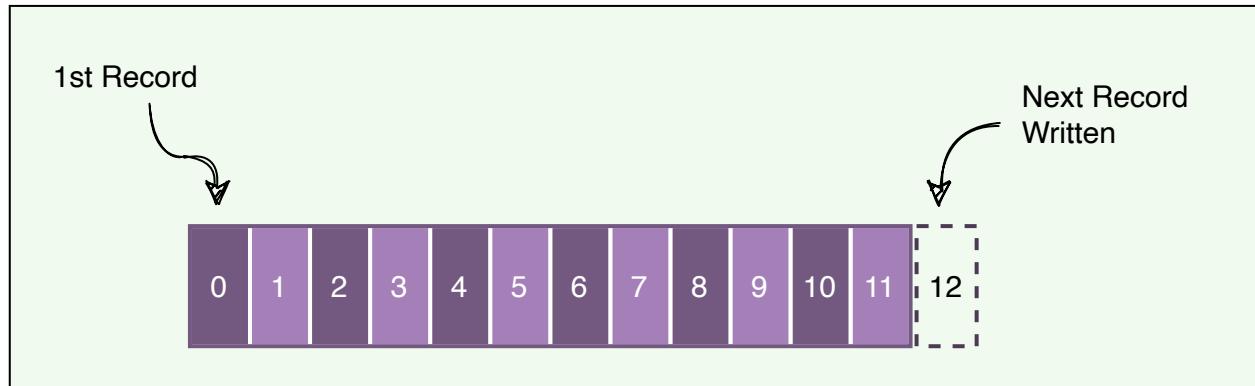
Background

Kafka was created at LinkedIn around 2010 to track various events, such as page views, messages from the messaging system, and logs from various services. Later, it was made open-source and developed into a comprehensive system which is used for:

1. Reliably storing a huge amount of data.
2. Enabling high throughput of message transfer between different entities.
3. Streaming real-time data.

At a high level, we can call Kafka a distributed **Commit Log**. A Commit Log (*also known as a Write-Ahead log or a Transactions log*) is an **append-only** data structure that can persistently store a sequence of records. Records are

always appended to the end of the log, and once added, records cannot be deleted or modified. Reading from a commit log always happens from left to right (*or old to new*).



Kafka as a write-ahead log

Kafka stores all of its messages on disk. Since all reads and writes happen in sequence, Kafka takes advantage of sequential disk reads (*more on this later*).

Kafka use cases

Kafka can be used for collecting big data and real-time analysis. Here are some of its top use cases:

1. **Metrics:** Kafka can be used to collect and aggregate monitoring data. Distributed services can push different operational metrics to Kafka servers. These metrics can then be pulled from Kafka to produce aggregated statistics.
2. **Log Aggregation:** Kafka can be used to collect logs from multiple sources and make them available in a standard format to multiple consumers.
3. **Stream processing:** Kafka is quite useful for use cases where the collected data undergoes processing at multiple stages. For example, the

raw data consumed from a topic is transformed, enriched, or aggregated and pushed to a new topic for further consumption. This way of data processing is known as stream processing.

4. **Commit Log:** Kafka can be used as an external commit log for any distributed system. Distributed services can log their transactions to Kafka to keep track of what is happening. This transaction data can be used for replication between nodes and also becomes very useful for disaster recovery, for example, to help failed nodes to recover their states.
5. **Website activity tracking:** One of Kafka's original use cases was to build a user activity tracking pipeline. User activities like page clicks, searches, etc., are published to Kafka into separate topics. These topics are available for subscription for a range of use cases, including real-time processing, real-time monitoring, or loading into Hadoop (<https://hadoop.apache.org/>) or data warehousing systems for offline processing and reporting.
6. **Product suggestions:** Imagine an online shopping site like amazon.com ([http://amazon.com/](http://amazon.com)), which offers a feature of 'similar products' to suggest lookalike products that a customer could be interested in buying. To make this work, we can track every consumer action, like search queries, product clicks, time spent on any product, etc., and record these activities in Kafka. Then, a consumer application can read these messages to find correlated products that can be shown to the customer in real-time. Alternatively, since all data is persistent in Kafka, a batch job can run overnight on the 'similar product' information gathered by the system, generating an email for the customer with product suggestions.

← Back

Next →

High-level Architecture

This lesson gives a brief overview of Kafka's architecture.

We'll cover the following



- Kafka common terms
 - Brokers
 - Records
 - Topics
 - Producers
 - Consumers
- High-level architecture
 - Kafka cluster
 - ZooKeeper

Kafka common terms

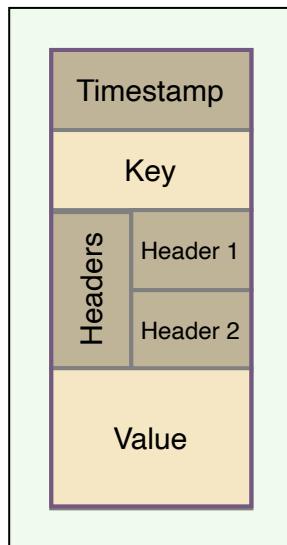
Before digging deep into Kafka's architecture, let's first go through some of its common terms.

Brokers

A Kafka server is also called a broker. Brokers are responsible for reliably storing data provided by the producers and making it available to the consumers.

Records

A record is a message or an event that gets stored in Kafka. Essentially, it is the data that travels from producer to consumer through Kafka. A record contains a key, a value, a timestamp, and optional metadata headers.



Kafka message

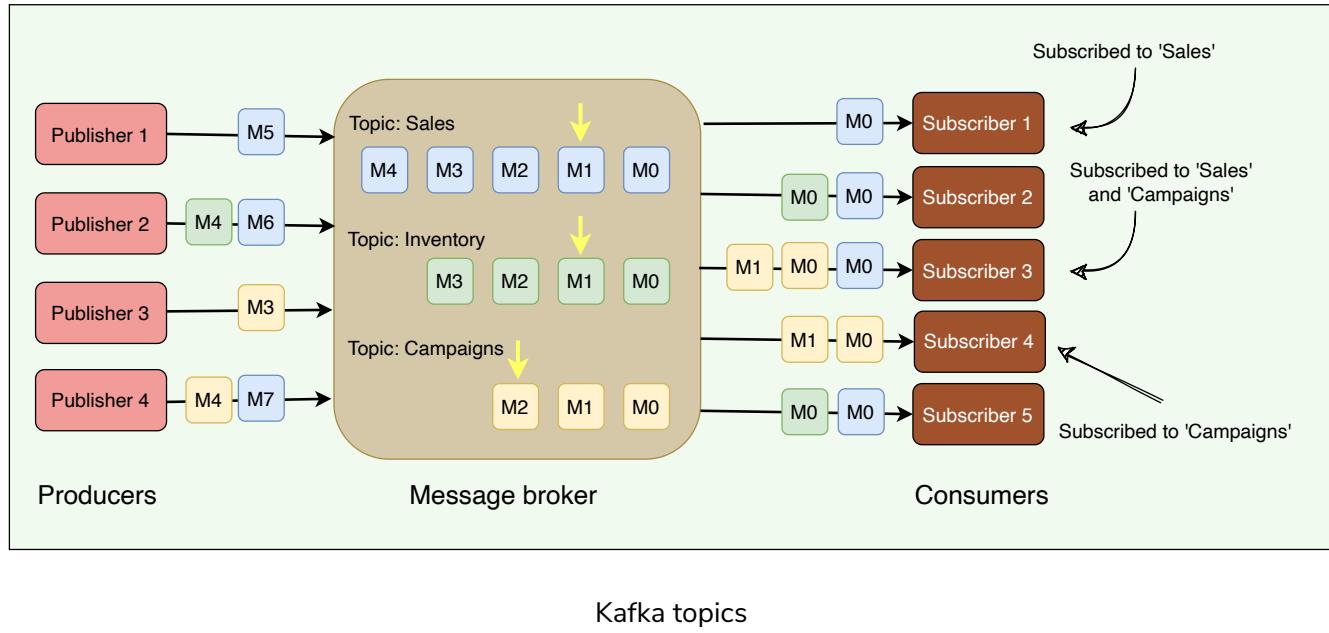
Topics

Kafka divides its messages into categories called Topics. In simple terms, a topic is like a table in a database, and the messages are the rows in that table.

- Each message that Kafka receives from a producer is associated with a topic.
- Consumers can subscribe to a topic to get notified when new messages are added to that topic.
- A topic can have multiple subscribers that read messages from it.
- In a Kafka cluster, a topic is identified by its name and must be unique.

Messages in a topic can be read as often as needed — unlike traditional messaging systems, messages are not deleted after consumption. Instead, Kafka retains messages for a configurable amount of time or until a storage

size is exceeded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.



Producers

Producers are applications that publish (or write) records to Kafka.

Consumers

Consumers are the applications that subscribe to (read and process) data from Kafka topics. Consumers subscribe to one or more topics and consume published messages by pulling data from the brokers.

In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers.

High-level architecture

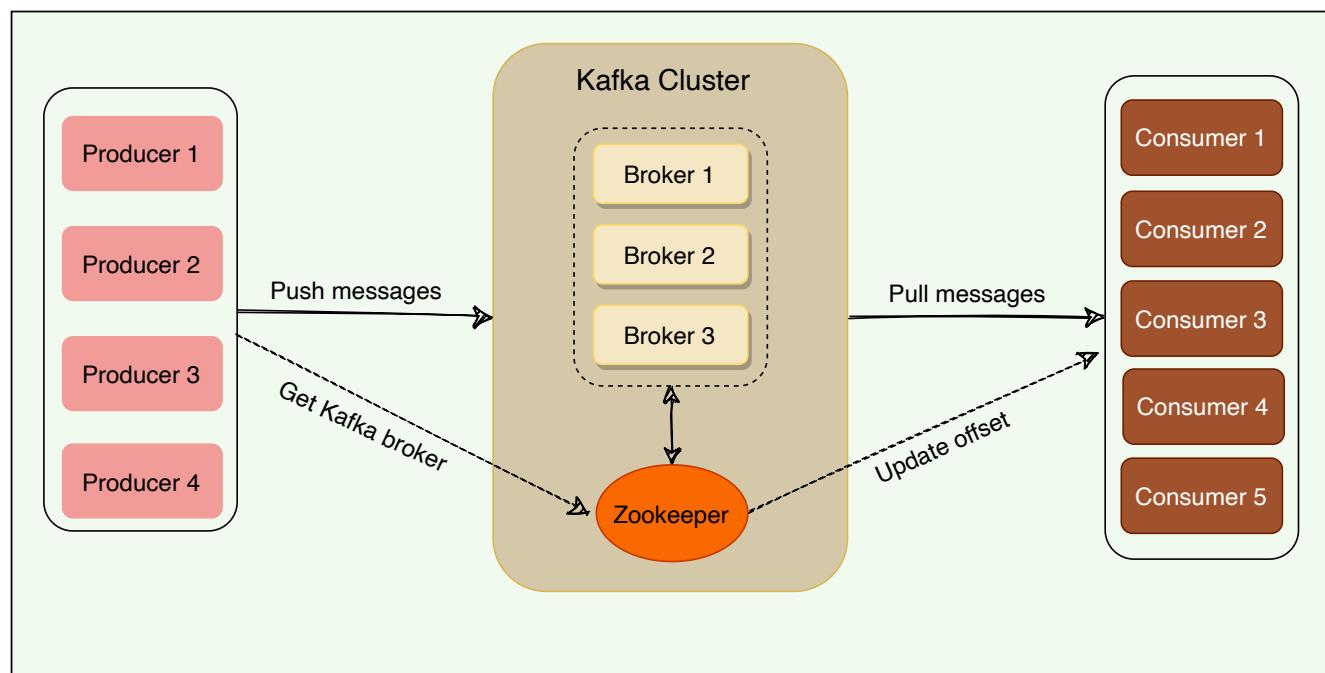
At a high level, applications (producers) send messages to a Kafka broker, and these messages are read by other applications called consumers. Messages get stored in a topic, and consumers subscribe to the topic to receive new messages.

Kafka cluster

Kafka is run as a cluster of one or more servers, where each server is responsible for running one Kafka broker.

ZooKeeper

ZooKeeper is a distributed key-value store and is used for coordination and storing configurations. It is highly optimized for reads. Kafka uses ZooKeeper to coordinate between Kafka brokers; ZooKeeper maintains metadata information about the Kafka cluster. We will be looking into this in detail later.



High level architecture of Kafka

Kafka: Deep Dive

As of now, we have discussed the core concepts of Kafka. Let us now throw some light on the workflow of Kafka.

We'll cover the following



- Topic partitions
 - Leader
 - Follower
 - In-sync replicas
- High-water mark

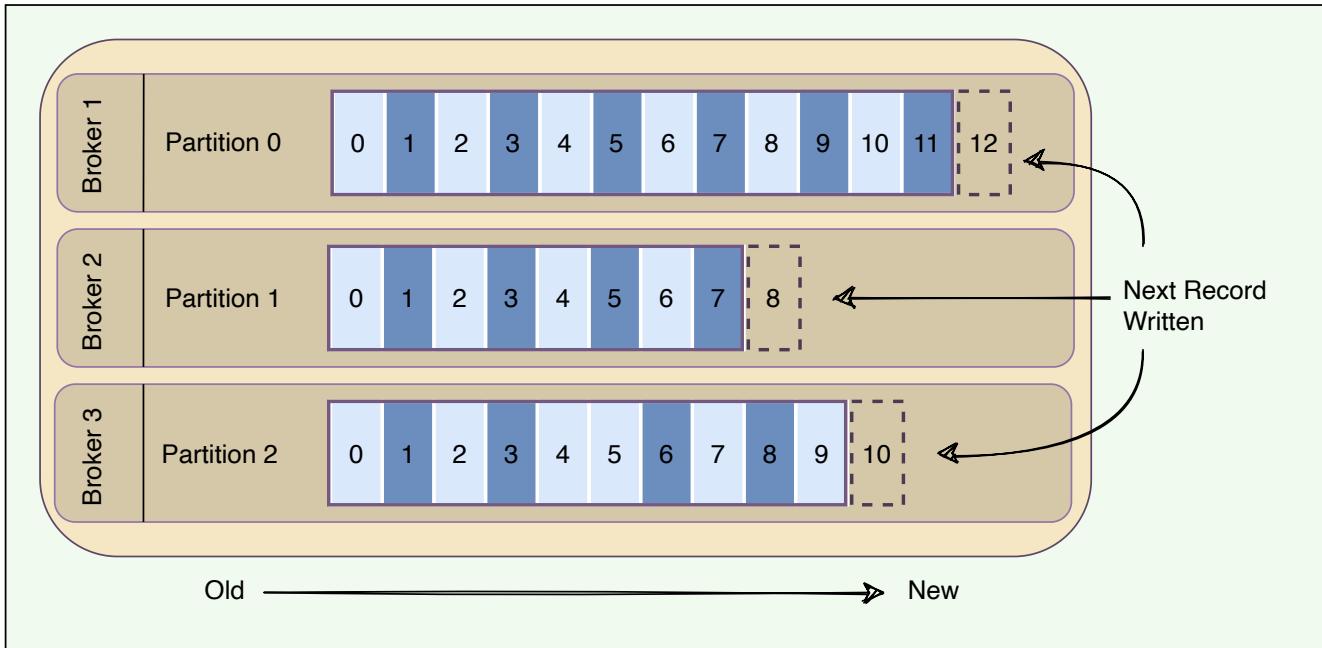
Kafka is simply a **collection of topics**. As topics can get quite big, they are **split into partitions** of a smaller size for better performance and scalability.

Topic partitions

Kafka topics are partitioned, meaning a topic is spread over a number of ‘fragments’. Each partition can be placed on a separate Kafka broker. When a new message is published on a topic, it gets appended to one of the topic’s partitions. The producer controls which partition it publishes messages to based on the data. For example, a producer can decide that all messages related to a particular ‘city’ go to the same partition.

Essentially, a partition is an ordered sequence of messages. Producers continually append new messages to partitions. Kafka guarantees that all messages inside a partition are stored in the sequence they came in.

Ordering of messages is maintained at the partition level, not across the topic.



A topic having three partitions residing on three brokers

- A unique sequence ID called an **offset** gets assigned to every message that enters a partition. These numerical offsets are used to identify every message's sequential position within a topic's partition.
- Offset sequences are unique only to each partition. This means, to locate a specific message, we need to know the Topic, Partition, and Offset number.
- Producers can choose to publish a message to any partition. If ordering within a partition is not needed, a round-robin partition strategy can be used, so records get distributed evenly across partitions.
- Placing each partition on separate Kafka brokers enables multiple consumers to read from a topic in parallel. That means, different consumers can concurrently read different partitions present on separate brokers.
- Placing each partition of a topic on a separate broker also enables a topic to hold more data than the capacity of one server.

- Messages once written to partitions are **immutable** and cannot be updated.
- A producer can add a ‘**key**’ to any message it publishes. Kafka guarantees that messages with the same key are written to the same partition.
- Each broker manages a set of partitions belonging to different topics.

Kafka follows the principle of a **dumb broker** and **smart consumer**. This means that Kafka does not keep track of what records are read by the consumer. Instead, consumers, themselves, poll Kafka for new messages and say what records they want to read. This allows them to increment/decrement the offset they are at as they wish, thus being able to replay and reprocess messages. Consumers can read messages starting from a specific offset and are allowed to read from any offset they choose. This also enables consumers to join the cluster at any point in time.

Every topic can be replicated to multiple Kafka brokers to make the data fault-tolerant and highly available. Each topic partition has one leader broker and multiple replica (follower) brokers.

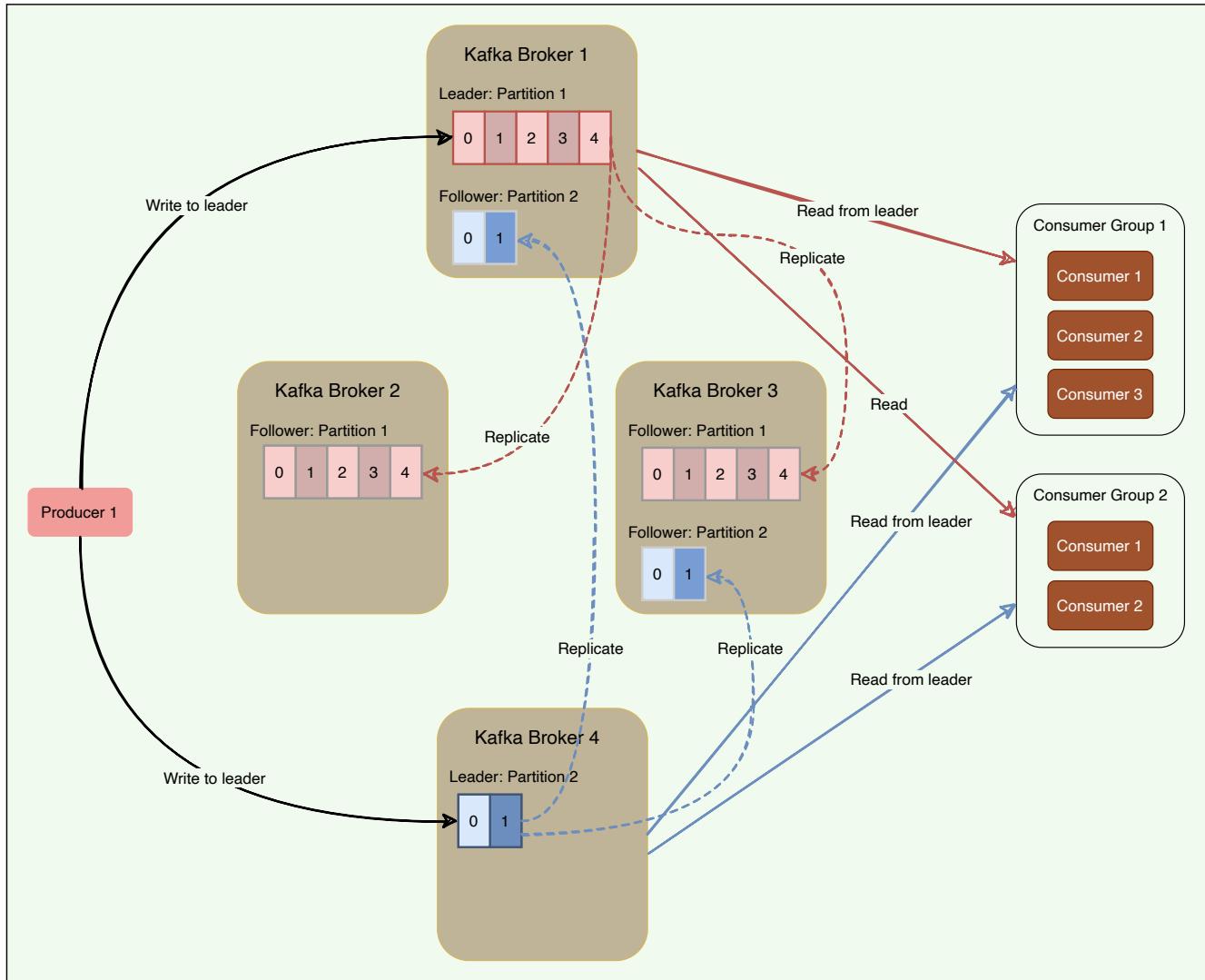
Leader

A leader is the node responsible for all reads and writes for the given partition. Every partition has one Kafka broker acting as a leader.

Follower

To handle single point of failure, Kafka can replicate partitions and distribute them across multiple broker servers called followers. Each follower’s responsibility is to replicate the leader’s data to serve as a ‘backup’ partition. This also means that any follower can take over the leadership if the leader goes down.

In the following diagram, we have two partitions and four brokers. Broker 1 is the leader of Partition 1 and follower of Partition 2. Consumers work together in groups to process messages efficiently. More details on consumer groups later.



Leader and followers of partitions

Kafka stores the location of the leader of each partition in ZooKeeper. As all writes/reads happen at/from the leader, producers and consumers directly talk to ZooKeeper to find a partition leader.

In-sync replicas

An in-sync replica (ISR) is a broker that has the latest data for a given partition. A leader is always an in-sync replica. A follower is an in-sync replica only if it has fully caught up to the partition it is following. In other words, ISRs cannot be behind on the latest records for a given partition.

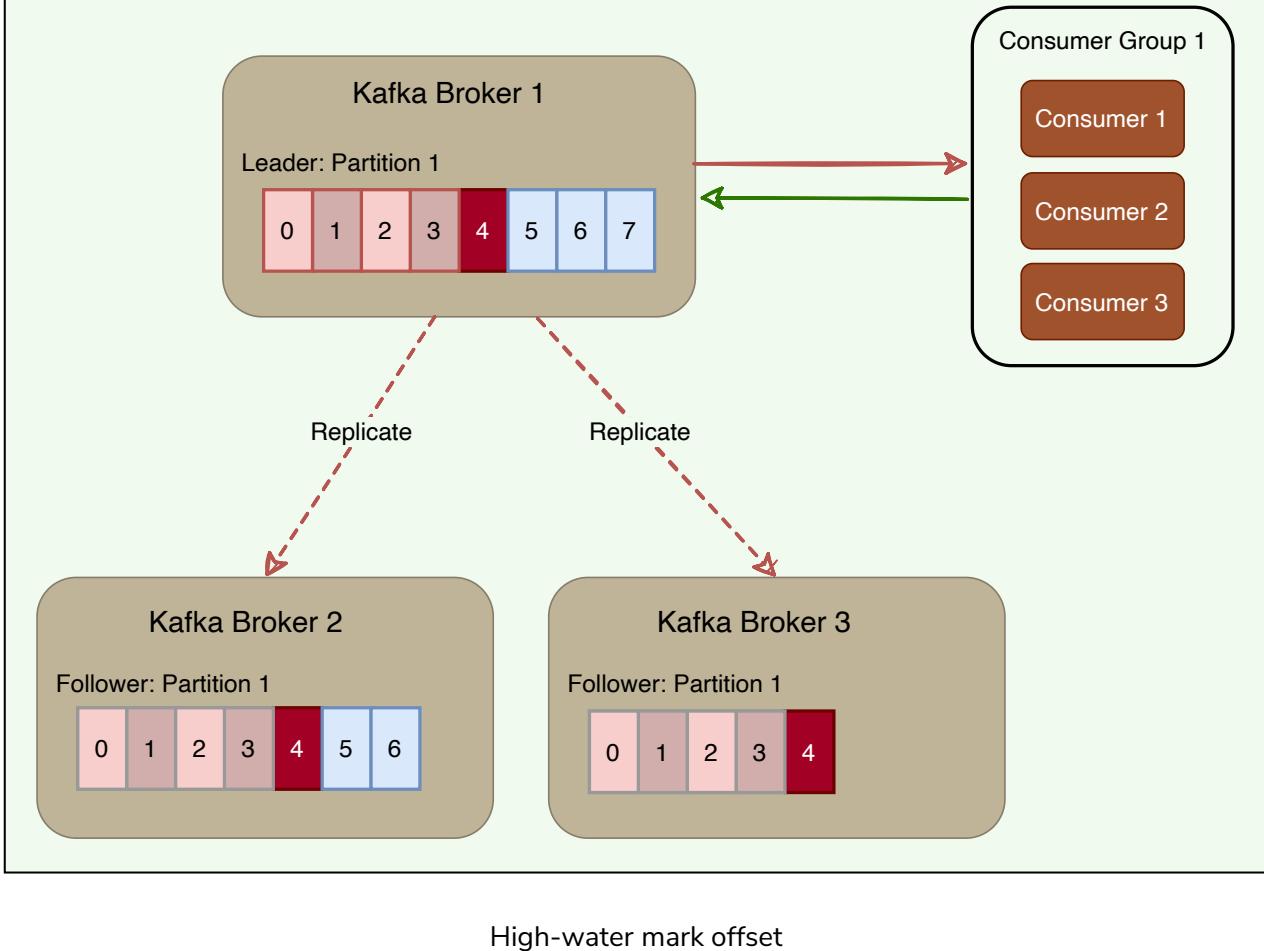
Only ISRs are eligible to become partition leaders. Kafka can choose the minimum number of ISRs required before the data becomes available for consumers to read.

High-water mark

To ensure data consistency, the leader broker never returns (or exposes) messages which have not been replicated to a minimum set of ISRs. For this, brokers keep track of the high-water mark, which is the highest offset that all ISRs of a particular partition share. The leader exposes data only up to the high-water mark offset and propagates the high-water mark offset to all followers. Let's understand this with an example.

In the figure below, the leader does not return messages greater than offset '4', as it is the highest offset message that has been replicated to all follower brokers.

High-water mark offset is 4



If a consumer reads the record with offset '7' from the leader (*Broker 1*), and later, if the current leader fails, and one of the followers becomes the leader before the record is replicated to the followers, the consumer will not be able to find that message on the new leader. The client, in this case, will experience a non-repeatable read. Because of this possibility, Kafka brokers only return records up to the high-water mark.

← Back

Next →

High-level Architecture

Consumer Groups

Consumer Groups

Let's explore the role of consumer groups in Kafka.

We'll cover the following



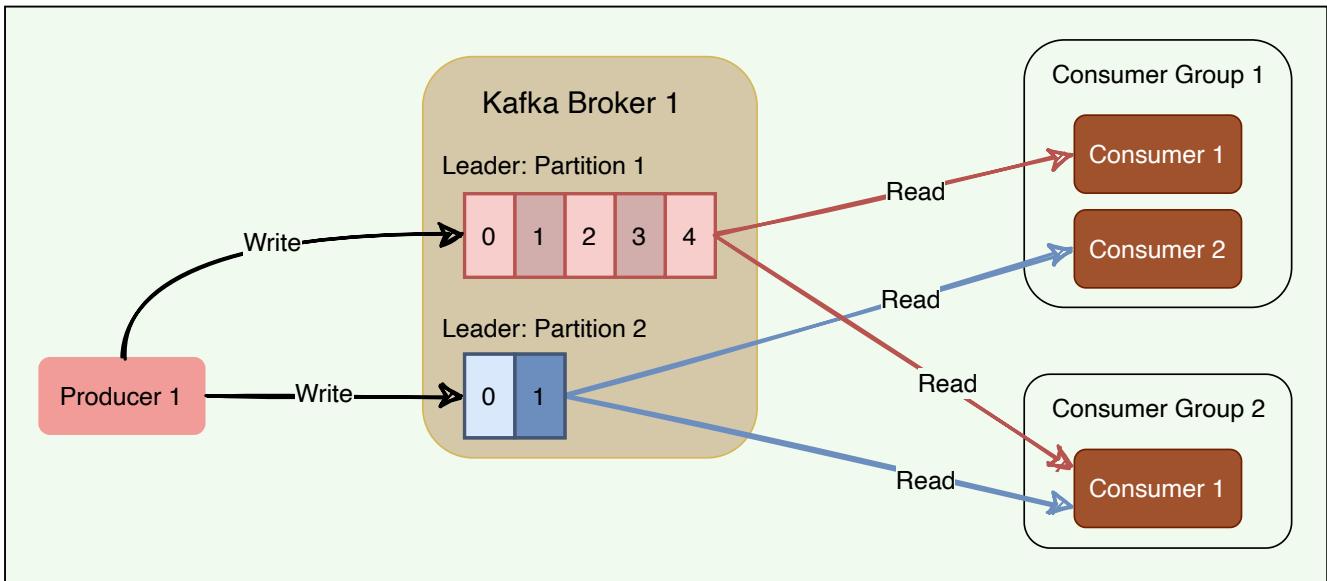
- What is a consumer group?
- Distributing partitions to consumers within a consumer group

What is a consumer group?

A consumer group is basically a set of one or more consumers working together in parallel to consume messages from topic partitions. Messages are equally divided among all the consumers of a group, with no two consumers receiving the same message.

Distributing partitions to consumers within a consumer group

Kafka ensures that **only a single consumer reads messages from any partition within a consumer group**. In other words, topic partitions are a unit of parallelism – only one consumer can work on a partition in a consumer group at a time. If a consumer stops, Kafka spreads partitions across the remaining consumers in the same consumer group. Similarly, every time a consumer is added to or removed from a group, the consumption is rebalanced within the group.



How Kafka distributes partitions to consumers within a consumer group

Consumers pull messages from topic partitions. Different consumers can be responsible for different partitions. Kafka can support a large number of consumers and retain large amounts of data with very little overhead. By using consumer groups, consumers can be parallelized so that multiple consumers can read from multiple partitions on a topic, allowing a very high message processing throughput. The number of partitions impacts consumers' maximum parallelism, as there cannot be more consumers than partitions.

Kafka stores the current offset per consumer group per topic per partition, as it would for a single consumer. This means that unique messages are only sent to a single consumer in a consumer group, and the load is balanced across consumers as equally as possible.

When the number of consumers exceeds the number of partitions in a topic, all new consumers wait in idle mode until an existing consumer unsubscribes from that partition. Similarly, as new consumers join a consumer group, Kafka initiates a rebalancing if there are more consumers than partitions. Kafka uses any unused consumers as failovers.

Here is a summary of how Kafka manages the distribution of partitions to consumers within a consumer group:

- **Number of consumers in a group = number of partitions:** each consumer consumes one partition.
- **Number of consumers in a group > number of partitions:** some consumers will be idle.
- **Number of consumers in a group < number of partitions:** some consumers will consume more partitions than others.

[!\[\]\(48ede73b67a7785777f47d61f346cc47_img.jpg\) Back](#)

Kafka: Deep Dive

[!\[\]\(ffcc3f412e0ab9c172454deb22125051_img.jpg\) Next](#)

Kafka Workflow

Kafka Workflow

Let's explore different messaging workflows that Kafka offers.

We'll cover the following



- Kafka workflow as pub-sub messaging
- Kafka workflow for consumer group

Kafka provides both pub-sub and queue-based messaging systems in a fast, reliable, persisted, fault-tolerance, and zero downtime manner. In both cases, producers simply send the message to a topic, and consumers can choose any one type of messaging system depending on their need. Let us follow the steps in the next section, to understand how the consumer can choose the messaging system of their choice.

Kafka workflow as pub-sub messaging

Following is the stepwise workflow of the Pub-Sub Messaging:

- Producers publish messages on a topic.
- Kafka broker stores messages in the partitions configured for that particular topic. If the producer did not specify the partition in which the message should be stored, the broker ensures that the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.

- Consumer subscribes to a specific topic.
- Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves that offset in the ZooKeeper.
- Consumer will request Kafka at regular intervals for new messages.
- Once Kafka receives the messages from producers, it forwards these messages to the consumer.
- Consumer will receive the message and process it.
- Once the messages are processed, the consumer will send an acknowledgment to the Kafka broker.
- Upon receiving the acknowledgment, Kafka increments the offset and updates it in the ZooKeeper. Since offsets are maintained in the ZooKeeper, the consumer can read the next message correctly, even during broker outages.
- The above flow will repeat until the consumer stops sending the request.
- Consumers can rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Kafka workflow for consumer group

Instead of a single consumer, a group of consumers from one consumer group subscribes to a topic, and the messages are shared among them. Let us check the workflow of this system:

- Producers publish messages on a topic.
- Kafka stores all messages in the partitions configured for that particular topic, similar to the earlier scenario.
- A single consumer subscribes to a specific topic, assume Topic-01 with Group ID as Group-1 .

- Kafka interacts with the consumer in the same way as pub-sub messaging until a new consumer subscribes to the same topic, Topic-01 , with the same Group ID as Group-1 .
- Once the new consumer arrives, Kafka switches its operation to share mode, such that each message is passed to only one of the subscribers of the consumer group Group-1 . This message transfer is similar to queue-based messaging, as only one consumer of the group consumes a message. Contrary to queue-based messaging, messages are not removed after consumption.
- This message transfer can go on until the number of consumers reaches the number of partitions configured for that particular topic.
- Once the number of consumers exceeds the number of partitions, the new consumer will not receive any message until an existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition. Once all the partitions are assigned to the existing consumers, the new consumers will have to wait.

[!\[\]\(118f60eda6d7067cf1bf5800e46358a1_img.jpg\) Back](#)

Consumer Groups

[!\[\]\(e9a3d6d84f92bc951f654bb385f61a60_img.jpg\) Next](#)

Role of ZooKeeper

Role of ZooKeeper

Let's delve into how Kafka interacts with ZooKeeper.

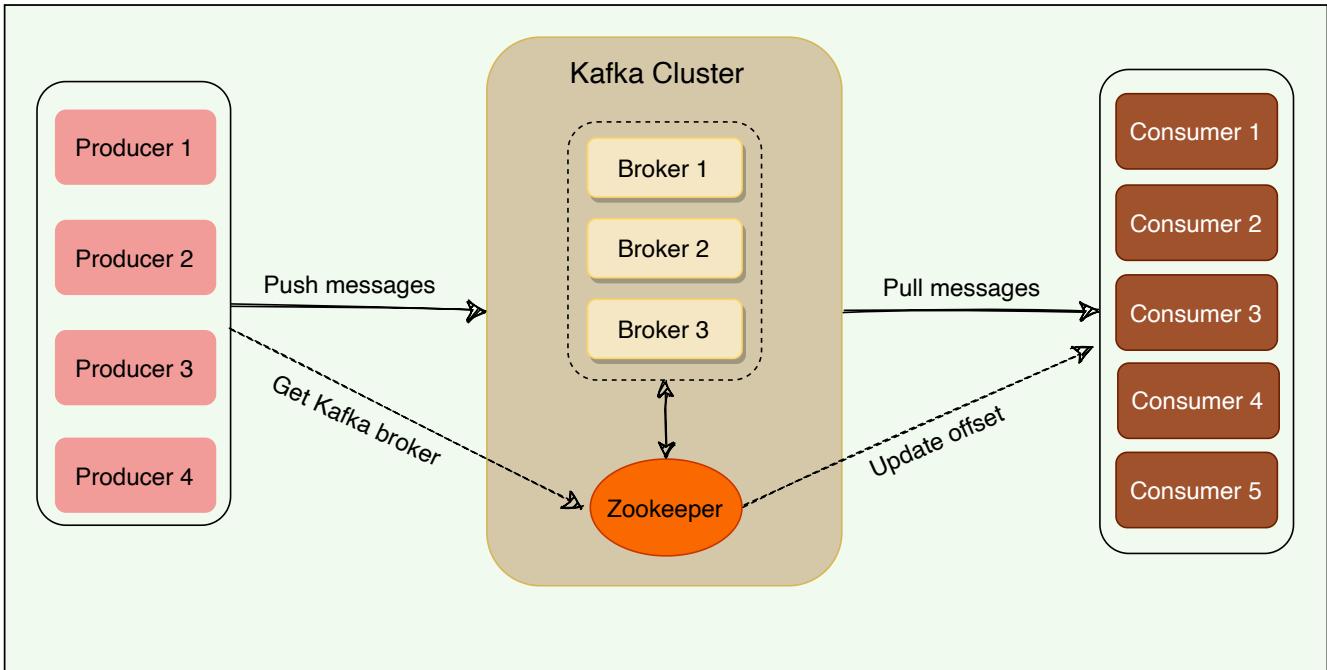
We'll cover the following



- What is ZooKeeper?
- ZooKeeper as the central coordinator
 - How do producers or consumers find out who the leader of a partition is?

What is ZooKeeper?

A critical dependency of Apache Kafka is Apache ZooKeeper, which is a distributed configuration and synchronization service. ZooKeeper serves as the coordination interface between the Kafka brokers, producers, and consumers. Kafka stores basic metadata in ZooKeeper, such as information about brokers, topics, partitions, partition leader/followers, consumer offsets, etc.



ZooKeeper as the central coordinator

#

As we know, Kafka brokers are stateless; they rely on ZooKeeper to maintain and coordinate brokers, such as notifying consumers and producers of the arrival of a new broker or failure of an existing broker, as well as routing all requests to partition leaders.

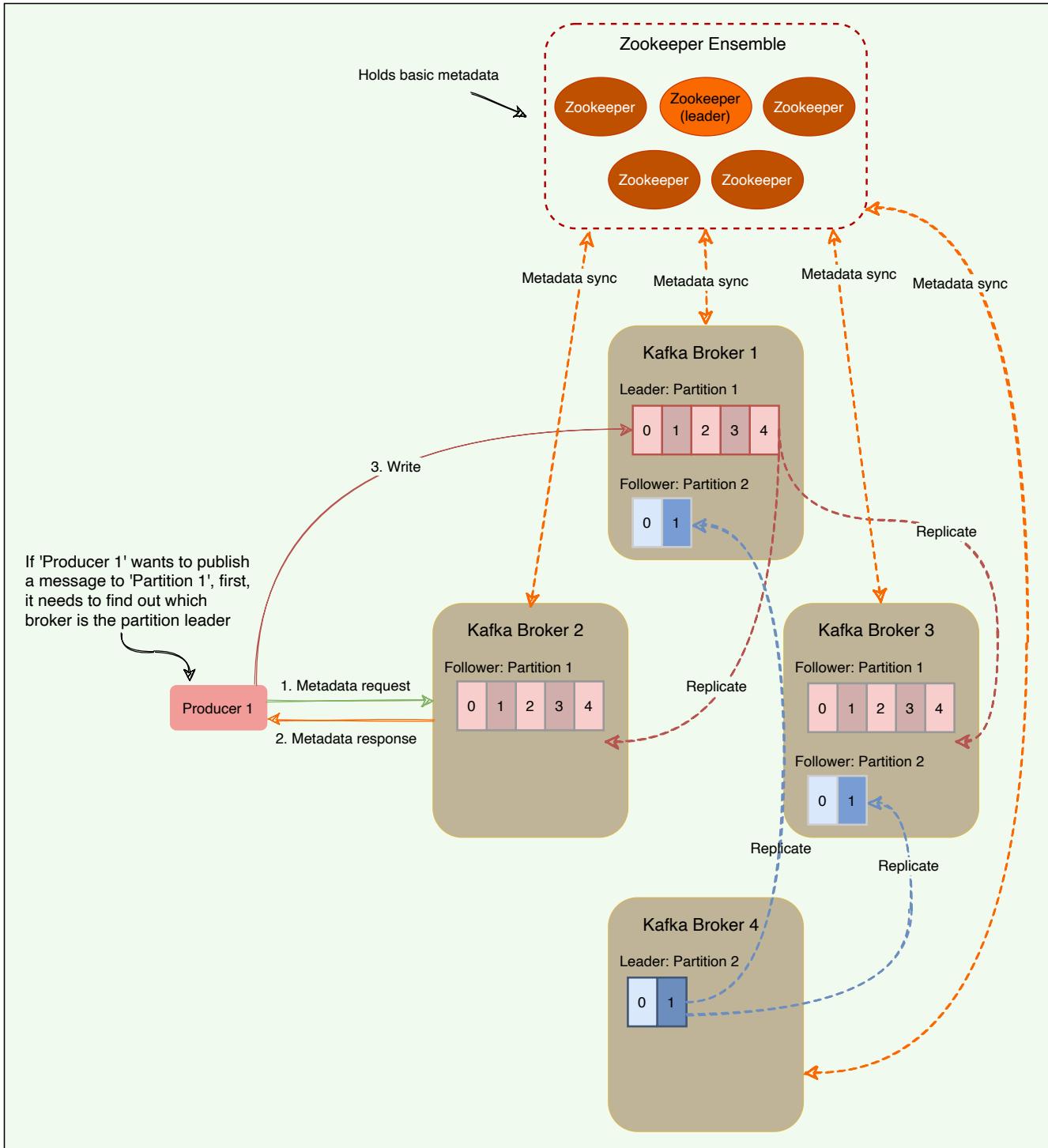
ZooKeeper is used for storing all sorts of metadata about the Kafka cluster:

- It maintains the last offset position of each consumer group per partition, so that consumers can quickly recover from the last position in case of a failure (although modern clients store offsets in a separate Kafka topic).
- It tracks the topics, number of partitions assigned to those topics, and leaders'/followers' location in each partition.
- It also manages the access control lists (ACLs) to different topics in the cluster. ACLs are used to enforce access or authorization.

How do producers or consumers find out who the leader of a partition is?

In the older versions of Kafka, all clients (i.e., producers and consumers) used to directly talk to ZooKeeper to find the partition leader. Kafka has moved away from this coupling, and in Kafka's latest releases, clients fetch metadata information from Kafka brokers directly; brokers talk to ZooKeeper to get the latest metadata. In the diagram below, the producer goes through the following steps before publishing a message:

1. The producer connects to any broker and asks for the leader of 'Partition 1'.
2. The broker responds with the identification of the leader broker responsible for 'Partition 1'.
3. The producer connects to the leader broker to publish the message.



Role of ZooKeeper in Kafka

All the critical information is stored in the ZooKeeper and ZooKeeper replicates this data across its cluster, therefore, failure of Kafka broker (or ZooKeeper itself) does not affect the state of the Kafka cluster. Upon ZooKeeper failure, Kafka will always be able to restore the state once the

ZooKeeper restarts after failure. Zookeeper is also responsible for coordinating the partition leader election between the Kafka brokers in case of leader failure.

[← Back](#)

Kafka Workflow

[Next →](#)

Controller Broker

Controller Broker

This lesson will explain the role of the controller broker in Kafka.

We'll cover the following



- What is the controller broker?
- Split brain
- Generation clock

What is the controller broker?

Within the Kafka cluster, one broker is elected as the Controller. This Controller broker is responsible for admin operations, such as creating/deleting a topic, adding partitions, assigning leaders to partitions, monitoring broker failures, etc. Furthermore, the Controller periodically checks the health of other brokers in the system. In case it does not receive a response from a particular broker, it performs a failover to another broker. It also communicates the result of the partition leader election to other brokers in the system.

Split brain

When a controller broker dies, Kafka elects a new controller. One of the problems is that we cannot truly know if the leader has stopped for good and has experienced an intermittent failure like a stop-the-world GC pause or a

temporary network disruption. Nevertheless, the cluster has to move on and pick a new controller. If the original Controller had an intermittent failure, the cluster would end up having a so-called **zombie controller**. A zombie controller can be defined as a controller node that had been previously deemed dead by the cluster and has come back online. Another broker has taken its place, but the zombie controller might not know that yet. This common scenario in distributed systems with two or more active controllers (or central servers) is called split-brain.

We will have two controllers under split-brain, which will be giving out potentially conflicting commands in parallel. If something like this happens in a cluster, it can result in major inconsistencies. How do we handle this situation?

Generation clock

Split-brain is commonly solved with a **generation clock**, which is simply a monotonically increasing number to indicate a server's generation. In Kafka, the generation clock is implemented through an epoch number. If the old leader had an epoch number of '1', the new one would have '2'. This epoch is included in every request that is sent from the Controller to other brokers. This way, brokers can now easily differentiate the real Controller by simply trusting the Controller with the highest number. The Controller with the highest number is undoubtedly the latest one, since the epoch number is always increasing. This epoch number is stored in ZooKeeper.

← Back

Next →

Kafka Delivery Semantics

Let's explore what message delivery guarantees Kafka provides to its clients.

We'll cover the following



- Producer delivery semantics
- Consumer delivery semantics

Producer delivery semantics

As we know, a producer writes only to the leader broker, and the followers asynchronously replicate the data. How can a producer know that the data is successfully stored at the leader or that the followers are keeping up with the leader? Kafka offers three options to denote the number of brokers that must receive the record before the producer considers the write as successful:

- **Async:** Producer sends a message to Kafka and does not wait for acknowledgment from the server. This means that the write is considered successful the moment the request is sent out. This **fire-and-forget** approach gives the best performance as we can write data to Kafka at network speed, but no guarantee can be made that the server has received the record in this case.
- **Committed to Leader:** Producer waits for an acknowledgment from the leader. This ensures that the data is committed at the leader; it will be slower than the 'Async' option, as the data has to be written on disk on the leader. Under this scenario, the leader will respond without waiting for acknowledgments from the followers. In this case, the record will be

lost if the leader crashes immediately after acknowledging the producer but before the followers have replicated it.

- **Committed to Leader and Quorum:** Producer waits for an acknowledgment from the leader and the quorum. This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This will be the slowest write but guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.

As we can see, the above options enable us to configure our preferred trade-off between durability and performance.

- If we would like to be sure that our records are safely stored in Kafka, we have to go with the last option – Committed to Leader and Quorum.
- If we value latency and throughput more than durability, we can choose one of the first two options. These options will have a greater chance of losing messages but will have better speed and throughput.

Consumer delivery semantics

A consumer can read only those messages that have been written to a set of in-sync replicas. There are three ways of providing consistency to the consumer:

- **At-most-once** (Messages may be lost but are never redelivered): In this option, a message is delivered a maximum of one time only. Under this option, the consumer upon receiving a message, commit (or increment) the offset to the broker. Now, if the consumer crashes before fully consuming the message, that message will be lost, as when the consumer restarts, it will receive the next message from the last committed offset.

- **At-least-once** (Messages are never lost but maybe redelivered): Under this option, a message might be delivered more than once, but no message should be lost. This scenario occurs when the consumer receives a message from Kafka, and it does not immediately commit the offset. Instead, it waits till it completes the processing. So, if the consumer crashes after processing the message but before committing the offset, it has to reread the message upon restart. Since, in this case, the consumer never committed the offset to the broker, the broker will redeliver the same message. Thus, duplicate message delivery could happen in such a scenario.
- **Exactly-once** (each message is delivered once and only once): It is very hard to achieve this unless the consumer is working with a transactional system. Under this option, the consumer puts the message processing and the offset increment in one transaction. This will ensure that the offset increment will happen only if the whole transaction is complete. If the consumer crashes while processing, the transaction will be rolled back, and the offset will not be incremented. When the consumer restarts, it can reread the message as it failed to process it last time. This option leads to no data duplication and no data loss but can lead to decreased throughput.

[← Back](#)

Controller Broker

[Next →](#)

Kafka Characteristics

Kafka Characteristics

Let's explore different characteristics of Kafka.

We'll cover the following



- Storing messages to disks
- Record retention in Kafka
- Client quota
- Kafka performance

Storing messages to disks

Kafka writes its messages to the local disk and does not keep anything in RAM. Disks storage is important for durability so that the messages will not disappear if the system dies and restarts. Disks are generally considered to be slow. However, there is a huge difference in disk performance between random block access and sequential access. Random block access is slower because of numerous disk seeks, whereas the sequential nature of writing or reading, enables disk operations to be thousands of times faster than random access. Because all writes and reads happen sequentially, Kafka has a very high throughput.

Writing or reading sequentially from disks are heavily optimized by the OS, via **read-ahead** (prefetch large block multiples) and **write-behind** (group small logical writes into big physical writes) techniques.

Also, modern operating systems cache the disk in free RAM. This is called Pagecache (https://en.wikipedia.org/wiki/Page_cache). Since Kafka stores messages in a standardized binary format unmodified throughout the whole flow (producer → broker → consumer), it can make use of the zero-copy (<https://en.wikipedia.org/wiki/Zero-copy>) optimization. That is when the operating system copies data from the Pagecache directly to a socket, effectively bypassing the Kafka broker application entirely.

Kafka has a protocol that groups messages together. This allows network requests to group messages together and reduces network overhead. The server, in turn, persists chunks of messages in one go, and consumers fetch large linear chunks at once.

All of these optimizations allow Kafka to deliver messages at near network-speed.

Record retention in Kafka

By default, Kafka retains records until it runs out of disk space. We can set time-based limits (configurable retention period), size-based limits (configurable based on size), or compaction (keeps the latest version of record using the key). For example, we can set a retention policy of three days, or two weeks, or a month, etc. The records in the topic are available for consumption until discarded by time, size, or compaction.

Client quota

It is possible for Kafka producers and consumers to produce/consume very high volumes of data or generate requests at a very high rate and thus monopolize broker resources, cause network saturation, and, in general, deny service to other clients and the brokers themselves. Having quotas

protects against these issues. Quotas become even more important in large multi-tenant clusters where a small set of badly behaved clients can degrade the user experience for the well-behaved ones.

In Kafka, quotas are byte-rate thresholds defined per `client-ID`. A `client-ID` logically identifies an application making a request. A single `client-ID` can span multiple producer and consumer instances. The quota is applied for all instances as a single entity. For example, if a `client-ID` has a producer quota of 10 MB/s, that quota is shared across all instances with that same ID.

The broker does not return an error when a client exceeds its quota but instead attempts to slow the client down. When the broker calculates that a client has exceeded its quota, it slows the client down by holding the client's response for enough time to keep the client under the quota. This approach keeps the quota violation transparent to clients. This also prevents clients from having to implement special back-off and retry behavior.

Kafka performance

Here are a few reasons behind Kafka's performance and popularity:

Scalability: Two important features of Kafka contribute to its scalability.

- A Kafka cluster can easily expand or shrink (brokers can be added or removed) while in operation and without an outage.
- A Kafka topic can be expanded to contain more partitions. Because a partition cannot expand across multiple brokers, its capacity is bounded by broker disk space. Being able to increase the number of partitions and the number of brokers means there is no limit to how much data a single topic can store.

Fault-tolerance and reliability: Kafka is designed in such a way that a broker failure is detectable by ZooKeeper and other brokers in the cluster. Because each topic can be replicated on multiple brokers, the cluster can recover from broker failures and continue to work without any disruption of service.

Throughput: By using consumer groups, consumers can be parallelized, so that multiple consumers can read from multiple partitions on a topic, allowing a very high message processing throughput.

Low Latency: 99.99% of the time, data is read from disk cache and RAM; very rarely, it hits the disk.

[← Back](#)

Kafka Delivery Semantics

[Next →](#)

Summary: Kafka

Summary: Kafka

Here is a quick summary of Kafka for you!

We'll cover the following



- Summary
- System design patterns
 - High-water mark
 - Leader and follower
 - Split-brain
 - Segmented log
- References and further reading

Summary

1. Kafka provides low-latency, high-throughput, fault-tolerant **publish and subscribe pipelines** and can process huge continuous streams of events.
2. Kafka can function both as a **message queue** and a **publisher-subscriber** system.
3. At a high level, Kafka works as a **distributed commit log**.
4. Kafka server is also called a **broker**. A Kafka cluster can have one or more brokers.
5. A Kafka **topic** is a logical aggregation of messages.
6. Kafka solves the scaling problem of a messaging system by splitting a topic into multiple **partitions**.

7. Every topic partition is replicated for fault tolerance and redundancy.
8. A partition has one leader replica and zero or more follower replicas.
9. Partition leader is responsible for all reads and writes. Each follower's responsibility is to replicate the leader's data to serve as a 'backup' partition.
10. Message ordering is preserved only on a per-partition basis (not across partitions of a topic).
11. Every partition replica needs to fit on a broker, and a partition cannot be divided over multiple brokers.
12. Every broker can have one or more leaders, covering different partitions and topics.
13. Kafka supports a single queue model with multiple readers by enabling consumer groups.
14. Kafka supports a publish-subscribe model by allowing consumers to subscribe to topics for which they want to receive messages.
15. ZooKeeper functions as a centralized configuration management service.

System design patterns

Here is a summary of system design patterns used in Kafka.

High-water mark

To deal with non-repeatable reads and ensure data consistency, brokers keep track of the high-water mark, which is the largest offset that all ISRs of a particular partition share. Consumers can see messages only until the high watermark.

Leader and follower

Each Kafka partition has a designated leader responsible for all reads and writes for that partition. Each follower's responsibility is to replicate the leader's data to serve as a 'backup' partition.

Split-brain

To handle split-brain (where we have multiple active controller brokers), Kafka uses 'epoch number,' which is simply a monotonically increasing number to indicate a server's generation. This means if the old Controller had an epoch number of '1', the new one would have '2'. This epoch is included in every request that is sent from the Controller to other brokers. This way, brokers can easily differentiate the real Controller by simply trusting the Controller with the highest number. This epoch number is stored in ZooKeeper.

Segmented log

Kafka uses log segmentation to implement storage for its partitions. As Kafka regularly needs to find messages on disk for purging, a single long file could be a performance bottleneck and error-prone. For easier management and better performance, the partition is split into segments.

References and further reading

- Confluent Docs (<https://docs.confluent.io/current/kafka/design.html#ak-design>)
- New York Times use case (<https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/>)
- Kafka Summit (<https://www.confluent.io/resources/kafka-summit-san-francisco-2019/>)
- Kafka Acks Explained (<https://medium.com/better-programming/kafka-acks-explained-c0515b3b707e>)

- Kafka as distributed log (<https://www.youtube.com/watch?v=ElilYxUOjOQ>)
- Minimizing Kafka Latency (<https://www.confluent.io/blog/configure-kafka-to-minimize-latency/>)
- Kafka internal storage (<https://thehoard.blog/how-kafkas-storage-internals-work-3a29b02e026>)
- Exactly-Once semantics (<https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>)
- Split-brain
(https://techthoughts.typepad.com/managing_computers/2007/10/split-brain-quo.html)

← Back

Kafka Characteristics

Next →

Quiz: Kafka

Chubby: How to Design a Distributed Locking Service?

Chubby: Introduction

This lesson will introduce Chubby and its use cases.

We'll cover the following



- Goal
- What is Chubby?
- Chubby use cases
 - Leader/master election
 - Naming service (like DNS)
 - Storage (small objects that rarely change)
 - Distributed locking mechanism
 - When not to use Chubby
- Background
 - Chubby and Paxos

Goal

Design a highly available and consistent service that can store small objects and provide a locking mechanism on those objects.

What is Chubby?

Chubby is a service that provides a distributed locking mechanism and also stores small files. Internally, it is implemented as a **key/value store** that also provides a locking mechanism on each object stored in it. It is extensively used in various systems inside Google to provide storage and coordination services for systems like **GFS** and **BigTable**. Apache **ZooKeeper** is the open-source alternative to Chubby.

In sum, Chubby is a centralized service offering developer-friendly interfaces (to acquire/release locks and create/read/delete small files). Moreover, it does all this with just a few extra lines of code to any existing application without a lot of modification to application logic.

Chubby use cases

Primarily Chubby was developed to provide a reliable locking service. Over time, some interesting uses of Chubby have evolved. Following are the top use cases where Chubby is practically being used:

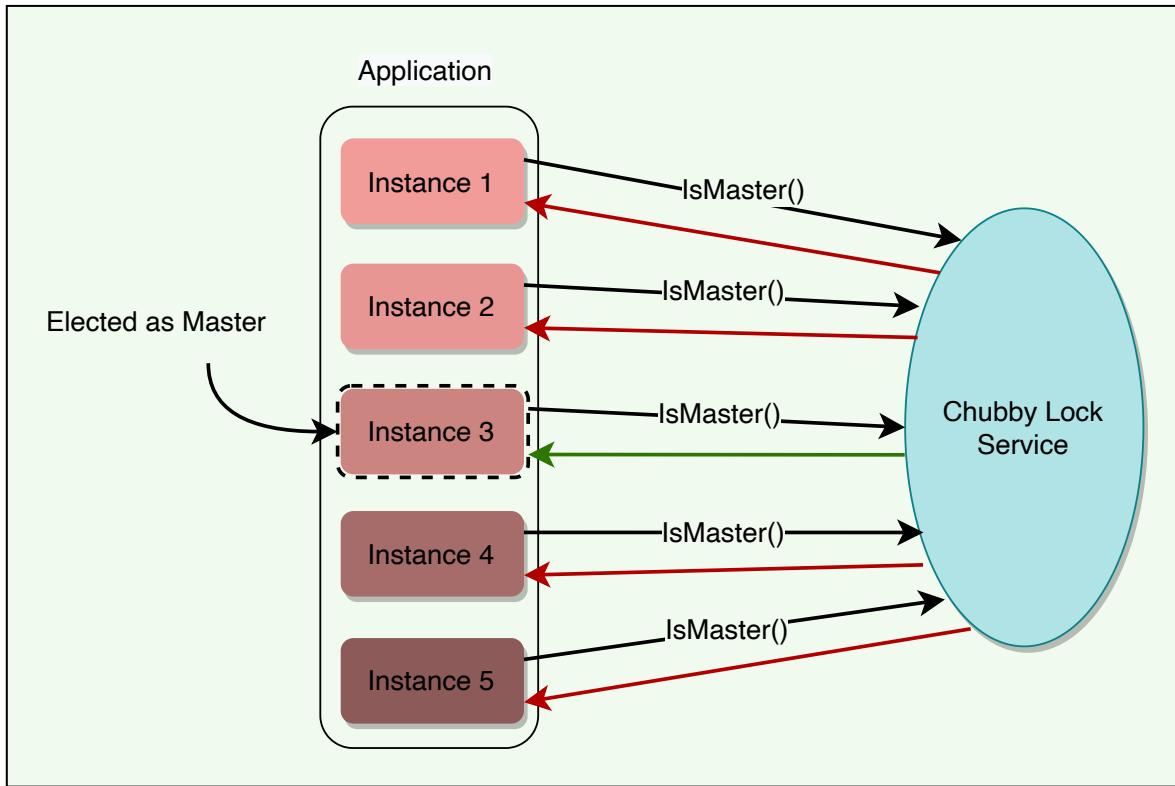
- Leader/master election
- Naming service (like DNS)
- Storage (small objects that rarely change)
- Distributed locking mechanism

Let's look into these use cases in detail.

Leader/master election

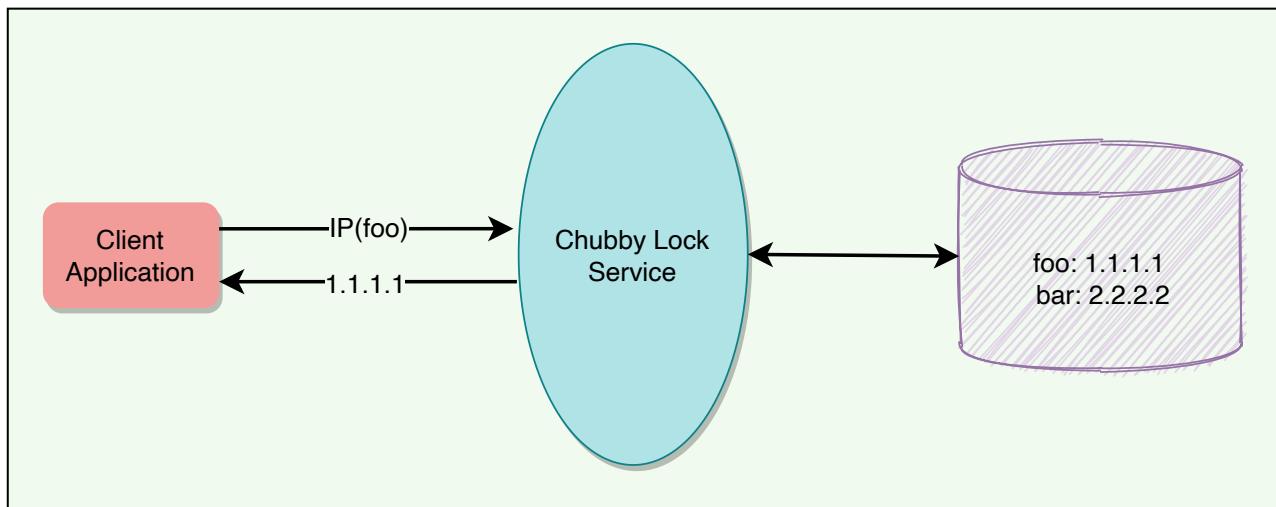
Any lock service can be seen as a consensus service, as it converts the problem of reaching consensus to handing out locks. Basically, a set of distributed applications compete to acquire a lock, and whoever gets the lock first gets the resource. Similarly, an application can have multiple replicas

running and wants one of them to be chosen as the leader. Chubby can be used for leader election among a set of replicas, e.g., the leader/master for GFS and BigTable.



Naming service (like DNS)

It is hard to make faster updates to DNS due to its time-based caching nature, which means there is generally a potential delay before the latest DNS mapping is effective. As a result, chubby has replaced DNS inside Google as the main way to discover servers.

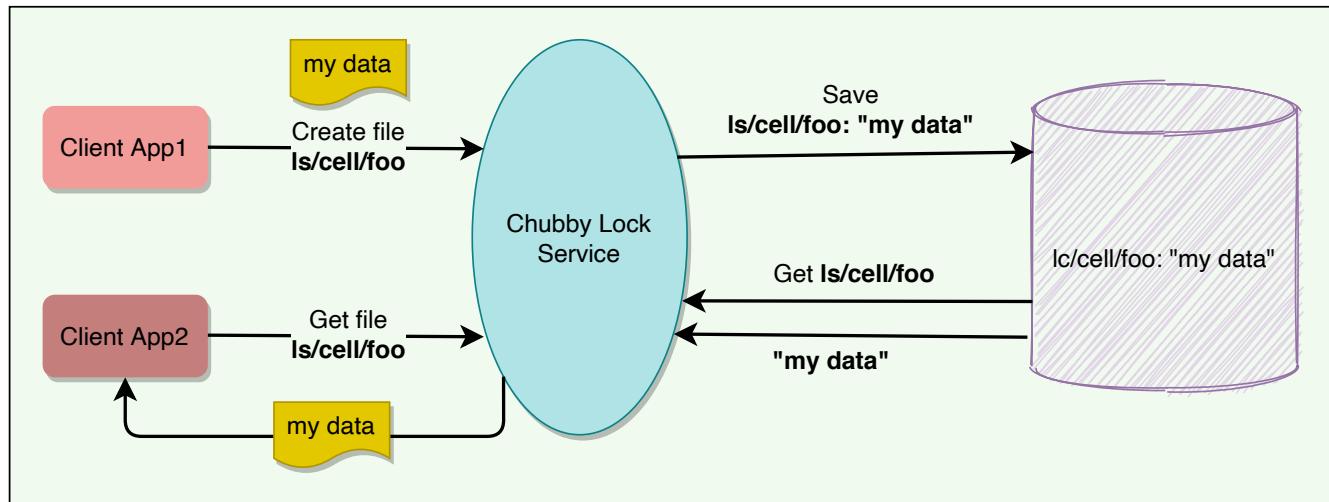


Chubby as DNS

Storage (small objects that rarely change)

Chubby provides a Unix-style interface to reliably store small files that do not change frequently (complementing the service offered by GFS).

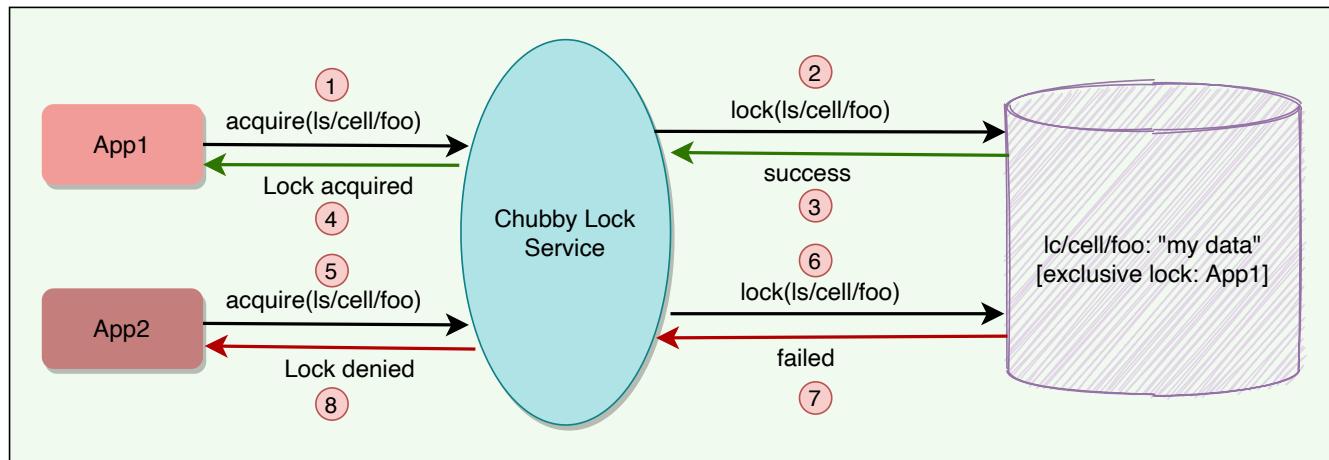
Applications can then use these files for any usage like DNS, configs, etc. GFS and Bigtable store their metadata in Chubby. Some services use Chubby to store ACL files.



Chubby as a storage service for small objects

Distributed locking mechanism

Chubby provides a developer-friendly interface for coarse-grained distributed locks (as opposed to fine-grained locks) to synchronize distributed activities in a distributed environment. All an application needs is a few code lines, and Chubby service takes care of all the lock management so that developers can focus on application business logic. In other words, we can say that Chubby provides mechanisms like semaphores and mutexes for a distributed environment.



Chubby as a distributed locking service

All these use cases are discussed in detail later.

At a high level, Chubby provides a framework for distributed consensus. All the above-mentioned use cases have emerged from this core service.

When not to use Chubby

Because of its design choices and proposed usage, Chubby should not be used when:

- Bulk storage is needed.
- Data update rate is high.
- Locks are acquired/released frequently.
- Usage is more like a publish/subscribe model.

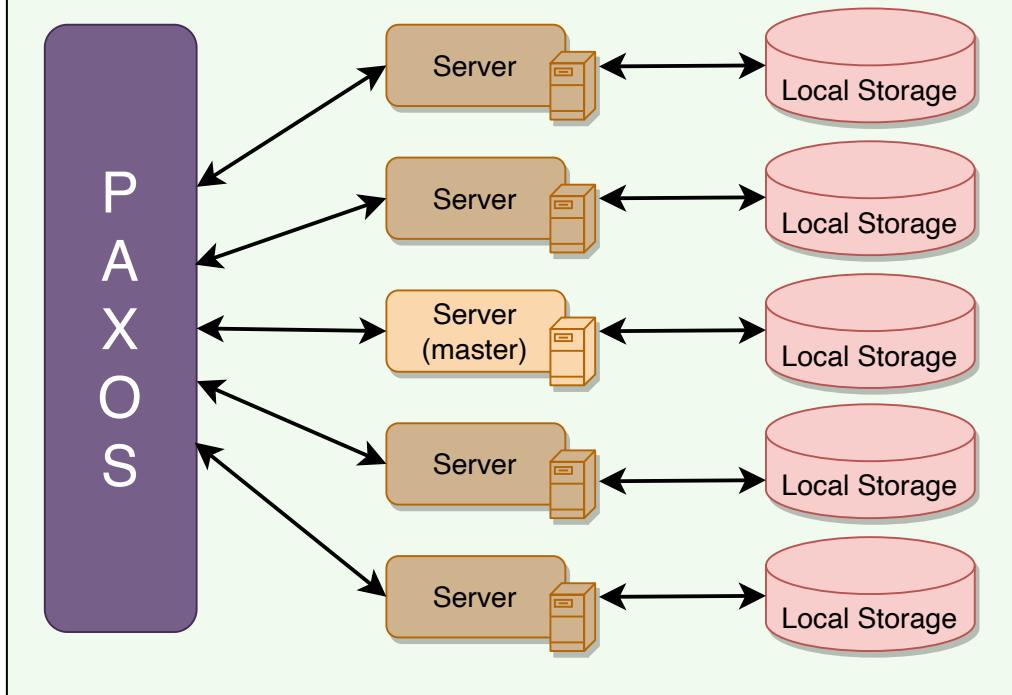
Background

Chubby is neither really a research effort, nor does it claim to introduce any new algorithms. Instead, chubby describes a certain design and implementation done at Google in order to provide a way for its clients to synchronize their activities and agree on basic information about their environment. More precisely, at Google, it has become primary to implement the above-mentioned use cases.

Chubby and Paxos

Paxos ([https://en.wikipedia.org/wiki/Paxos_\(computer_science\)](https://en.wikipedia.org/wiki/Paxos_(computer_science))) plays a major role inside Chubby. Readers familiar with Distributed Computing recognize that getting all nodes in a distributed system to agree on anything (e.g., election of primary among peers) is basically a kind of distributed consensus problem. Distributed consensus using Asynchronous Communication is already solved by Paxos protocol, and Chubby actually uses Paxos underneath to manage the state of the Chubby system at any point in time.

Chubby



Chubby uses Paxos to manage its system

← Back

Next →

Mock Interview: Kafka

High-level Architecture

High-level Architecture

This lesson gives a brief overview of Chubby's architecture.

We'll cover the following



- Chubby common terms
 - Chubby cell
 - Chubby servers
 - Chubby client library
- Chubby APIs
 - General
 - File
 - Locking
 - Sequencer

Chubby common terms

Before digging deep into Chubby's architecture, let's first go through some of its common terms:

Chubby cell

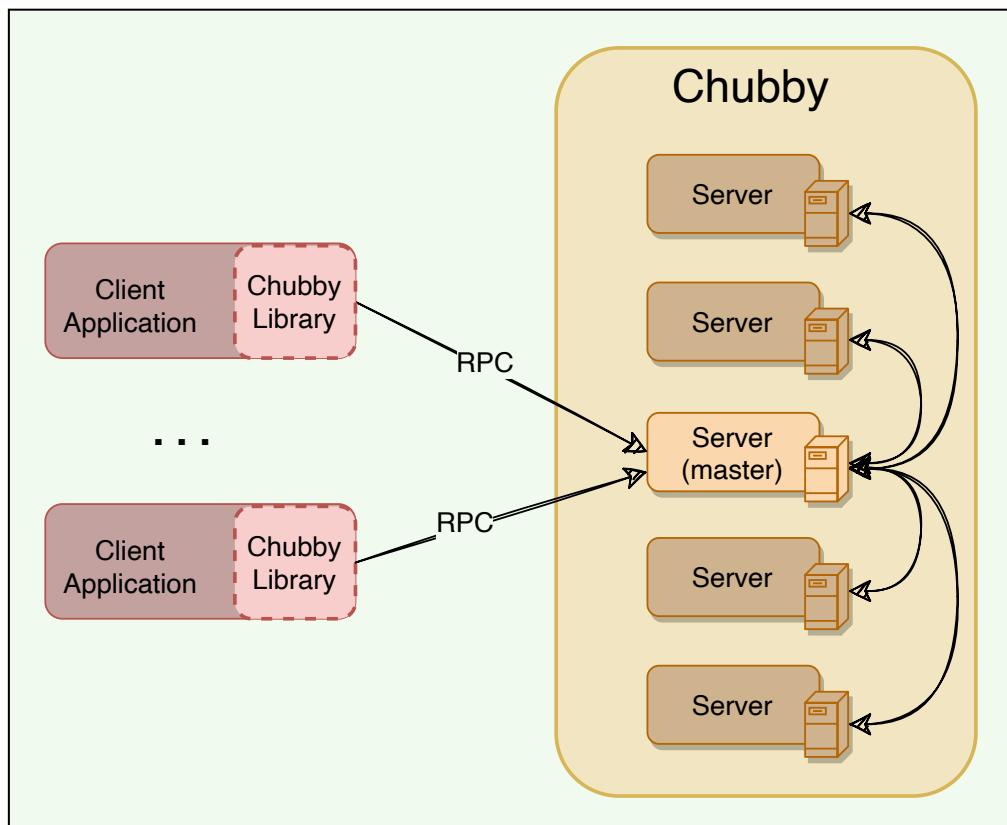
A Chubby Cell basically refers to a Chubby cluster. Most Chubby cells are confined to a single data center or machine room, though there can be a Chubby cell whose replicas are separated by thousands of kilometers. A single Chubby cell has two main components, server and client, that communicate via remote procedure call (RPC).

Chubby servers

- A chubby cell consists of a small set of servers (typically 5) known as replicas.
- Using Paxos, one of the servers is chosen as the master who handles all client requests. If the master fails, another server from replicas becomes the master.
- Each replica maintains a small database to store files/directories/locks. The master writes directly to its own local database, which gets synced asynchronously to all the replicas. That's how Chubby ensures data reliability and a smooth experience for clients even if the master fails.
- For fault tolerance, Chubby replicas are placed on different racks.

Chubby client library

Client applications use a Chubby library to communicate with the replicas in the chubby cell using RPC.



Chubby APIs

Chubby exports a file system interface similar to POSIX but simpler. It consists of a strict tree of files and directories in the usual way, with name components separated by slashes.

File format: /ls/chubby_cell/directory_name/.../file_name

Where `/ls` refers to the lock service, designating that this is part of the Chubby system, and `chubby_cell` is the name of a particular instance of a Chubby system (the term cell is used in Chubby to denote an instance of the system). This is followed by a series of directory names culminating in a `file_name`.

A special name, `/ls/local`, will be resolved to the most local cell relative to the calling application or service.

Chubby was originally designed as a lock service, such that every entity in it will be a lock. But later, its creators realized that it is useful to associate a small amount of data with each entity. Hence, each entity in Chubby can be used for locking or storing a small amount of data or both, i.e., storing small files with locks.

We can divide Chubby APIs into following groups:

- General
- File
- Locking
- Sequencer

General

1. `Open()` : Opens a given named file or directory and returns a handle.
2. `Close()` : Closes an open handle.
3. `Poison()` : Allows a client to cancel all Chubby calls made by other threads without fear of deallocating the memory being accessed by them.
4. `Delete()` : Deletes the file or directory.

File

1. `GetContentsAndStat()` : Returns (atomically) the whole file contents and metadata associated with the file. This approach of reading the whole file is designed to discourage the creation of large files, as it is not the intended use of Chubby.
2. `GetStat()` : Returns just the metadata.
3. `ReadDir()` : Returns the contents of a directory – that is, names and metadata of all children.
4. `SetContents()` : Writes the whole contents of a file (atomically).
5. `SetACL()` : Writes new access control list information

Locking

1. `Acquire()` : Acquires a lock on a file.
2. `TryAcquire()` : Tries to acquire a lock on a file; it is a non-blocking variant of `Acquire`.
3. `Release()` : Releases a lock.

Sequencer

1. `GetSequencer()` : Get the sequencer of a lock. A sequencer is a string representation of a lock.
2. `SetSequencer()` : Associate a sequencer with a handle.
3. `CheckSequencer()` : Check whether a sequencer is valid.

Chubby does not support operations like append, seek, move files between directories, or making symbolic or hard links. Files can only be completely read or completely written/overwritten. This makes it practical only for storing very small files.

[← Back](#)

Chubby: Introduction

[Next →](#)

Design Rationale

Design Rationale

Let's explore the rationale behind Chubby's architecture.

We'll cover the following



- Why was Chubby built as a service?
- Why coarse-grained locks?
- Why advisory locks?
- Why Chubby needs storage?
- Why does Chubby export a Unix-like file system interface?
- High availability and reliability

Before we jump into further details and working of Chubby, it is important to know the logic behind certain design decisions. These learnings can be applied to other problems of similar nature.

Why was Chubby built as a service?

Let's first understand the reason behind building a service instead of having a client library that only provides Paxos distributed consensus. A lock service has some clear advantages over a client library:

- **Development becomes easy:** Sometimes high availability is not planned in the early phases of development. Systems start as a prototype with little load and lose availability guarantees. As a service matures and gains more clients, availability becomes important; replication and primary election are then added to design. While this

could be done with a library that provides distributed consensus, a lock server makes it easier to maintain the existing program structure and communication patterns. For example, electing a leader requires adding just a few lines of code. This technique is easier than making existing servers participate in a consensus protocol, especially if compatibility must be maintained during a transition period.

- **Lock-based interface is developer-friendly:** Programmers are generally familiar with locks. It is much easier to simply use a lock service in a distributed system than getting involved in managing Paxos protocol state locally, e.g., `Acquire()`, `TryAcquire()`, `Release()`.
- **Provide quorum & replica management:** Distributed consensus algorithms need a quorum to make a decision, so several replicas are used for high availability. One can view the lock service as a way of providing a generic electorate that allows a client application to make decisions correctly when less than a majority of its own members are up. Without such support from a service, each application needs to have and manage its own quorum of servers.
- **Broadcast functionality:** Clients and replicas of a replicated service may wish to know when the service's master changes; this requires an event notification mechanism. Such a mechanism is easy to build if there is a central service in the system.

The arguments above clearly show that building and maintaining a central locking service abstracts away and takes care of a lot of complex problems from client applications.

Why coarse-grained locks?

Chubby locks usage is not expected to be fine-grained in which they might be held for only a short period (i.e., seconds or less). For example, electing a leader is not a frequent event. Following are some main reasons why Chubby decided to only support coarse-grained locks:

- **Less load on lock server:** Coarse-grained locks impose far less load on the server as the lock acquisition rate is a lot less than the client's transaction rate.
- **Survive server failures:** As coarse-grained locks are acquired rarely, clients are not significantly delayed by the temporary unavailability of the lock server. With fine-grained locks, even a brief unavailability of a lock server would cause many clients to stall.
- **Fewer lock servers are needed:** Coarse-grained locks allow many clients to be adequately served by a modest number of lock servers with somewhat lower availability.

Why advisory locks?

Chubby locks are advisory, which means it is up to the application to honor the lock. Chubby doesn't make locked objects inaccessible to clients not holding their locks. It is more like record keeping and allows the lock requester to discover that lock is held. Holding a specific lock is neither necessary to access the file, nor prevents others from doing so.

Other types of locks are mandatory locks, which make objects inaccessible to clients not holding the lock. Chubby gave following reasons for not having mandatory locks:

- To enforce mandatory locking on resources implemented by other services would require more extensive modification of these services.

- Mandatory locks prevent users from accessing a locked file for debugging or administrative purposes. If a file must be accessed, an entire application would need to be shut down or rebooted to break the mandatory lock.
- Generally, a good developer practice is to write assertions such as `assert("Lock X is held")`, so mandatory locks bring only little benefit anyway.

Why Chubby needs storage?

Chubby's storage is important as client applications may need to advertise Chubby's results with others. For example, an application needs to store some info to:

- Advertise its selected primary (leader election use case)
- Resolve aliases to absolute addresses (naming service use case)
- Announce the scheme after repartitioning of data.

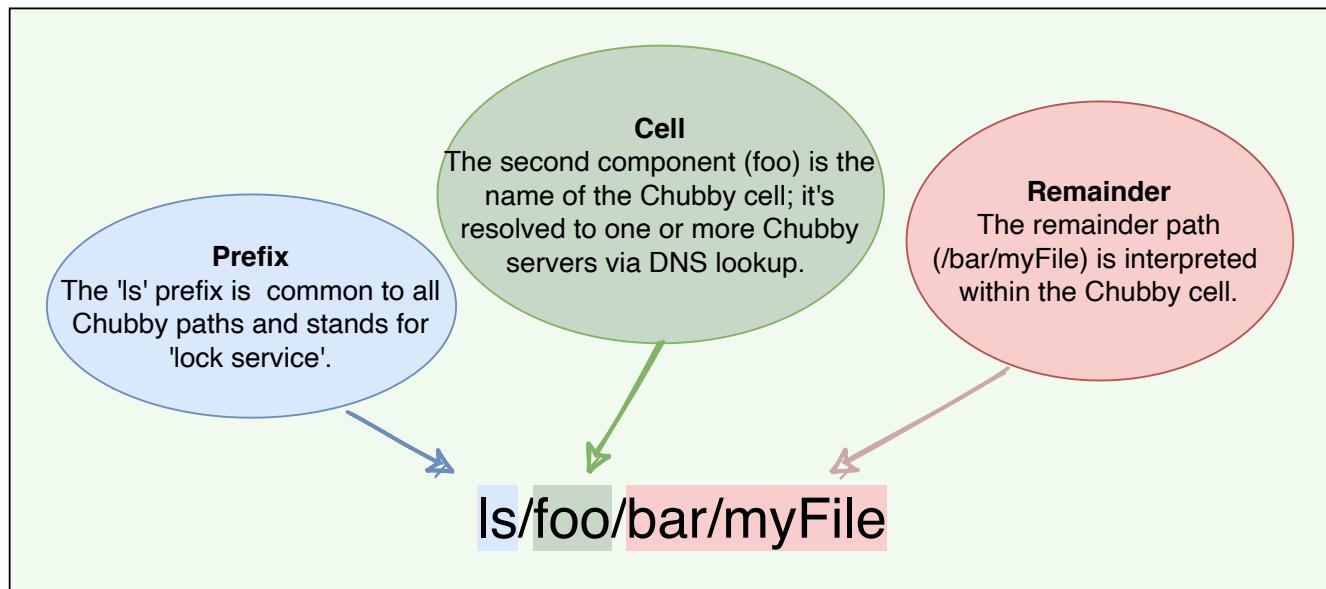
Not having a separate service for sharing the results reduces the number of servers that clients depend on. Chubby's storage requirements are really simple. i.e., store a small amount of data (KBs) and limited operation support (i.e., create/delete).

Why does Chubby export a Unix-like file system interface?

Recall that Chubby exports a file system interface similar to Unix but simpler. It consists of a strict tree of files and directories in the usual way, with name components separated by slashes.

File format: /ls/cell/remainder-path

The main reason why Chubby's naming structure resembles a file system to make it available to applications both with its own specialized API, and via interfaces used by our other file systems, such as the Google File System. This significantly reduced the effort needed to write basic browsing and namespace manipulation tools, and reduced the need to educate casual Chubby users. However, only a very limited number of operations can be performed on these files, e.g., Create, Delete, etc.



Breaking down Chubby paths

High availability and reliability

As proved by CAP theorem

(<https://www.educative.io/collection/page/5668639101419520/5559029852536832/5998984290631680>), no application can be highly available, strongly consistent, and high-performing at the same time. Due to the nature of Chubby's expected use cases, Chubby compromises on performance in favor of availability and consistency.

← Back

Next →

High-level Architecture

How Chubby Works

How Chubby Works

Let's learn how Chubby works.

We'll cover the following



- Service initialization
- Client initialization
- Leader election example using Chubby
 - Sample pseudocode for leader election

Service initialization

Upon initialization, Chubby performs the following steps:

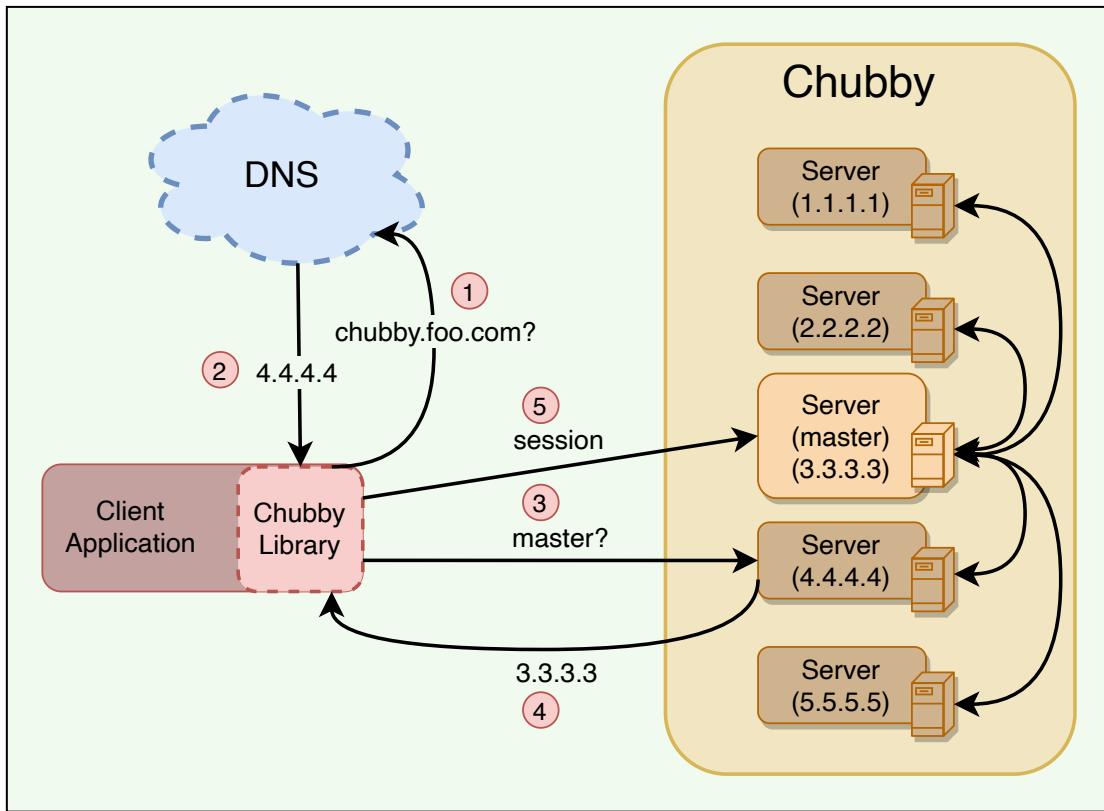
- A master is chosen among Chubby replicas using Paxos.
- Current master information is persisted in storage, and all replicas become aware of the master.

Client initialization

Upon initialization, a Chubby client performs the following steps:

- Client contacts the DNS to know the listed Chubby replicas.
- Client calls any Chubby server directly via Remote Procedure Call (RPC).
- If that replica is not the master, it will return the address of the current master.

- Once the master is located, the client maintains a session with it and sends all requests to it until it indicates that it is not the master anymore or stops responding.



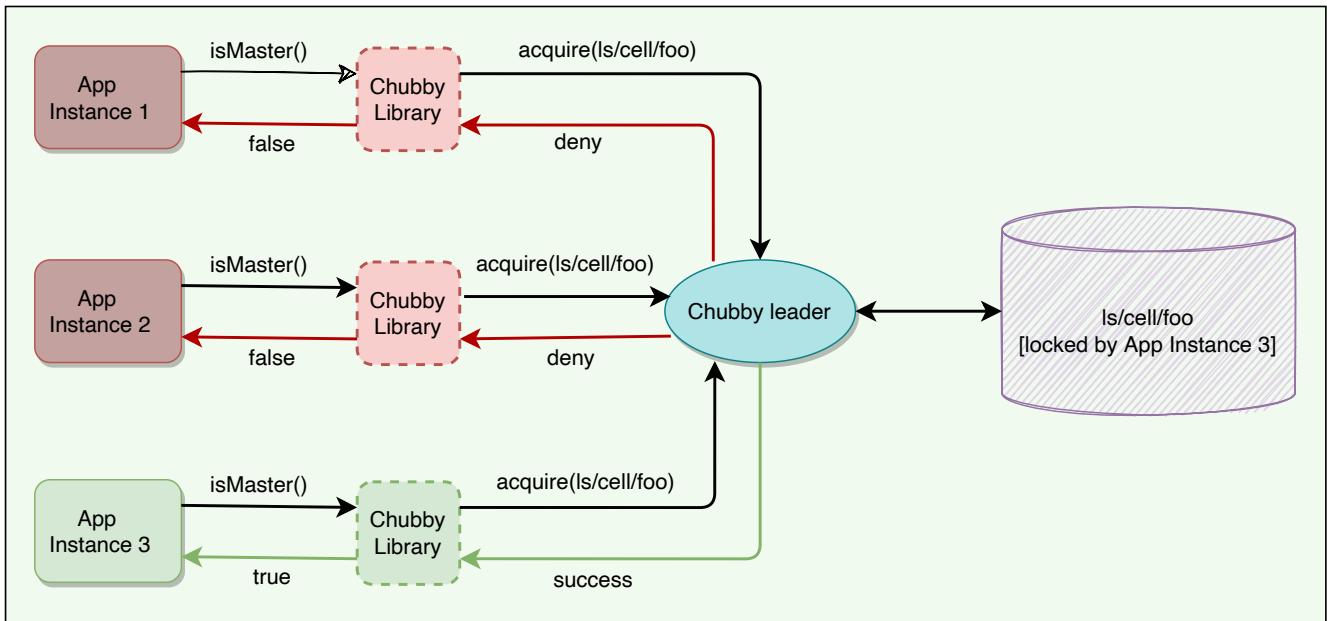
Chubby high-level architecture

Leader election example using Chubby

#

Let's take an example of an application that uses Chubby to elect a single master from a bunch of instances of the same application.

Once the master election starts, all candidates attempt to acquire a Chubby lock on a file associated with the election. Whoever acquires the lock first becomes the master. The master writes its identity on the file, so that other processes know who the current master is.



Sample pseudocode for leader election

The pseudocode below shows how easy it is to add leader election logic to existing applications with just a few additional code lines.

```

/* Create these files in Chubby manually once.
Usually there are at least 3-5 required for minimum quorum requirement. */
lock_file_paths = {
    "ls/cell1/foo",
    "ls/cell2/foo",
    "ls/cell3/foo",
}

Main() {
    // Initialize Chubby client library.
    chubbyLeader = newChubbyLeader(lock_file_paths)

    // Establish client's connection with Chubby service.
    chubbyLeader.Start()

    // Waiting to become the leader.
    chubbyLeader.Wait()

    // Becomes Leader
    Log("Is Leader: " + chubbyLeader.isLeader())

    While(chubbyLeader.renewLeader ()) {
        // Do work
    }
    // Not leader anymore.
}

```

[← Back](#)

Design Rationale

[Next →](#)

File, Directories, and Handles

File, Directories, and Handles

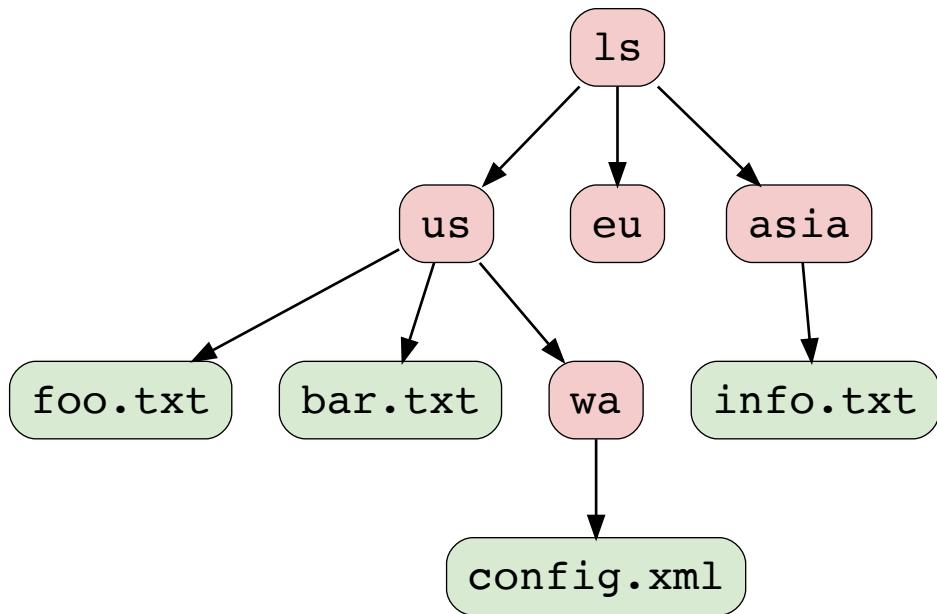
Let's explore how Chubby works with files, directories, and handles.

We'll cover the following



- Nodes
- Metadata
- Handles

Chubby file system interface is basically a tree of files and directories, where each directory contains a list of child files and directories. Each file or directory is called a node.



Chubby file system

Nodes

- Any node can act as an advisory reader/writer lock.
- Nodes may either be ephemeral or permanent.
- Ephemeral files are used as temporary files, and act as an indicator to others that a client is alive.
- Ephemeral files are also deleted if no client has them open.
- Ephemeral directories are also deleted if they are empty.
- Any node can be explicitly deleted.

Metadata

Metadata for each node includes Access Control Lists (ACLs), four monotonically increasing 64-bit numbers, and a checksum.

ACLs are used to control reading, writing, and changing the ACL names for the node.

- Node inherits the ACL names of its parent directory on creation.
- ACLs themselves are files located in an ACL directory, which is a well-known part of the cell's local namespace.
- Users are authenticated by a mechanism built into the RPC system.

Monotonically increasing 64-bit numbers: These numbers allow clients to detect changes easily.

- **An instance number:** This is greater than the instance number of any previous node with the same name.
- **A content generation number** (files only): This is incremented every time a file's contents are written.
- **A lock generation number:** This is incremented when the node's lock transitions from free to held.

- **An ACL generation number:** This is incremented when the node's ACL names are written.

Checksum: Chubby exposes a 64-bit file-content checksum so clients may tell whether files differ.

Handles

Clients open nodes to obtain handles (that are analogous to UNIX file descriptors). Handles include:

- **Check digits:** Prevent clients from creating or guessing handles, so full access control checks are performed only when handles are created.
- **A sequence number:** Enables a master to tell whether a handle was generated by it or by a previous master.
- **Mode information** (provided at open time): Enables the master to recreate its state if an old handle is presented to a newly restarted leader.

[← Back](#)

[Next →](#)

How Chubby Works

Locks, Sequencers, and Lock-delays

Locks, Sequencers, and Lock-delays

Let's explore how Chubby implements Locks and Sequencers.

We'll cover the following



- Locks
- Sequencer
- Lock-delay

Locks

Each chubby node can act as a reader-writer lock in one of the following two ways:

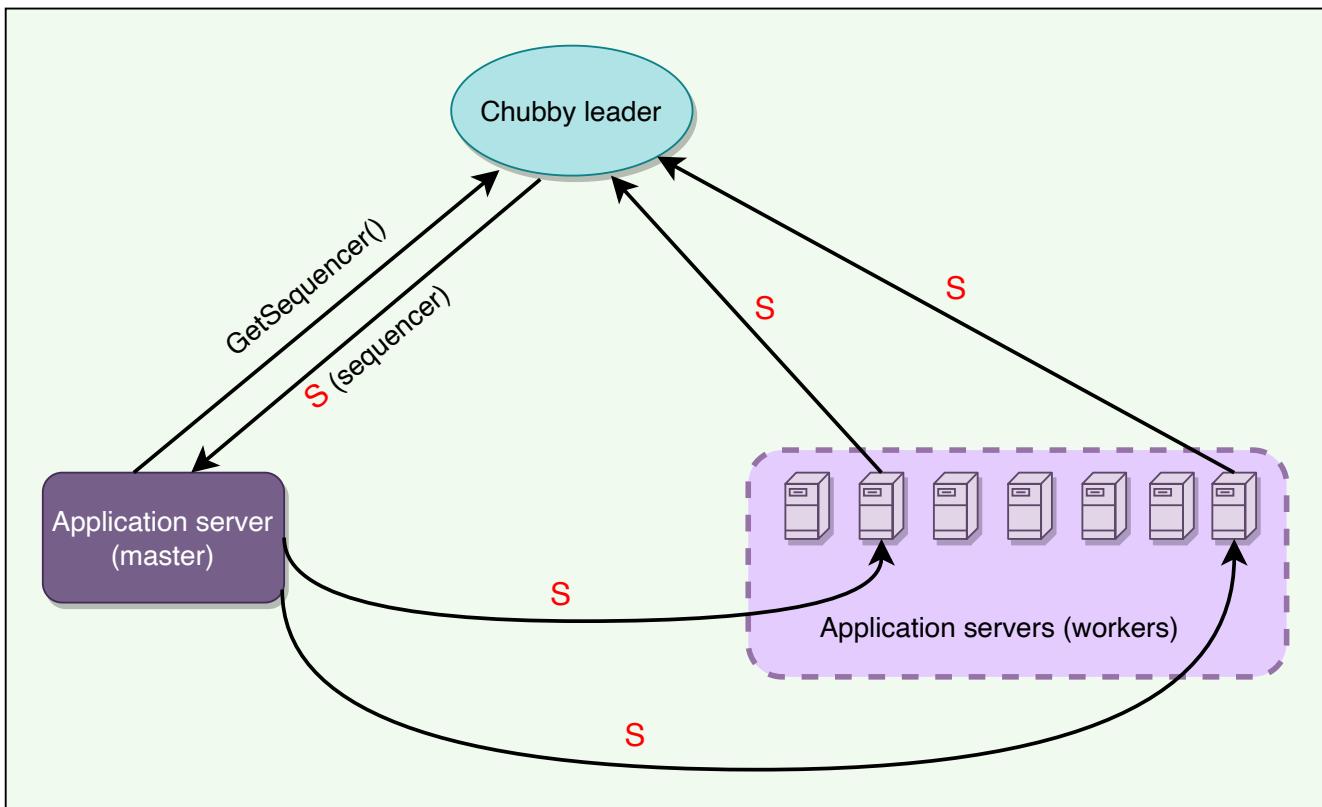
- **Exclusive:** One client may hold the lock in exclusive (write) mode.
- **Shared:** Any number of clients may hold the lock in shared (reader) mode.

Sequencer

With distributed systems, receiving messages out of order is a problem; Chubby uses sequence numbers to solve this problem. After acquiring a lock on a file, a client can immediately request a 'Sequencer,' which is an opaque byte string describing the state of the lock:

Sequencer = Name of the lock + Lock mode (*exclusive or shared*) + Lock generation number

An application's master server can generate a sequencer and send it with any internal order to other servers. Application servers that receive orders from a primary can check with Chubby if the sequencer is still good and does not belong to a stale primary (to handle the 'Brain split' scenario).



Application master generating a sequencer and passing it to worker servers

Lock-delay

For file servers that do not support sequencers, Chubby provides a lock-delay period to protect against message delays and server restarts.

If a client releases a lock in the normal way, it is immediately available for other clients to claim, as one would expect. However, if a lock becomes free because the holder has failed or become inaccessible, the lock server will prevent other clients from claiming the lock for a period called the lock-delay.

- Clients may specify any lock-delay up to some bound, defaults to one minute. This limit prevents a faulty client from making a lock (and thus some resource) unavailable for an arbitrarily long time.
- While imperfect, the lock-delay protects unmodified servers and clients from everyday problems caused by message delays and restarts.

[← Back](#)

File, Directories, and Handles

[Next →](#)

Sessions and Events

Sessions and Events

This lesson will explain Chubby sessions and what different Chubby events are.

We'll cover the following



- What is a Chubby session?
- Session protocol
- What is KeepAlive?
- Session optimization
- Failovers

What is a Chubby session?

A Chubby session is a relationship between a Chubby cell and a Chubby client.

- It exists for some interval of time and is maintained by periodic handshakes called **KeepAlives**.
- Client's handles, locks, and cached data only remain valid provided its session remains valid.

Session protocol

- Client requests a new session on first contacting the master of Chubby cell.

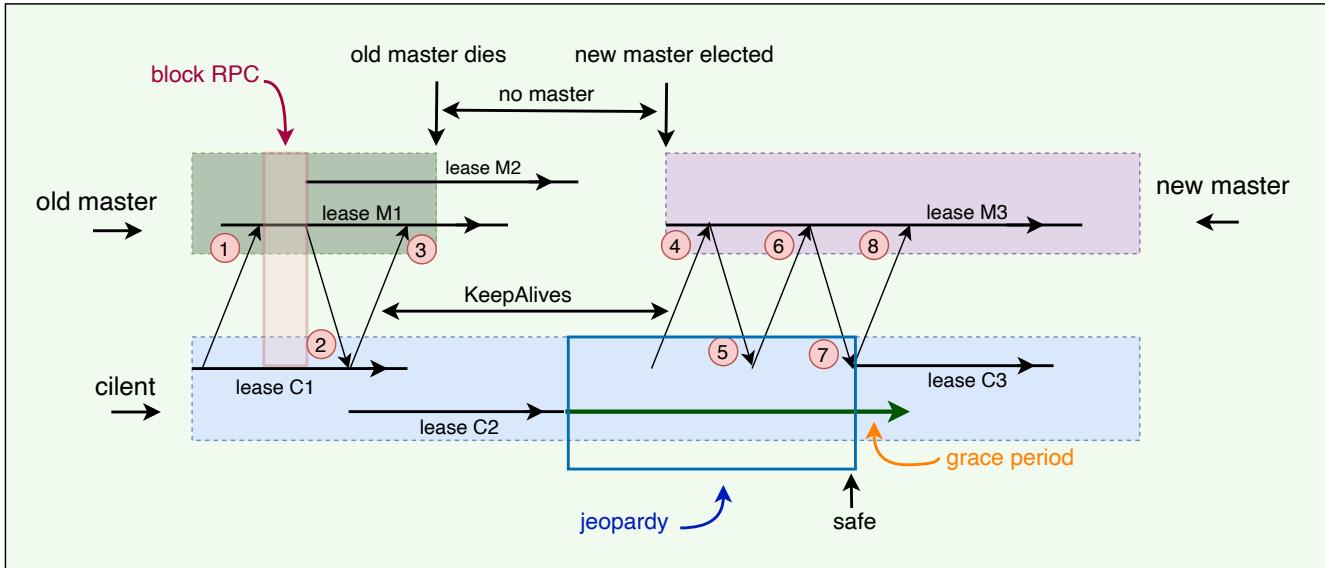
- A session ends if the client ends it explicitly or it has been idle. A session is considered idle if there are no open handles and calls for a minute.
- Each session has an associate lease, which is a time interval during which the master guarantees to not terminate the session unilaterally. The end of this interval is called ‘session lease timeout.’
- The master advances the ‘session lease timeout’ in the following three circumstances:
 - On session creation
 - When a master failover occurs
 - When the master responds to a KeepAlive RPC from the client

What is KeepAlive?

KeepAlive is basically a way for a client to maintain a constant session with Chubby cell. Following are basic steps of responding to a KeepAlive:

- On receiving a KeepAlive (step “1” in the diagram below), the master typically blocks the RPC (does not allow it to return) until the client’s previous lease interval is close to expiring.
- The master later allows the RPC to return to the client (step “2”) and thus informs the client of the new lease timeout (lease M2).
- The master may extend the timeout by any amount. The default extension is 12s, but an overloaded master may use higher values to reduce the number of KeepAlive calls it must process. Note the difference between the lease timeout of the client and the master (M1 vs. C1 and M2 vs. C2).
- The client initiates a new KeepAlive immediately after receiving the previous reply. Thus, the client ensures that there is almost always a KeepAlive call blocked at the master.

In the diagram below, thick arrows represent lease sessions, upward arrows are KeepAlive requests, and downward arrows are KeepAlive responses. We will discuss this diagram in detail in the next two sections.



Client maintaining a session with Chubby cell through KeepAlive

Session optimization

Piggybacking events: KeepAlive reply is used to transmit events and cache invalidations back to the client.

Local lease: The client maintains a local lease timeout that is a conservative approximation of the master's lease timeout.

Jeopardy: If a client's local lease timeout expires, it becomes unsure whether the master has terminated its session. The client empties and disables its cache, and we say that its session is in jeopardy.

Grace period: When a session is in jeopardy, the client waits for an extra time called the grace period - 45s by default. If the client and master manage to exchange a successful KeepAlive before the end of client's grace period,

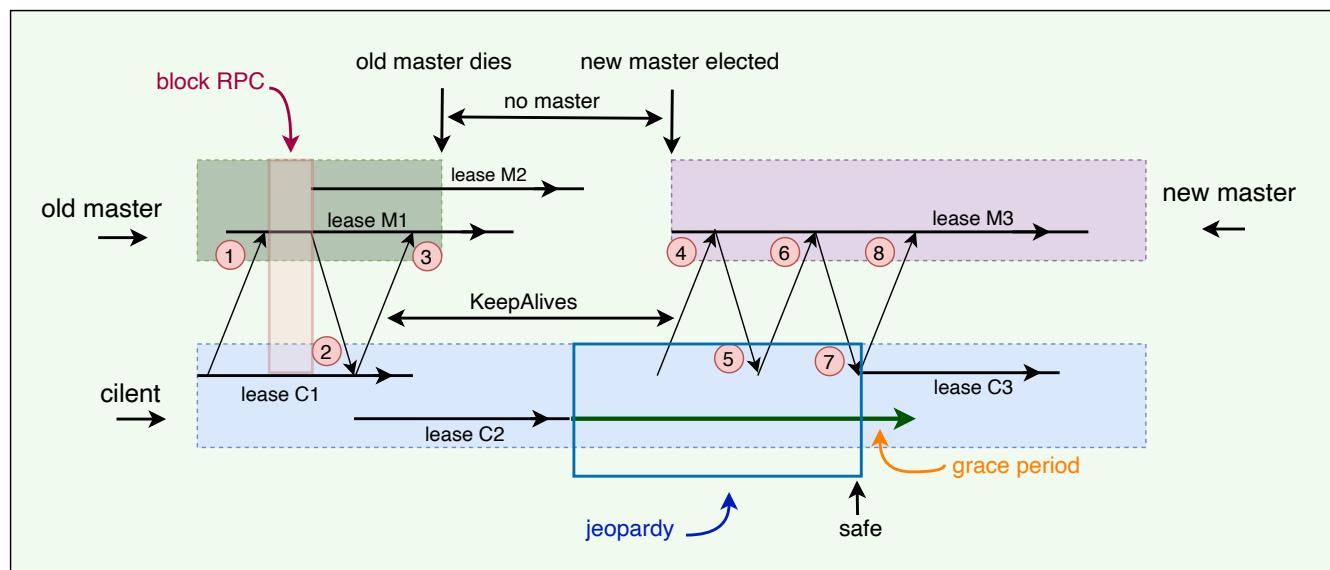
the client enables its cache once more. Otherwise, the client assumes that the session has expired.

Failovers

The failover scenario happens when a master fails or otherwise loses membership. Following is the summary of things that happen in case of a master failover:

- The failing master discards its in-memory state about sessions, handles, and locks.
- Session lease timer is stopped. This means no lease is expired during the time when the master failover is happening. This is equivalent to lease extension.
- If master election occurs quickly, the clients contact and continue with the new master before the client's local lease expires.
- If the election is delayed, the clients flush their caches (= jeopardy) and wait for the "grace period" (45s) while trying to find the new master.

Let's look at an example of failover in detail.



1. Client has lease M1 (& local lease C1) with master and pending KeepAlive request.
2. Master starts lease M2 and replies to the KeepAlive request.
3. Client extends the local lease to C2 and makes a new KeepAlive call.
Master dies before replying to the next KeepAlive. So, no new leases can be assigned. Client's C2 lease expires, and the client library flushes its cache and informs the application that it has entered jeopardy. The grace period starts on the client.
4. Eventually, a new master is elected and initially uses a conservative approximation M3 of the session lease that its predecessor may have had for the client. Client sends KeepAlive to new master (4).
5. The first KeepAlive request from the client to the new master is rejected (5) because it has the wrong master epoch number (described in the next section).
6. Client retries with another KeepAlive request.
7. Retried KeepAlive succeeds. Client extends its lease to C3 and optionally informs the application that its session is no longer in jeopardy (session is in the safe mode now).
8. Client makes a new KeepAlive call, and the normal protocol works from this point onwards.
9. Because the grace period was long enough to cover the interval between the end of lease C2 and the beginning of lease C3, the client saw nothing but a delay. If the grace period was less than that interval, the client would have abandoned the session and reported the failure to the application.

← Back

Next →

Master Election and Chubby Events

This lesson will explain what actions a newly elected master performs. Additionally, we will look into different Chubby events.

We'll cover the following



- Initializing a newly elected master
- Chubby events

Initializing a newly elected master

A newly elected master proceeds as follows:

1. **Picks epoch number:** It first picks up a new client epoch number to differentiate itself from the previous master. Clients are required to present the epoch number on every call. The master rejects calls from clients using older epoch numbers. This ensures that the new master will not respond to a very old packet that was sent to the previous master.
2. **Responds to master-location requests** but does not respond to session-related operations yet.
3. **Build in-memory data structures:**
 - It builds in-memory data structures for sessions and locks that are recorded in the database.
 - Session leases are extended to the maximum that the previous master may have been using.

- 4. Let clients perform KeepAlives** but no other session-related operations at this point.
- 5. Emits a failover event to each session:** This causes clients to flush their caches (because they may have missed invalidations) and warn applications that other events may have been lost.
- 6. Wait:** The master waits until each session acknowledges the failover event or lets its session expire.
- 7. Allow all operations to proceed.**
- 8. Honor older handles by clients:** If a client uses a handle created prior to the failover, the master recreates the in-memory representation of the handle and honors the call.
- 9. Deletes ephemeral files:** After some interval (a minute), the master deletes ephemeral files that have no open file handles. Clients should refresh handles on ephemeral files during this interval after a failover.

Chubby events

Chubby supports a simple event mechanism to let its clients subscribe to a variety of events. Events are delivered to the client asynchronously via callbacks from the Chubby library. Clients subscribe to a range of events while creating a handle. Here are examples of such events:

- File contents modified
- Child node added, removed, or modified
- Chubby master failed over
- A handle (and its lock) has become invalid.
- Lock acquired
- Conflicting lock request from another client

Additionally, the client sends the following session events to the application:

- **Jeopardy:** When session lease timeout and grace period begins.
- **Safe:** When a session is known to have survived a communication problem
- **Expired:** If the session times out

← Back

Sessions and Events

Next →

Caching

Caching

Let's learn how Chubby implements its cache.

We'll cover the following



- Chubby cache
- Cache invalidation

Chubby cache

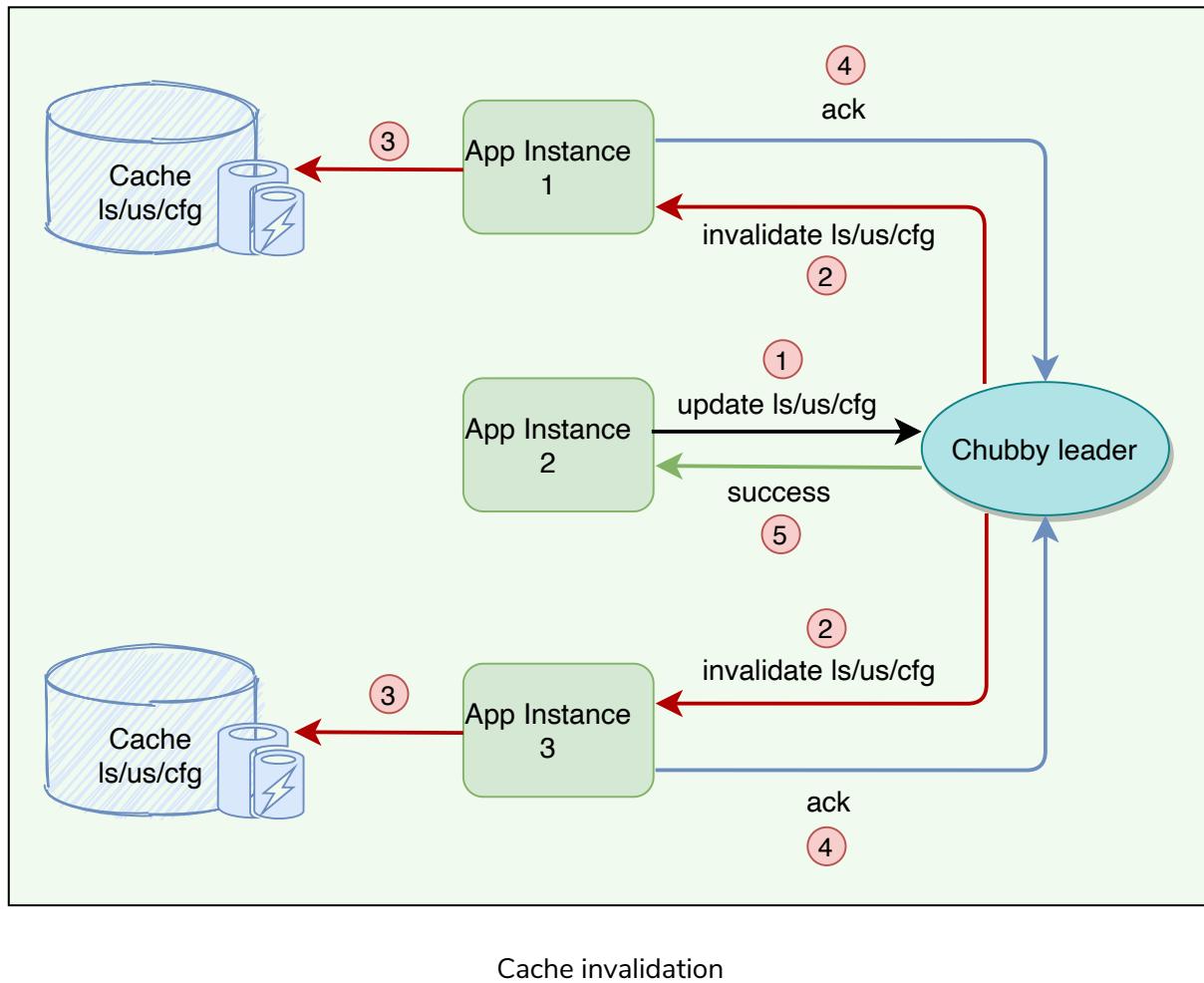
In Chubby, caching plays an important role, as read requests greatly outnumber write requests. To reduce read traffic, Chubby clients cache file contents, node metadata, and information on open handles in a consistent, write-through cache in the client's memory. Because of this caching, Chubby must maintain consistency between a file and a cache as well as between the different replicas of the file. Chubby clients maintain their cache by a lease mechanism and flush the cache when the lease expires.

Cache invalidation

Below is the protocol for invalidating the cache when file data or metadata is changed:

- Master receives a request to change file contents or node metadata.
- Master blocks modification and sends cache invalidations to all clients who have cached it. For this, the master must maintain a list of each client's cache contents.

- For efficiency, the invalidation requests are piggybacked onto KeepAlive replies from the master.
- Clients receive the invalidation signal, flushes the cache, and sends an acknowledgment to the master with its next KeepAlive call.
- Once acknowledgments are received from each active client, the master proceeds with the modification. The master updates its local database and sends an update request to the replicas.
- After receiving acknowledgments from the majority of replicas in the cell, the master sends an acknowledgment to the client who initiated the write.



Question: While the master is waiting for acknowledgments, are other clients allowed to read the file?

Answer: During the time the master is waiting for the acknowledgments from clients, the file is treated as ‘uncachable.’ This means that the clients can still read the file but will not cache it. This approach ensures that reads always get processed without any delay. This is useful because reads outnumber writes.

Question: Are clients allowed to cache locks? If yes, how is it used?

Answer: Chubby allows its clients to cache locks, which means the client can hold locks longer than necessary, hoping that they can be used again by the same client.

Question: Are clients allowed to cache open handles?

Answer: Chubby allows its clients to cache open handles. This way, if a client tries to open a file it has opened previously, only the first `open()` call goes to the master.

[← Back](#)

[Next →](#)

Master Election and Chubby Events

Database

Database

Let's learn how Chubby uses a database for storage.

We'll cover the following



- Backup
- Mirroring

Initially, Chubby used a replicated version of Berkeley DB (https://www.usenix.org/legacy/event/usenix99/full_papers/olson/olson.pdf) to store its data. Later, the Chubby team felt that using Berkeley DB exposes Chubby to more risks, so they decided to write a simplified custom database with the following characteristics:

- Simple key/value database using write-ahead logging and snapshotting.
- Atomic operations only and no general transaction support.
- Database log is distributed among replicas using Paxos.

Backup

For recovery in case of failure, all database transactions are stored in a transaction log (a write-ahead log). As this transaction log can become very large over time, every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS server in a different building. The use of a separate building ensures both that the backup will survive building damage, and that the backups introduce no cyclic dependencies in the system; a GFS cell in the same building potentially might rely on the Chubby cell for electing its master.

- Once a snapshot is taken, the previous transaction log is deleted. Therefore, at any time, the complete state of the system is determined by the last snapshot together with the set of transactions from the transaction log.
- Backup databases are used for disaster recovery and to initialize the database of a newly replaced replica without placing a load on other replicas.

Mirroring

Mirroring is a technique that allows a system to automatically maintain multiple copies.

- Chubby allows a collection of files to be mirrored from one cell to another.
- Mirroring is commonly used to copy configuration files to various computing clusters distributed around the world.
- Mirroring is fast because the files are small.
- Event mechanism informs immediately if a file is added, deleted, or modified.
- Usually, changes are reflected in dozens of mirrors worldwide under a second.
- Unreachable mirror remains unchanged until connectivity is restored. Updated files are then identified by comparing their checksums.
- A special “global” cell subtree `/ls/global/master` that is mirrored to the subtree `/ls/cell/replica` in every other Chubby cell.
- Global cell is special because its replicas are located in widely separated parts of the world. Global cell is used for:
 - Chubby’s own access control lists (ACLs).

- Various files in which Chubby cells and other systems advertise their presence to monitoring services.
- Pointers to allow clients to locate large data sets such as Bigtable cells, and many configuration files for other systems.

← Back

Caching

Next →

Scaling Chubby

Scaling Chubby

This lesson will explain different techniques that Chubby uses for scaling.

We'll cover the following



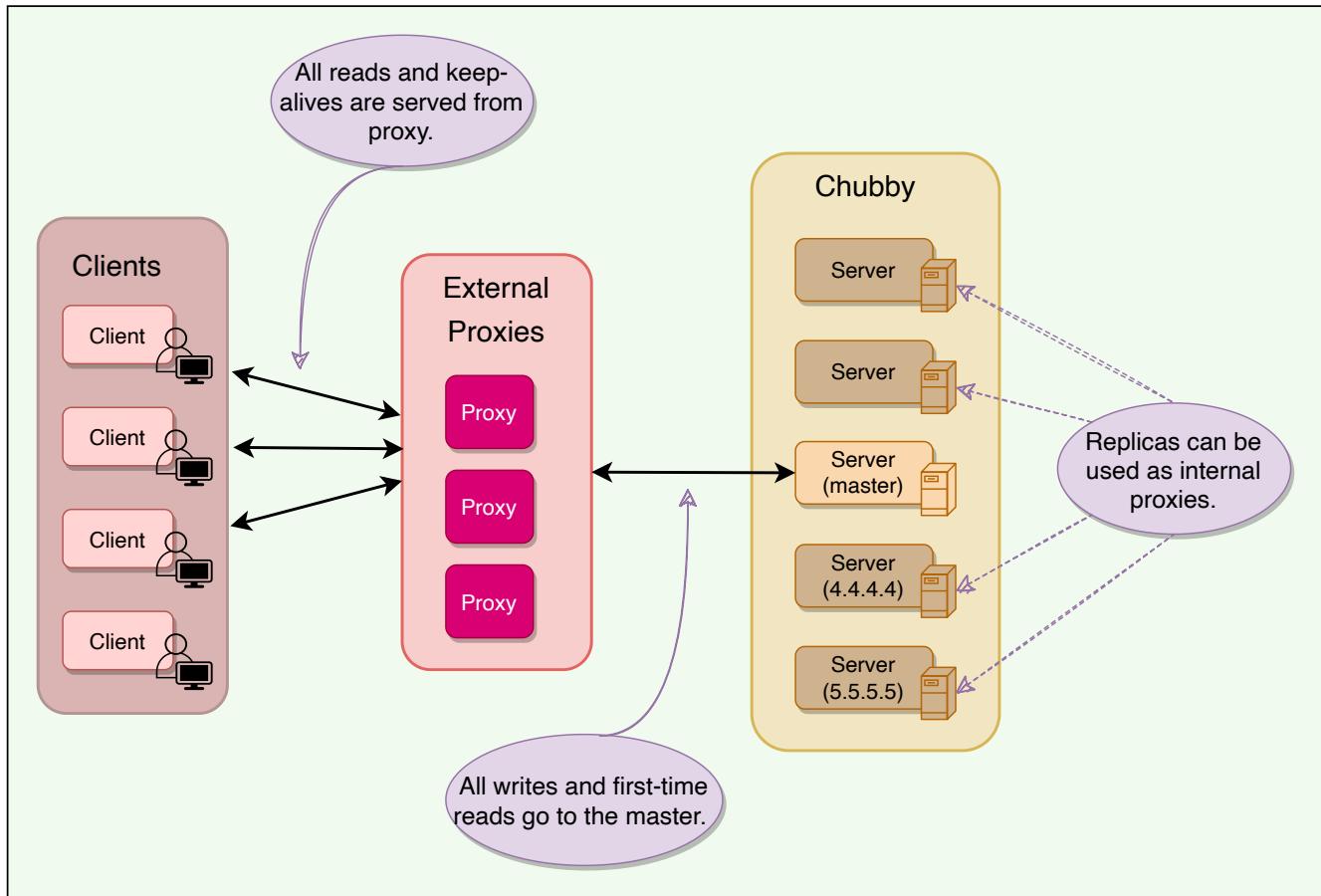
- Proxies
- Partitioning
- Learnings

Chubby's clients are individual processes, so Chubby handles more clients than expected. At Google, 90,000+ clients communicating with a single Chubby server is one such example. The following techniques have been used to reduce the communication with the master:

- **Minimize Request Rate:** Create more chubby cells so that clients almost always use a nearby cell (found with DNS) to avoid reliance on remote machines.
- **Minimize KeepAlives load:** KeepAlives are by far the dominant type of request; increasing the client lease time (from 12s to 60s) results in less load on KeepAlive.
- **Caching:** Chubby clients cache file data, metadata, handles, and any absences of files.
- **Simplified protocol-conversions:** Add servers that translate the Chubby protocol into a less complicated protocol. Proxies and partitioning are two such examples that help Chubby scale further and are discussed below.

Proxies

A proxy is an additional server that can act on behalf of the actual server.



Chubby proxies

A Chubby proxy can handle KeepAlives and read requests. If a proxy handles ' N ' clients, KeepAlive traffic is reduced by a factor of ' N .' All writes and first-time reads pass through the cache to reach the master. This means that proxy adds an additional RPC for writes and first-time reads. This is acceptable as Chubby is a read-heavy service.

Partitioning

Chubby's interface (files & directories) was designed such that namespaces can easily be partitioned between multiple Chubby cells if needed. This would result in reduced read/write traffic for any partition, for example:

- `ls/cell/foo` and everything in it, can be served by one Chubby cell, and
- `ls/cell/bar` and everything in it, can be served by another Chubby cell

There are some scenarios in which partitioning does not improve:

- When a directory is deleted, a cross partition call might be required.
- Partition does not necessarily reduce the KeepAlive traffic.
- Since ACLs can be stored in one partition only, so a cross partition call might be required to check for ACLs.

Learnings

Lack of aggressive caching: Initially, clients were not caching the absence of files or open file handles. An abusive client could write loops that retry indefinitely when a file is not present or poll a file by opening it and closing it repeatedly when one might expect they would open the file just once. Chubby educated its users to make use of aggressive caching for such scenarios.

Lack of quotas: Chubby was never intended to be used as a storage system for large amounts of data, so it has no storage quotas. In hindsight, this was naive. To handle this, Chubby later introduced a limit on file size (256kBytes).

Publish/subscribe: There have been several attempts to use Chubby's event mechanism as a publish/subscribe system. Chubby is a strongly consistent system, and the way it maintains a consistent cache makes it a slow and

inefficient choice for publish/subscribe. Chubby developers caught and stopped such uses early on.

Developers rarely consider availability: Developers generally fail to think about failure probabilities and wrongly assume that Chubby will always be available. Chubby educated its clients to plan for short Chubby outages so that it has little or no effect on their applications.

[← Back](#)

Database

[Next →](#)

Summary: Chubby

Summary: Chubby

Here is a quick summary of Chubby for you!

We'll cover the following

^

- Summary
- System design patterns
- References and further reading

Summary

1. Chubby is a **distributed lock service** used inside **Google** systems.
2. It provides **coarse-grained locking** (for hours or days) and is not recommended for fine-grained locking (for seconds) scenarios. Due to this nature, it is more suited for high-read and rare write scenarios.
3. Chubby's primary use cases include naming service, leader election, small files storage, and distributed locks.
4. A Chubby Cell basically refers to a **Chubby cluster**. A chubby cell has more than one server (typically 3-5 at least) known as replicas.
5. Using **Paxos**, one server is chosen as the master at any point and handles all the requests. If the master fails, another server from replicas becomes the master.
6. Each replica maintains a small database to store files/directories/locks. Master directly writes to its own local database, which gets synced asynchronously to all the replicas for fault tolerance.

7. Client applications use a Chubby library to communicate with the replicas in the chubby cell using RPC.
8. Like Unix, Chubby file system interface is basically a tree of files & directories (collectively called nodes), where each directory contains a list of child files and directories.
9. **Locks:** Each node can act as an advisory reader-writer lock in one of the following two ways:
 - **Exclusive:** One client may hold the lock in exclusive (write) mode.
 - **Shared:** Any number of clients may hold the lock in shared (reader) mode.
10. **Ephemeral nodes** are used as temporary files, and act as an indicator to others that a client is alive. Ephemeral nodes are also deleted if no client has them open. Ephemeral directories are also deleted if they are empty.
11. **Metadata:** Metadata for each node includes Access Control Lists (ACLs), monotonically increasing 64-bit numbers, and checksum.
12. **Events:** Chubby supports a simple event mechanism to let its clients subscribe for a variety of events for files such as a lock being acquired or a file being edited.
13. **Caching:** To reduce read traffic, Chubby clients cache file contents, node metadata, and information on open handles in a consistent, write-through cache in the client's memory.
14. **Sessions:** Clients maintain sessions by sending KeepAlive RPCs to Chubby. This constitutes about 93% of the example Chubby cluster's requests.
15. **Backup:** Every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS file server in a different building.
16. **Mirroring:** Chubby allows a collection of files to be mirrored from one cell to another. Mirroring is used most commonly to copy configuration files to various computing clusters distributed around the world.

System design patterns

Here is a summary of system design patterns used in Chubby.

- **Write-Ahead Log:** For fault tolerance and to handle a master crash, all database transactions are stored in a transaction log.
- **Quorum:** To ensure strong consistency, Chubby master sends all write requests to the replicas. After receiving acknowledgments from the majority of replicas in the cell, the master sends an acknowledgment to the client who initiated the write.
- **Generation clock:** To disregard requests from the previous master, every newly-elected master in Chubby uses ‘Epoch number’, which is simply a monotonically increasing number to indicate a server’s generation. This means if the old master had an epoch number of ‘1’, the new one would have ‘2’. This ensures that the new master will not respond to any old request which was sent to the previous master.
- **Lease:** Chubby clients maintain a time-bound session lease with the master. During this time interval, the master guarantees to not terminate the session unilaterally.

References and further reading

- Chubby paper (<https://research.google/pubs/pub27897/>)
- Chubby architecture video (<https://www.youtube.com/watch?v=PqItueBaiRg>)
- Chubby vs. ZooKeeper (<https://www.youtube.com/watch?v=zokwJeukDrI>)

- Hierarchical Chubby (https://www.scs.stanford.edu/17au-cs244b/labs/projects/bohn_dauterman.pdf)
- Bigtable (<https://research.google/pubs/pub27898/>)
- Google File System (<https://research.google/pubs/pub51/>)

← Back

Scaling Chubby

Next →

Quiz: Chubby

GFS: How to Design a Distributed File Storage System?

Google File System: Introduction

Let's explore Google File System and its use cases.

We'll cover the following



- Goal
- What is Google File System (GFS)?
- Background
- GFS use cases
- APIs

Goal

Design a distributed file system to store huge files (terabyte and larger). The system should be scalable, reliable, and highly available.

What is Google File System (GFS)?

GFS is a scalable distributed file system developed by Google for its large data-intensive applications.

Background

GFS was built for handling batch processing on large data sets and is designed for system-to-system interaction, not user-to-system interaction.