

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.

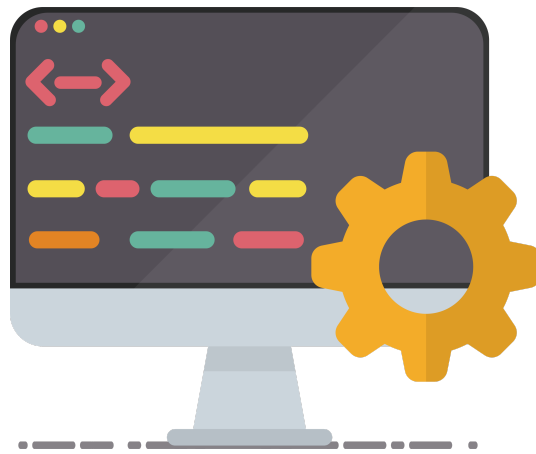
We'll cover the following



- 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**.

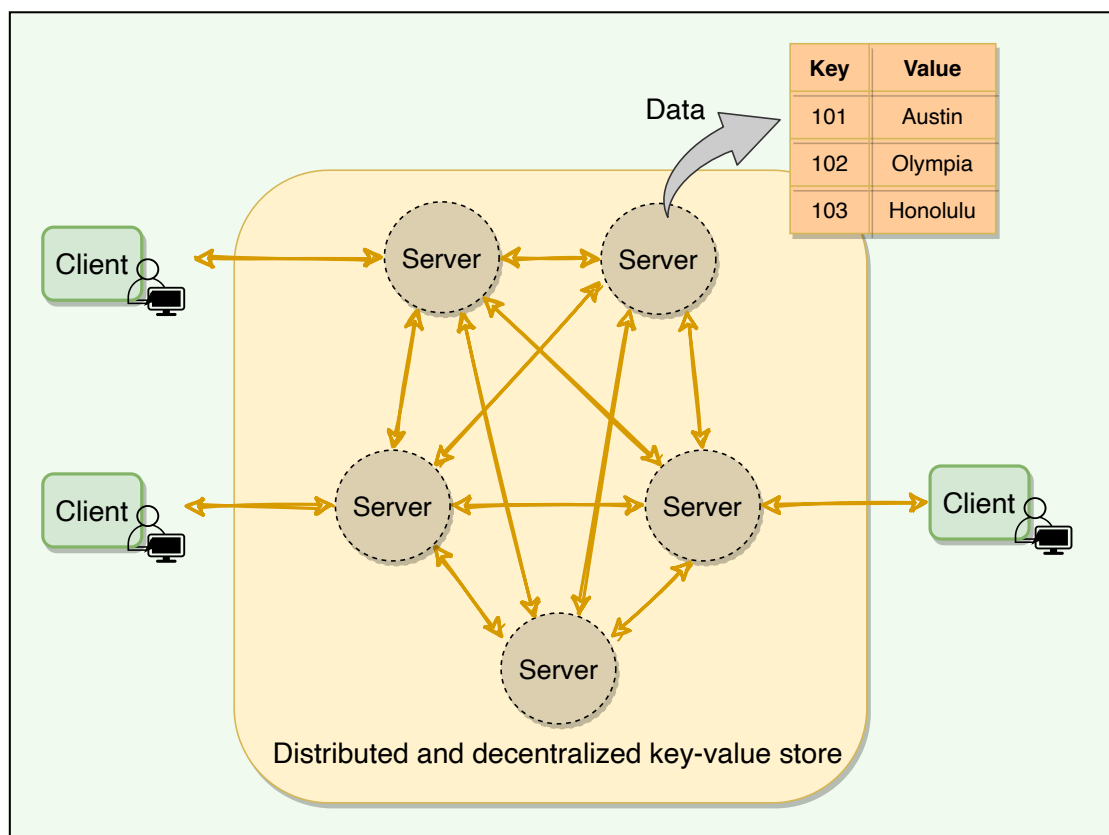
The primary motivation for designing Dynamo as a highly available system was the observation that the availability of a system directly correlates to the number of customers served. Therefore, the main goal is that the system, even when it is imperfect, should be available to the customer as it brings 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.

[← Back](#)

[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

Dynamo: Introduction

Next →

Data Partitioning

Data Partitioning

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

We'll cover the following



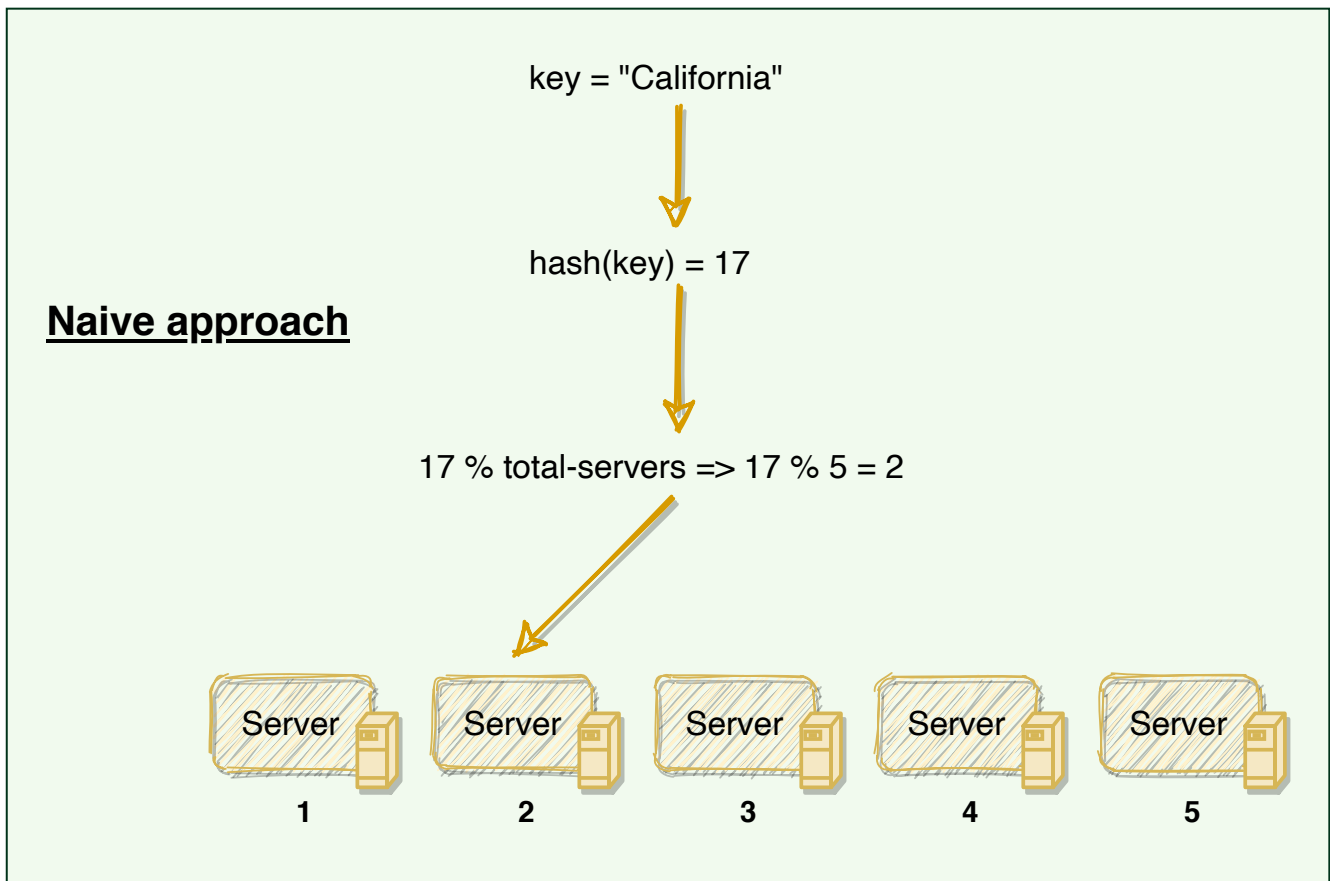
- What is data partitioning?
- Consistent hashing: Dynamo's data distribution
- Virtual nodes
 - Advantages of Vnodes

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:



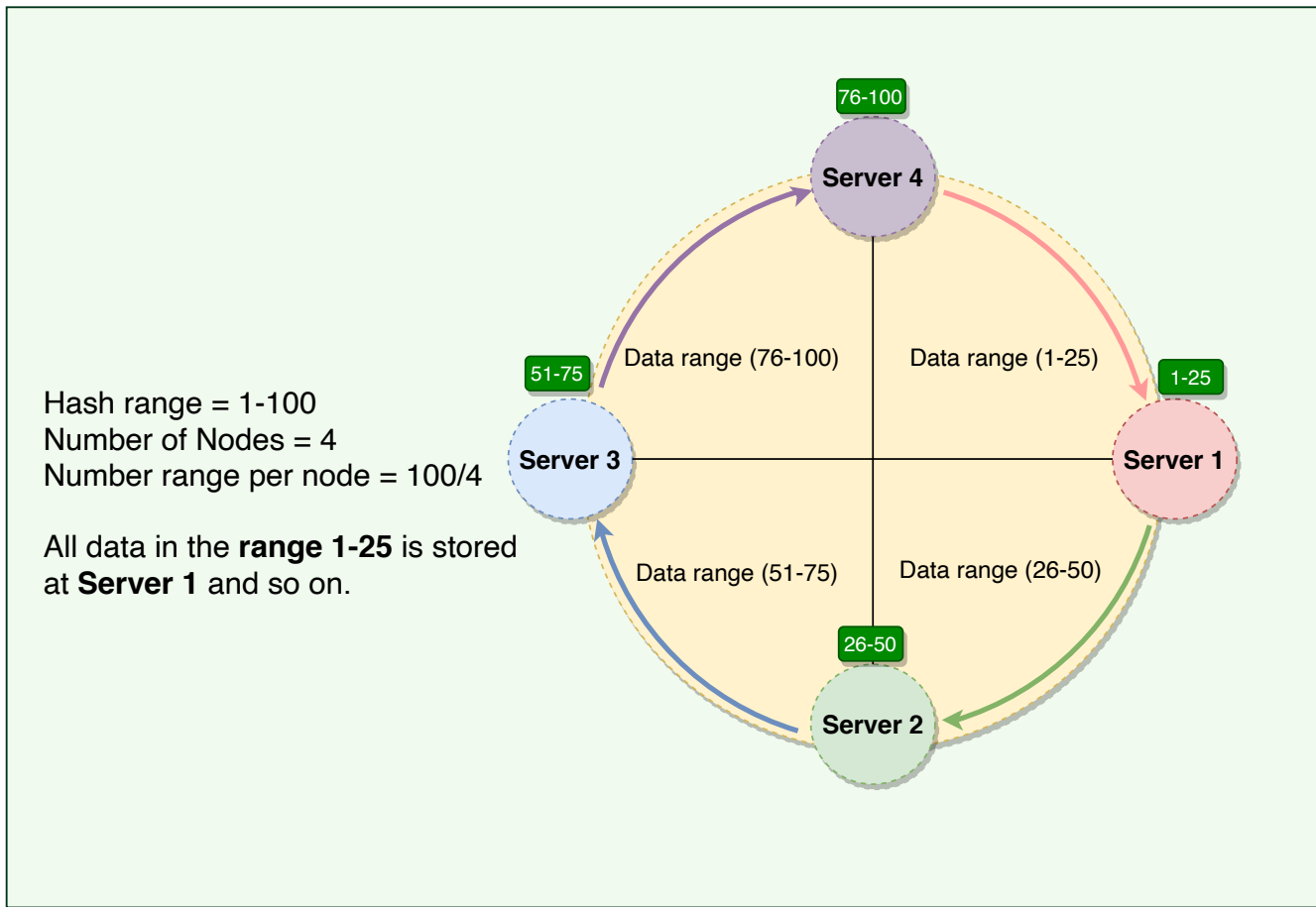
Data partitioning through simple hashing

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:

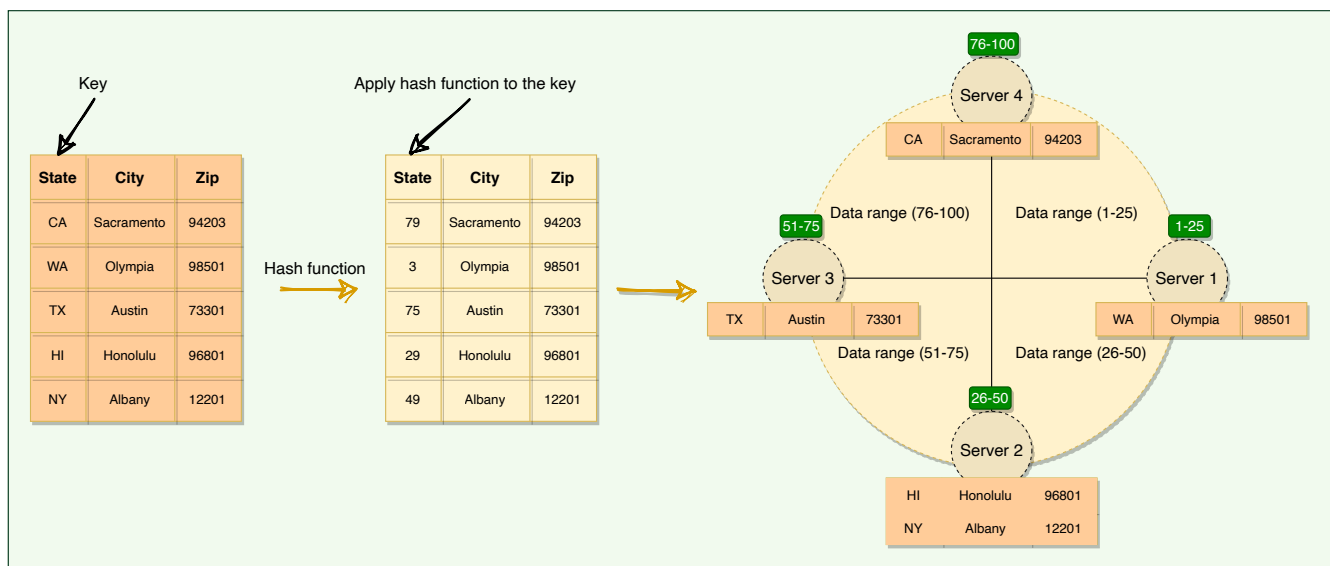
Range start: Token value

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.

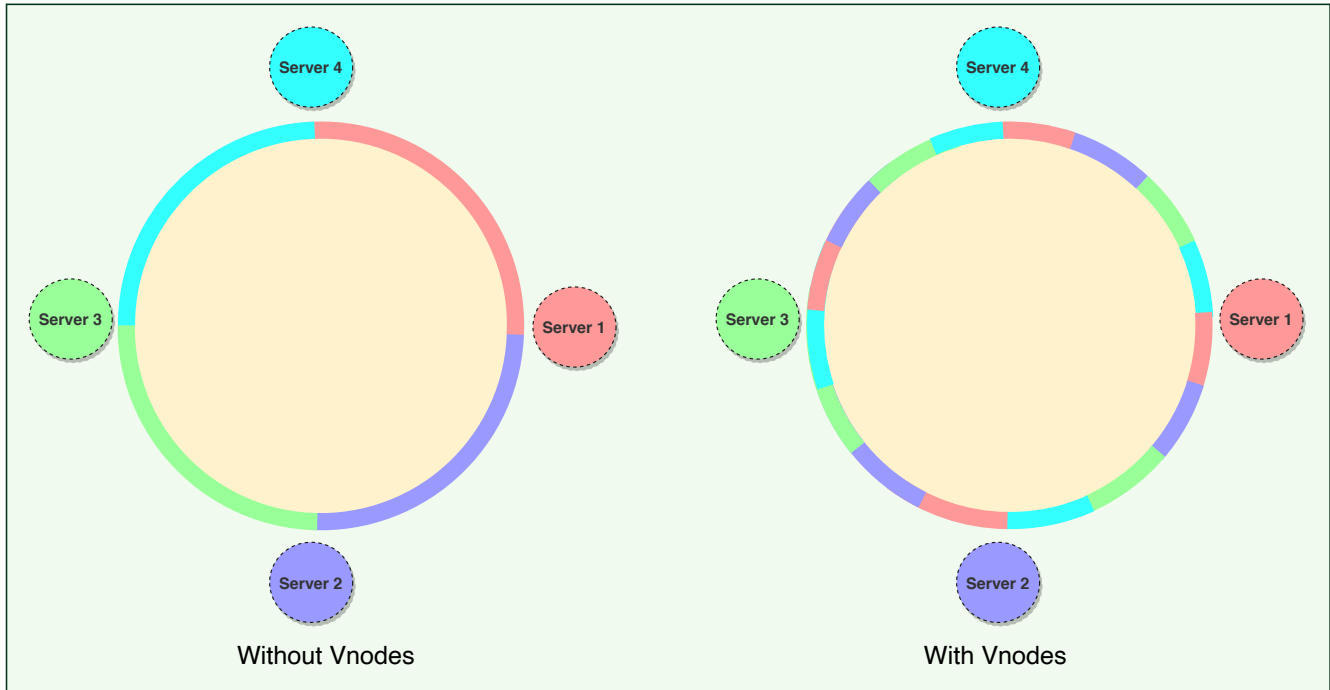
Virtual nodes

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:

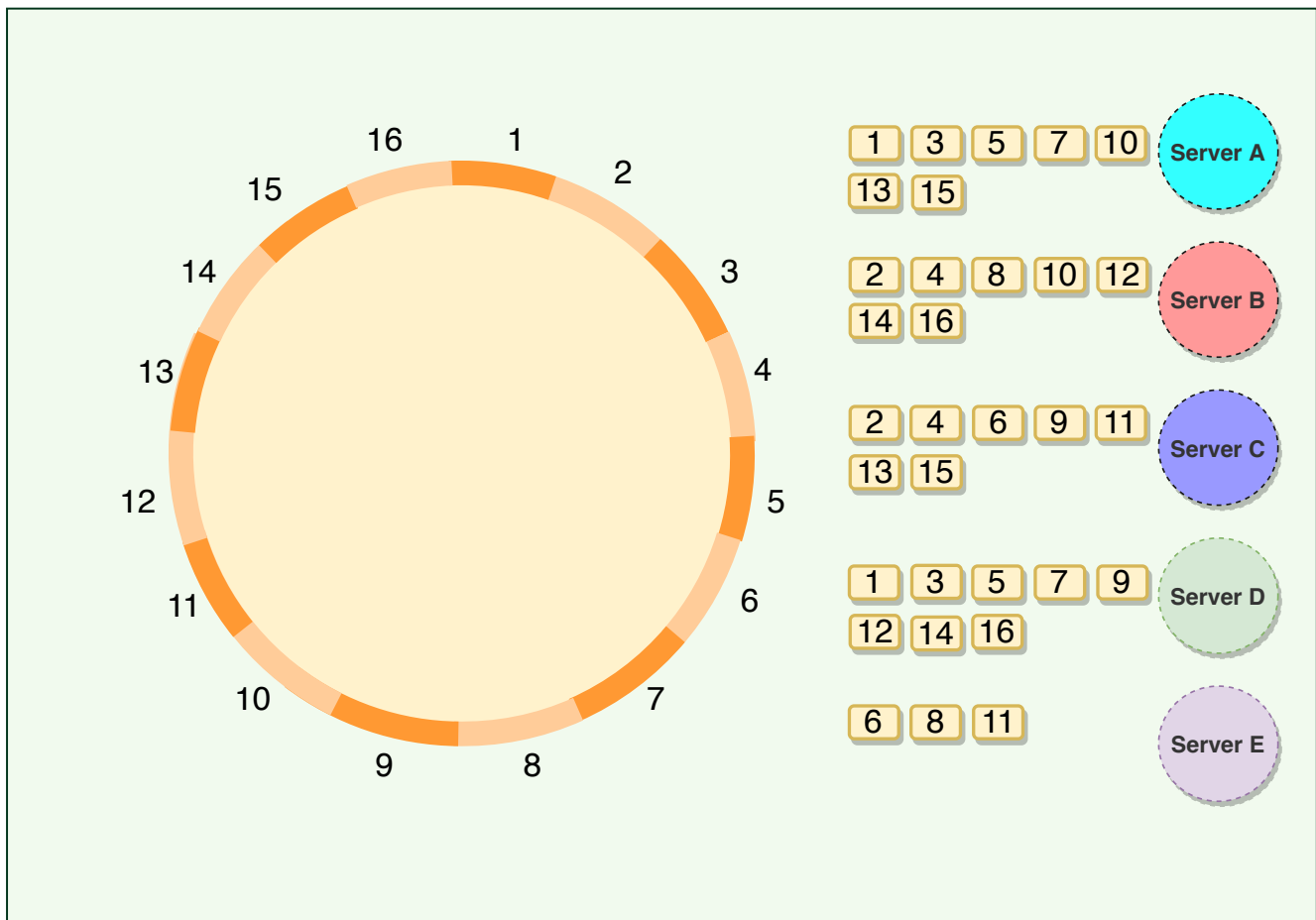
- **Adding or removing nodes:** Adding or removing nodes will result in recomputing the tokens causing a significant administrative overhead for a large cluster.
- **Hotspots:** Since each node is assigned one large range, if the data is not evenly distributed, some nodes can become hotspots.
- **Node 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.

← Back

High-level Architecture

Next →

Replication

Replication

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

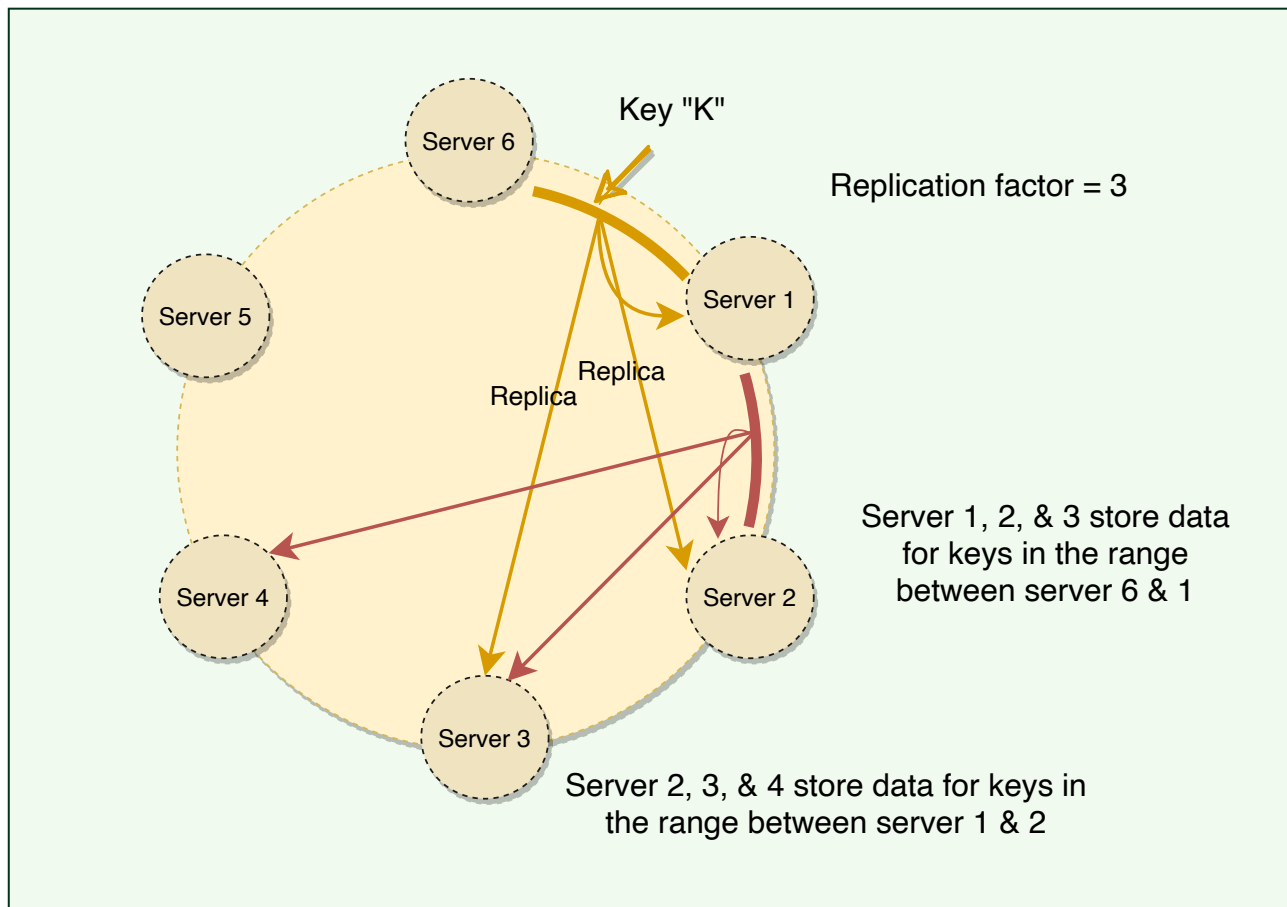
We'll cover the following



- 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.

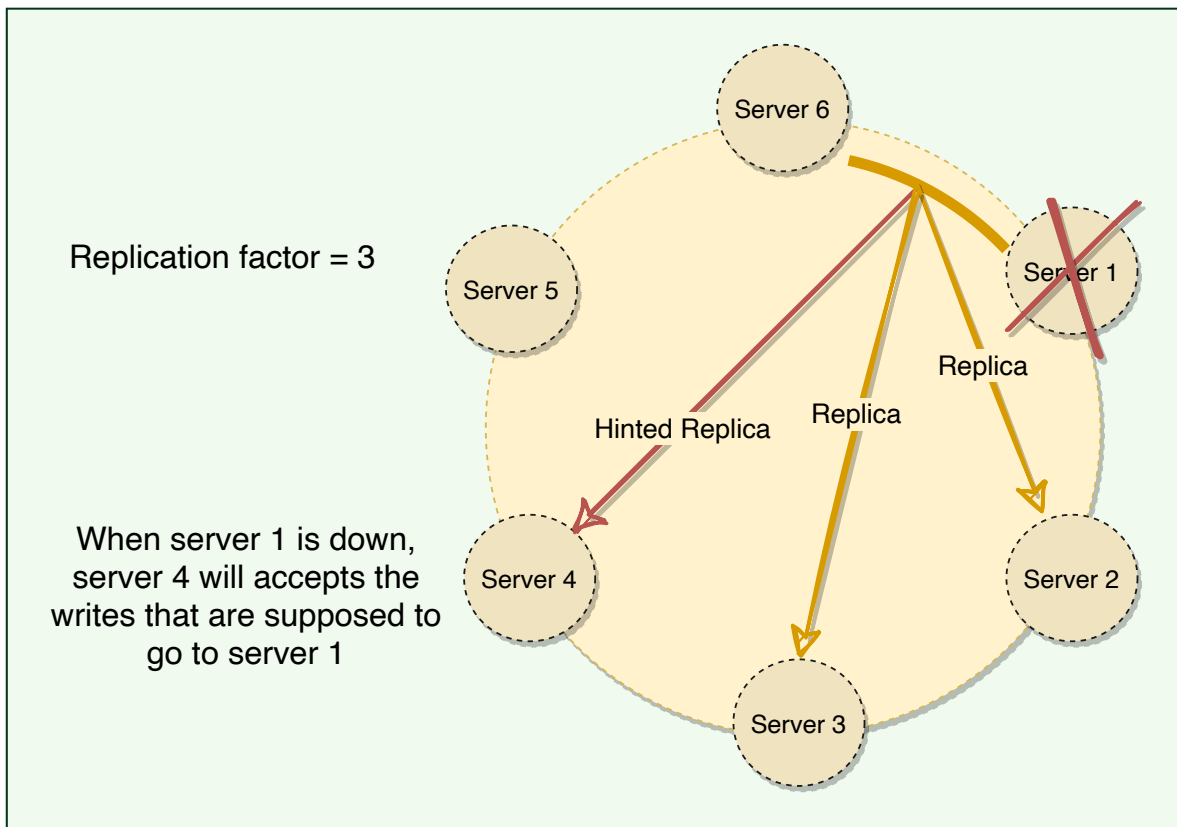
Preference List

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

Data Partitioning

Next →

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

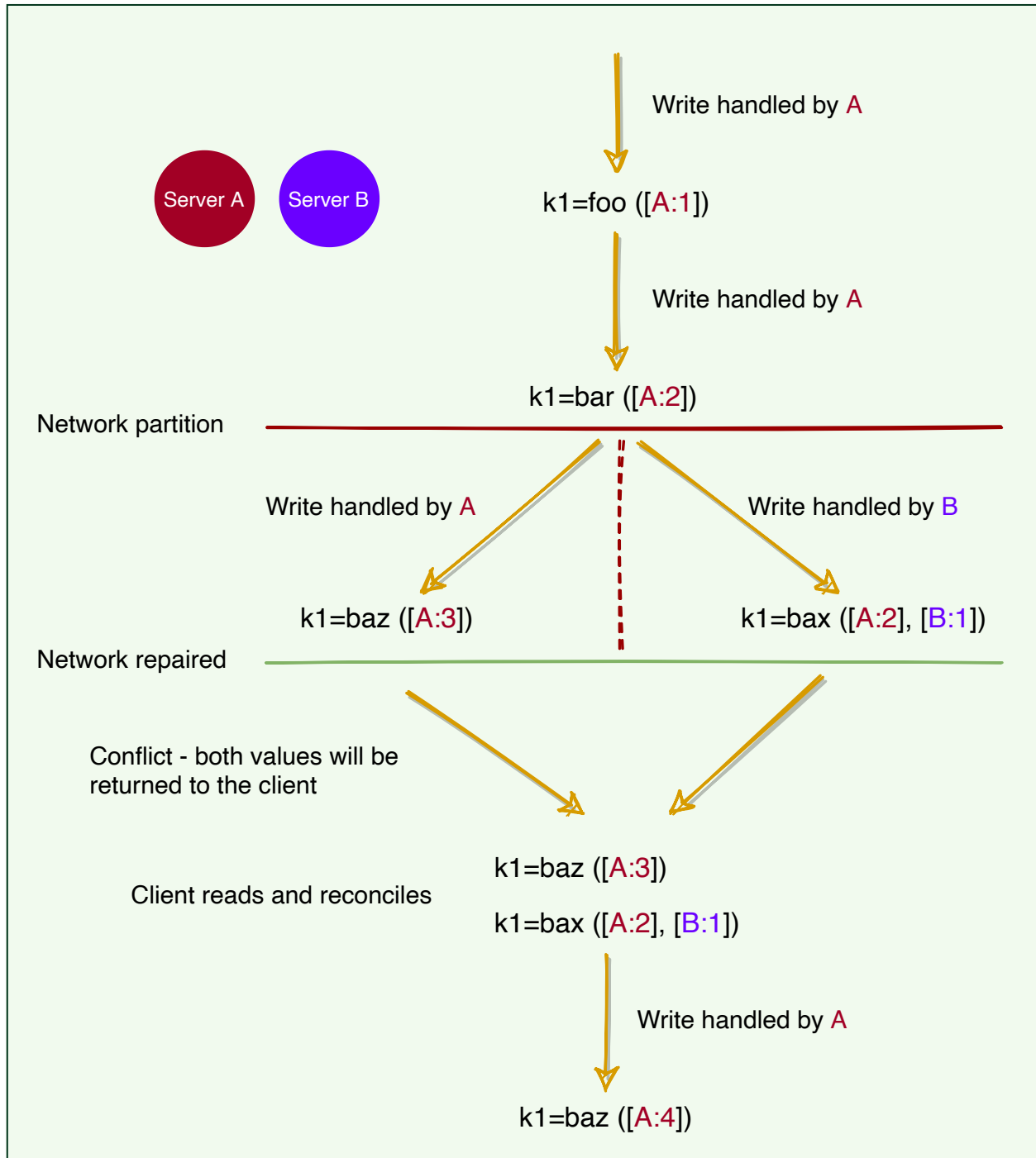
most practical techniques that help with synchronizing clocks, like NTP, still do not guarantee that every clock in a distributed system is synchronized at all times. So, without special hardware like GPS units and atomic clocks, just using wall clock timestamps is not enough.

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 k_1 , 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 k_1 , 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 k_1 , 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 k_1 , 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 k_1 . 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 →

Replication

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

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

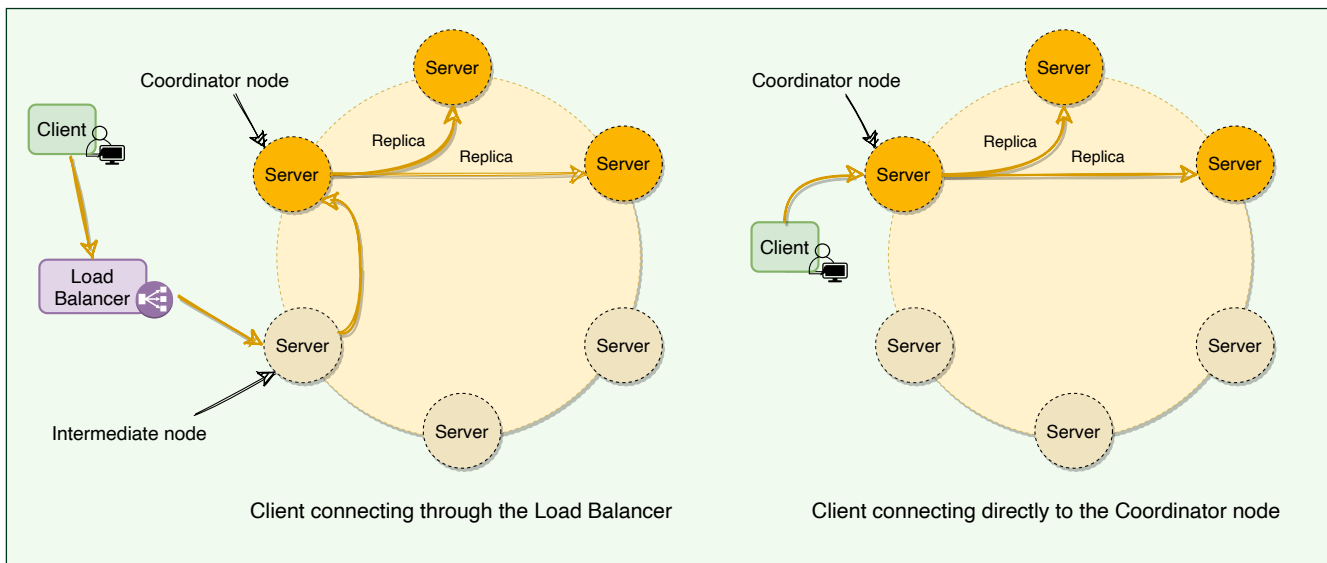
Dynamo clients can use one of the two strategies to choose a node for their get() and put() requests:

- 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

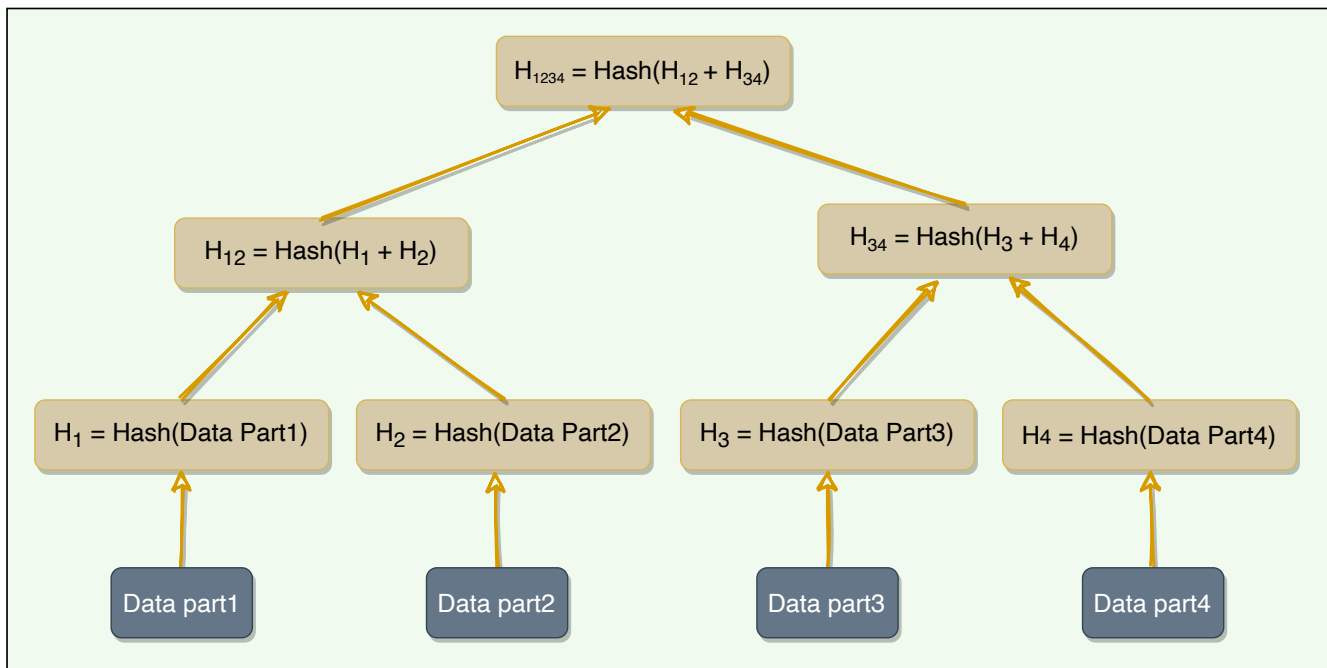


- What are Merkle trees?
- Merits and demerits of Merkle trees

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](#)

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

[Next >](#)

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

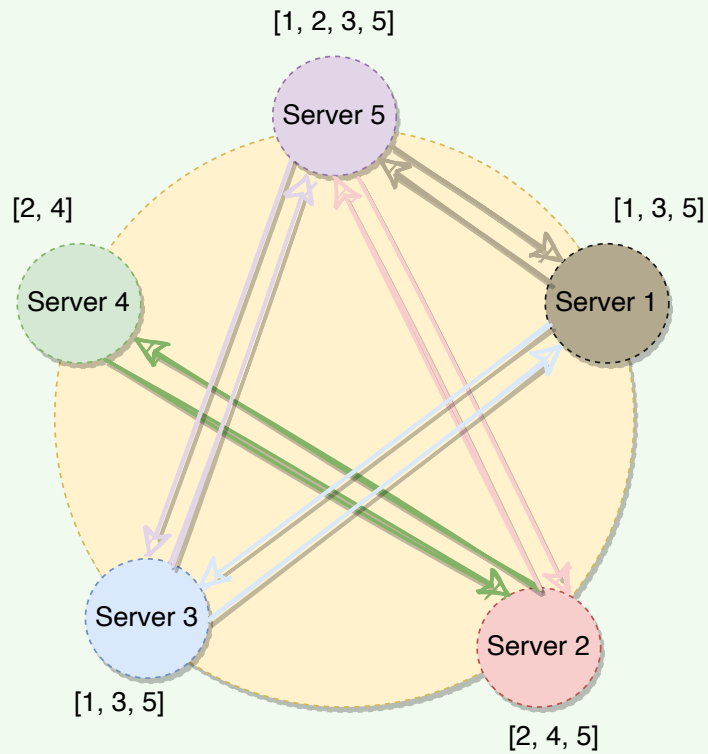
What is gossip protocol?

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.

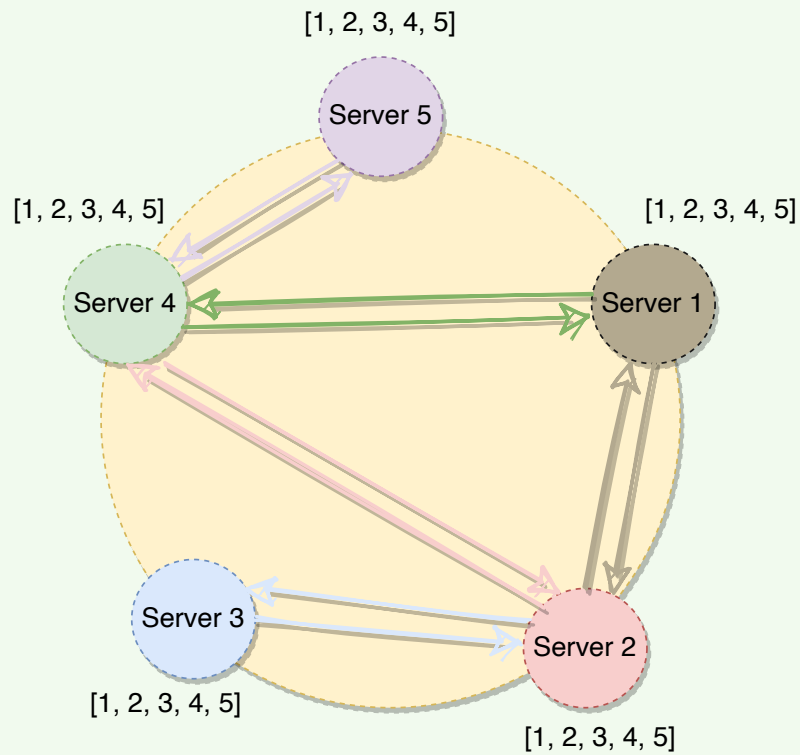
Dynamo uses **gossip protocol** that enables each node to keep track of state information about the other nodes in the cluster, like which nodes are reachable, what key ranges they are responsible for, and so on (*this is basically a copy of the hash ring*). Nodes share state information with each other 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

Next →

Mock Interview: Dynamo

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

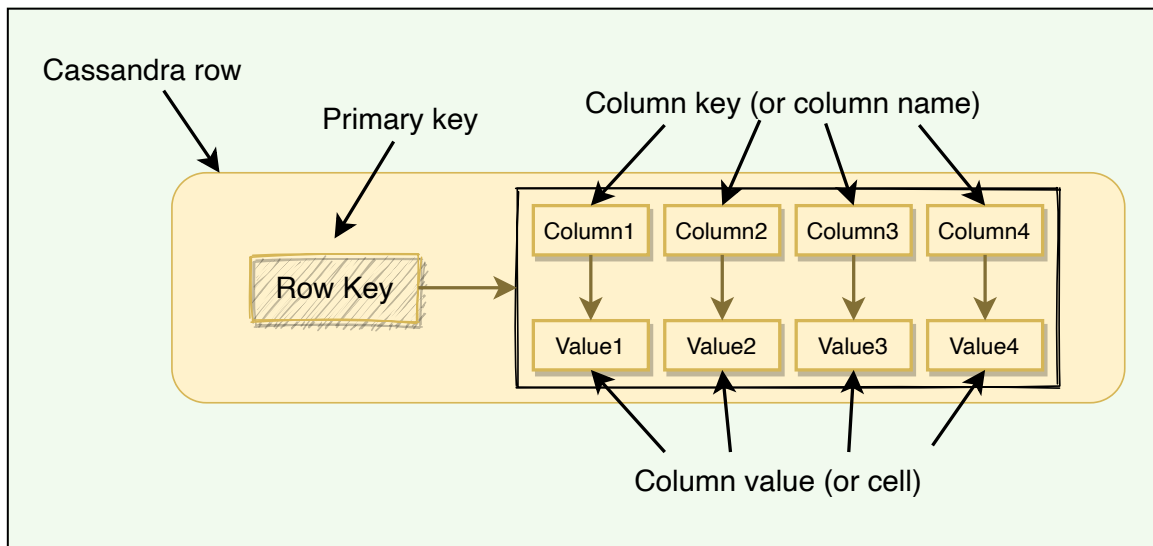
Cassandra common terms

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

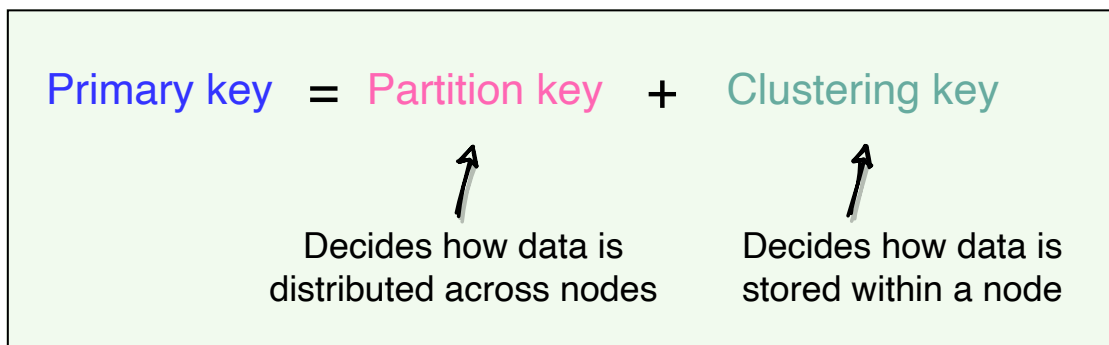
Data partitioning

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:



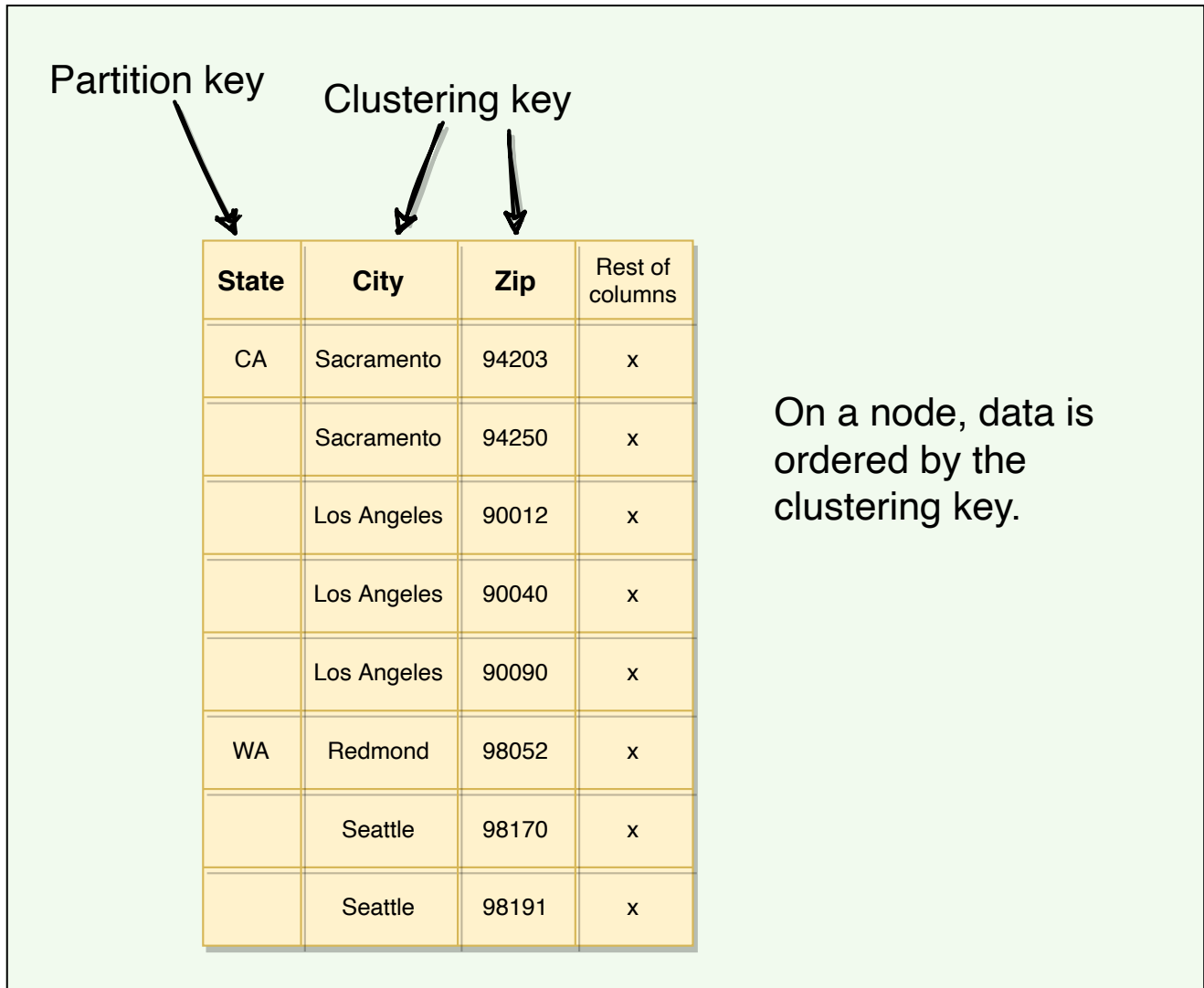
Parts of a primary key

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.

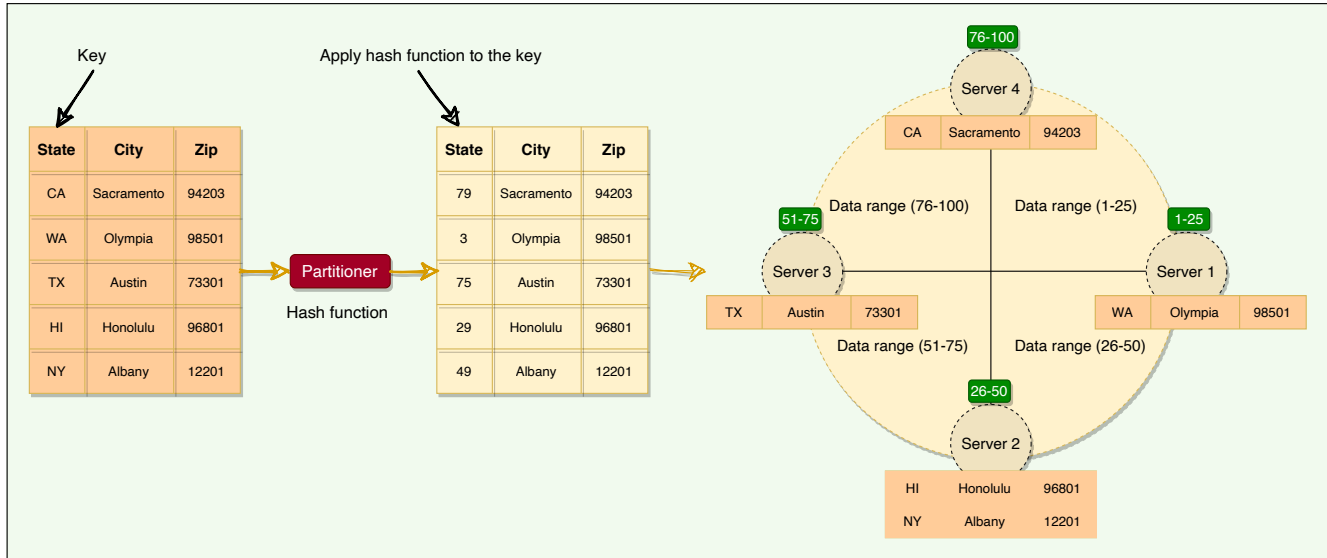


State	City	Zip	Rest of columns
CA	Sacramento	94203	x
	Sacramento	94250	x
	Los Angeles	90012	x
	Los Angeles	90040	x
	Los Angeles	90090	x
WA	Redmond	98052	x
	Seattle	98170	x
	Seattle	98191	x

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

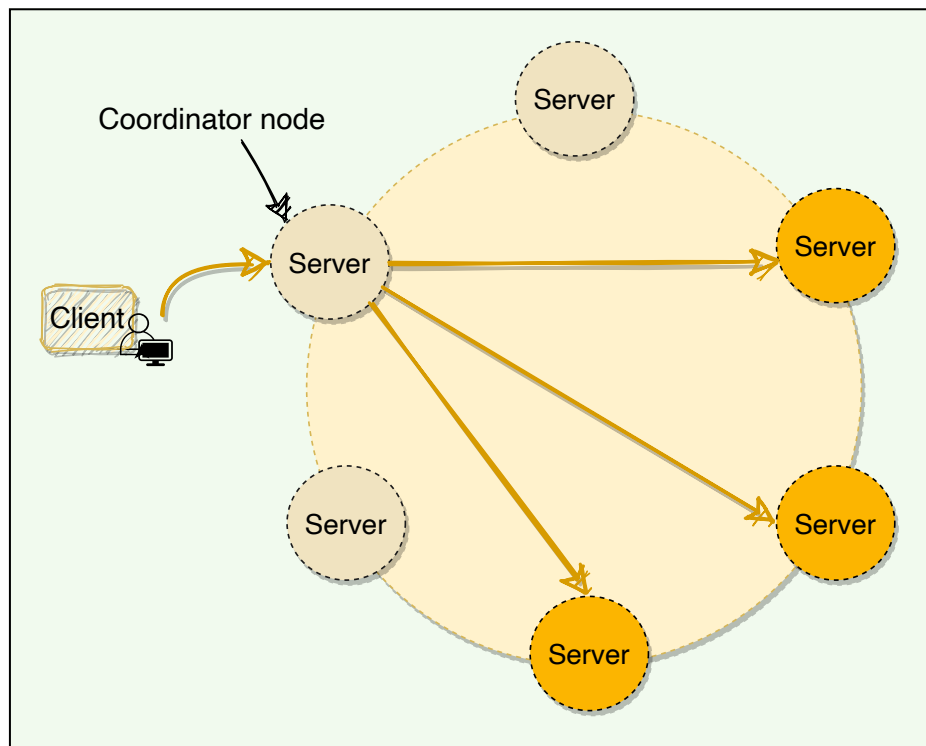
By default, Cassandra uses the Murmur3 hashing function. Murmur3 will always produce the same hash for a given partition key. This means that we can always find the node where a specific row is stored. Cassandra does allow custom hashing functions, however, once a cluster is initialized with a particular 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.

Replication

Let's explore Cassandra's replication strategy.

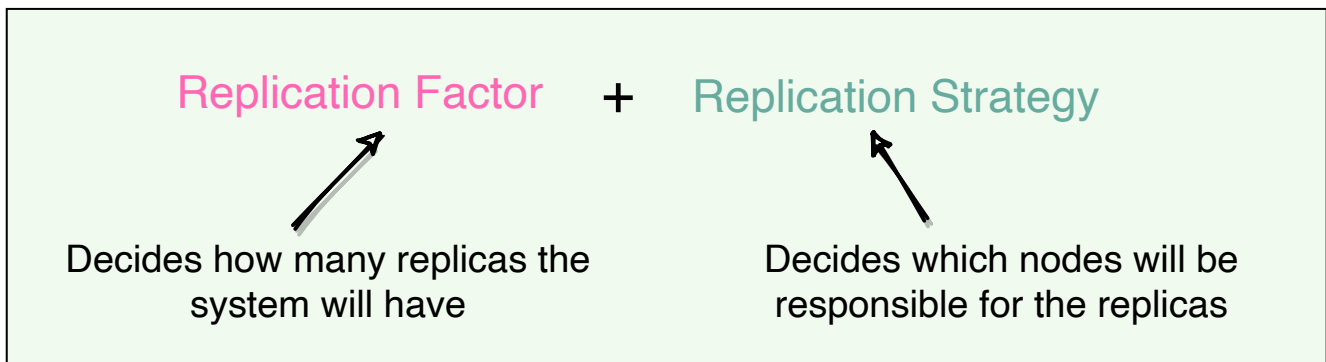
We'll cover the following



- 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

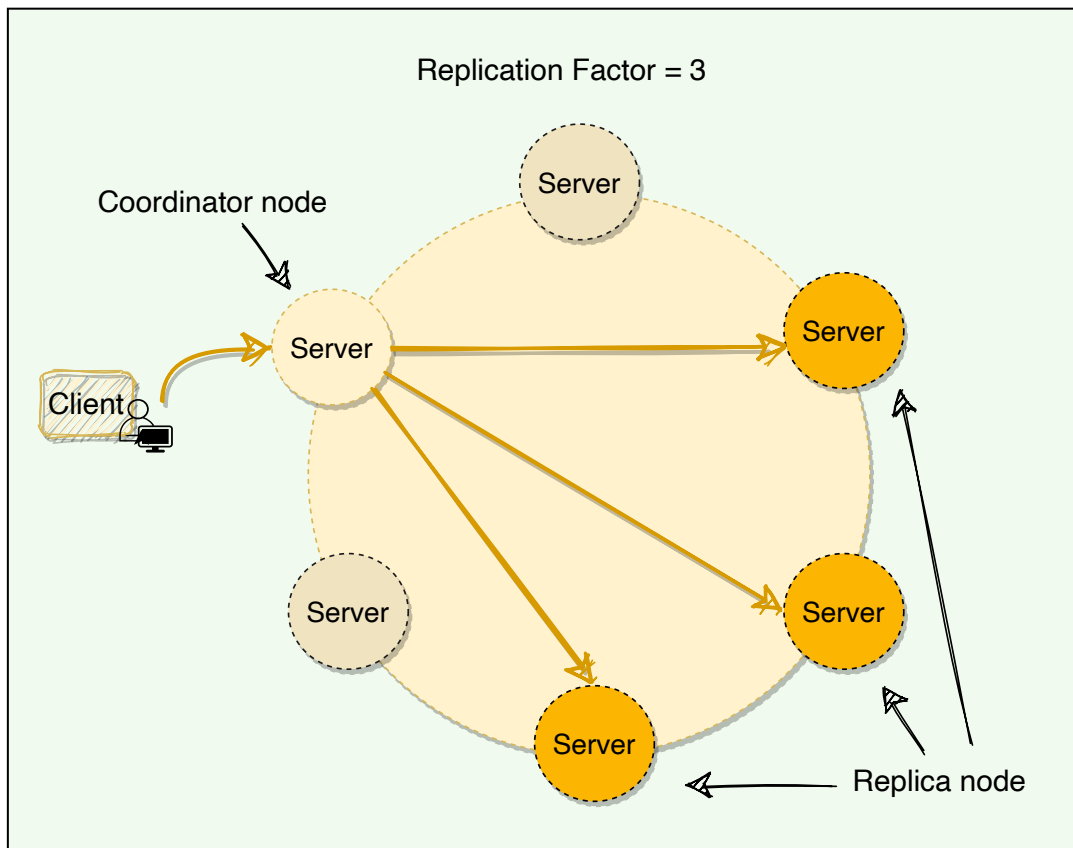
The replication factor is the number of nodes that will receive the copy of the same data. This means, if a cluster has a replication factor of 3, each row will be stored on three different nodes. Each keyspace in Cassandra can have a different replication factor.

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.

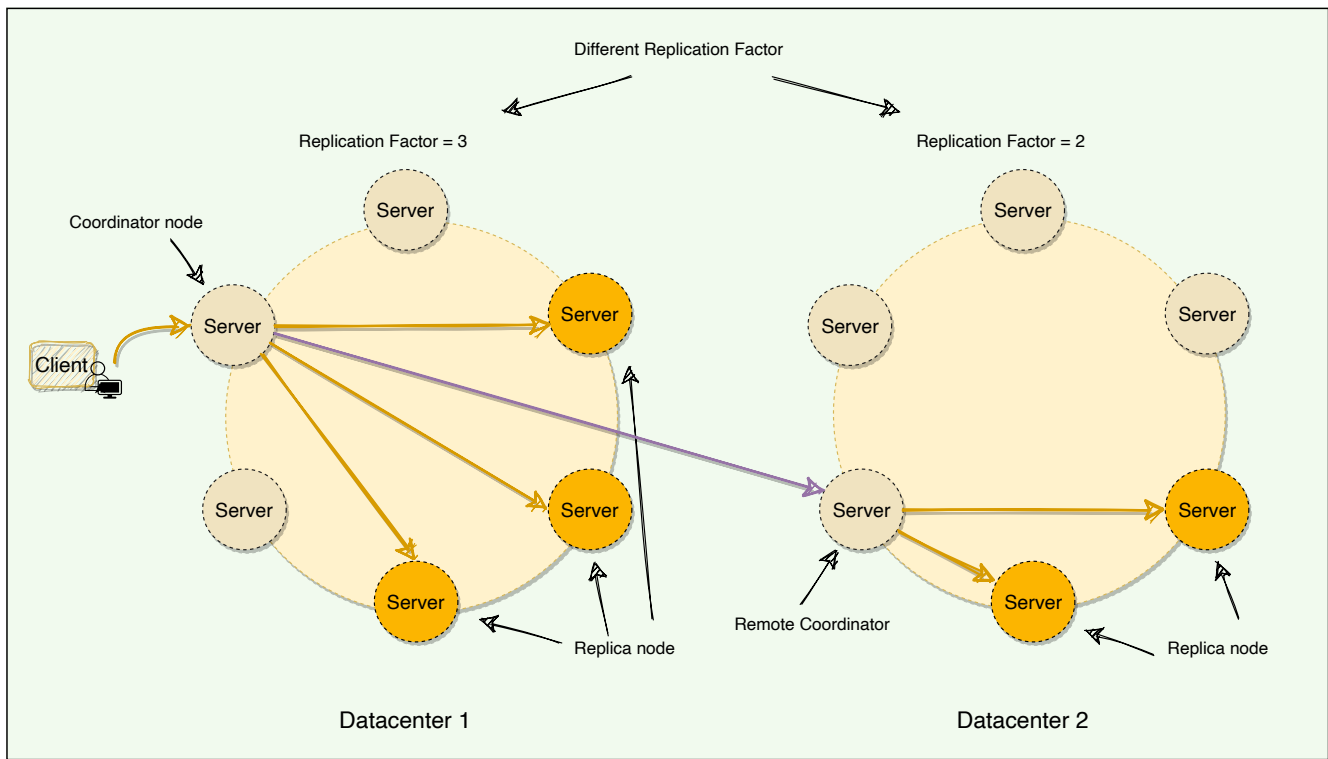


Simple replication with 3 replicas

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

Next →

High-level Architecture

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:

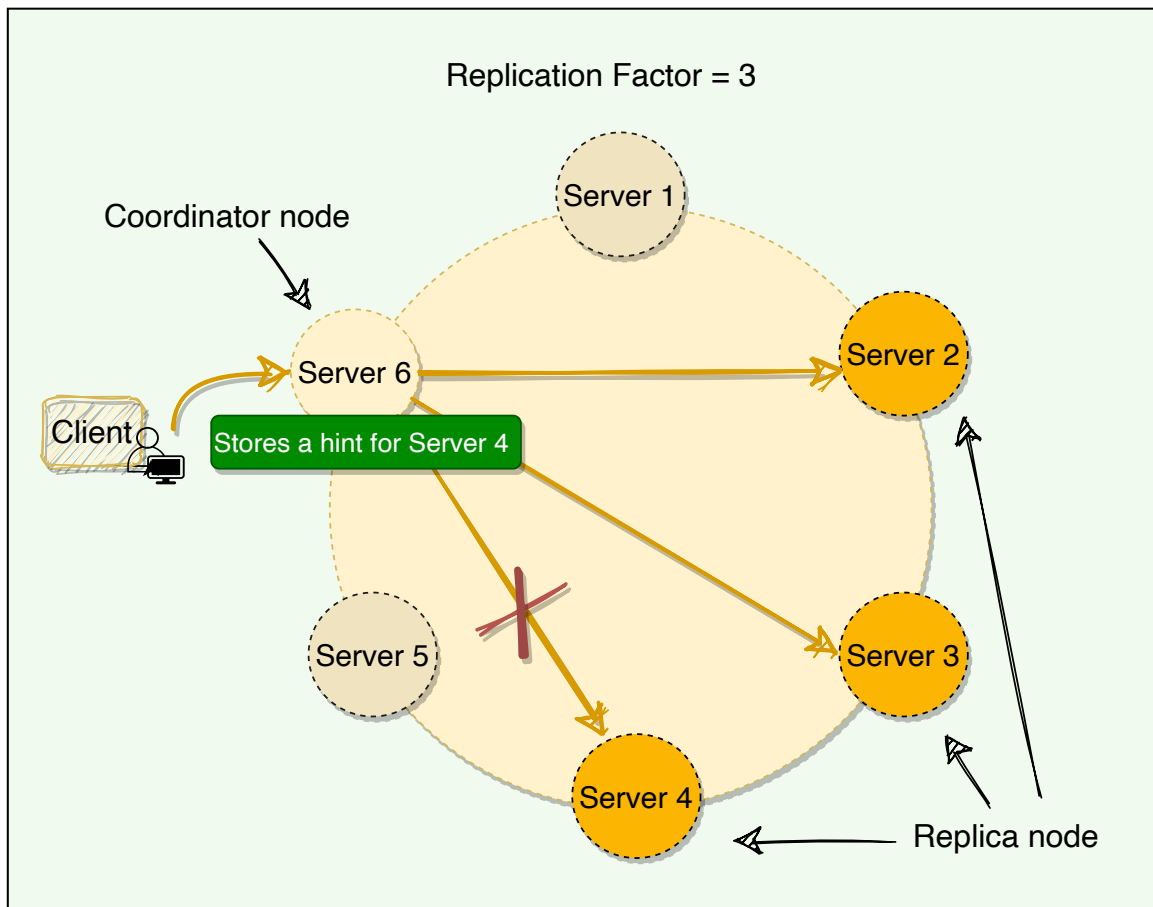
- **One or Two or Three:** The data must be written to at least the specified number of replica nodes before a write is considered successful.
- **Quorum:** The data must be written to at least a quorum (or majority) of replica nodes. Quorum is defined as $\text{floor}(RF/2 + 1)$, where RF represents 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.

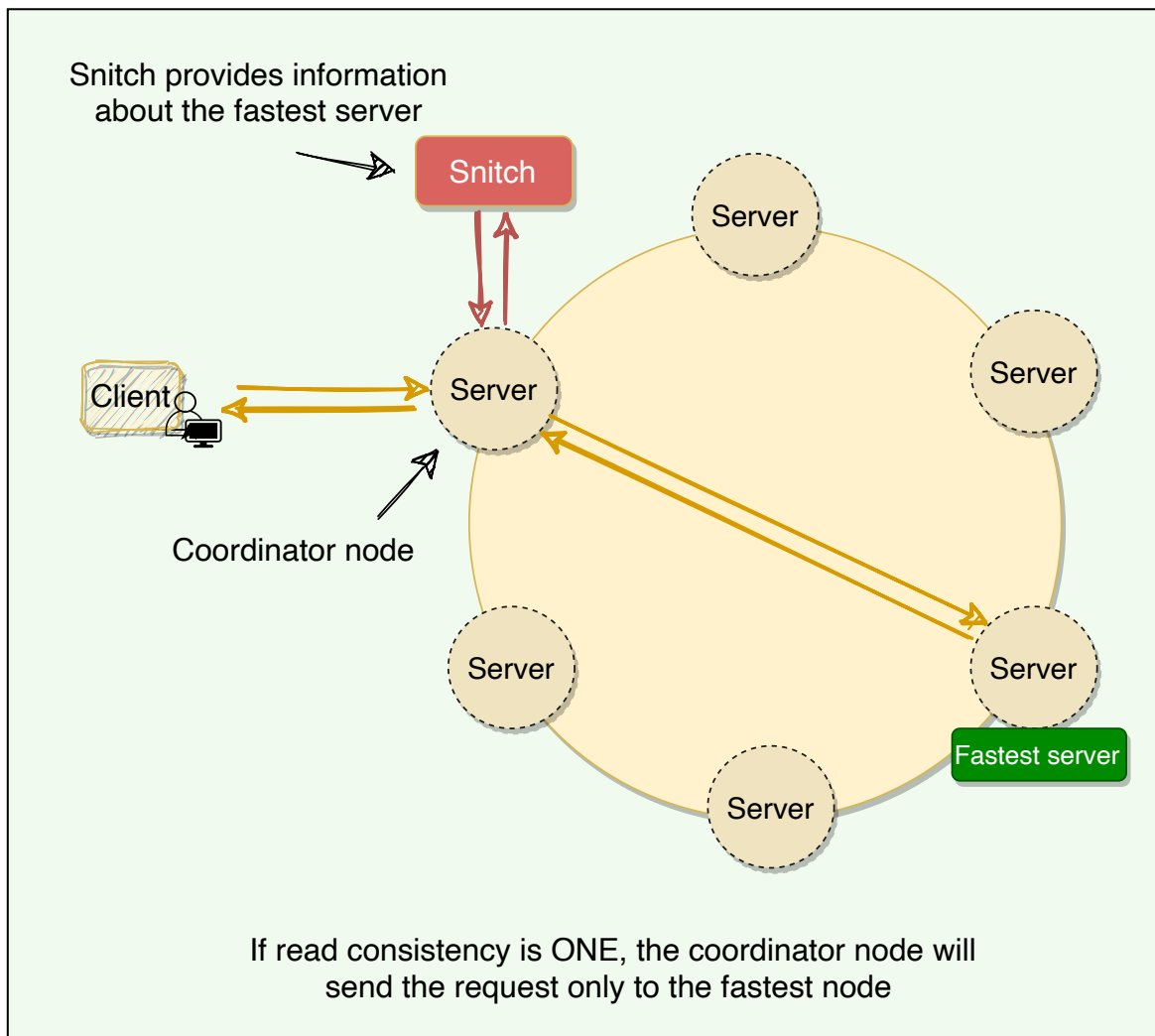
Read consistency levels

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.

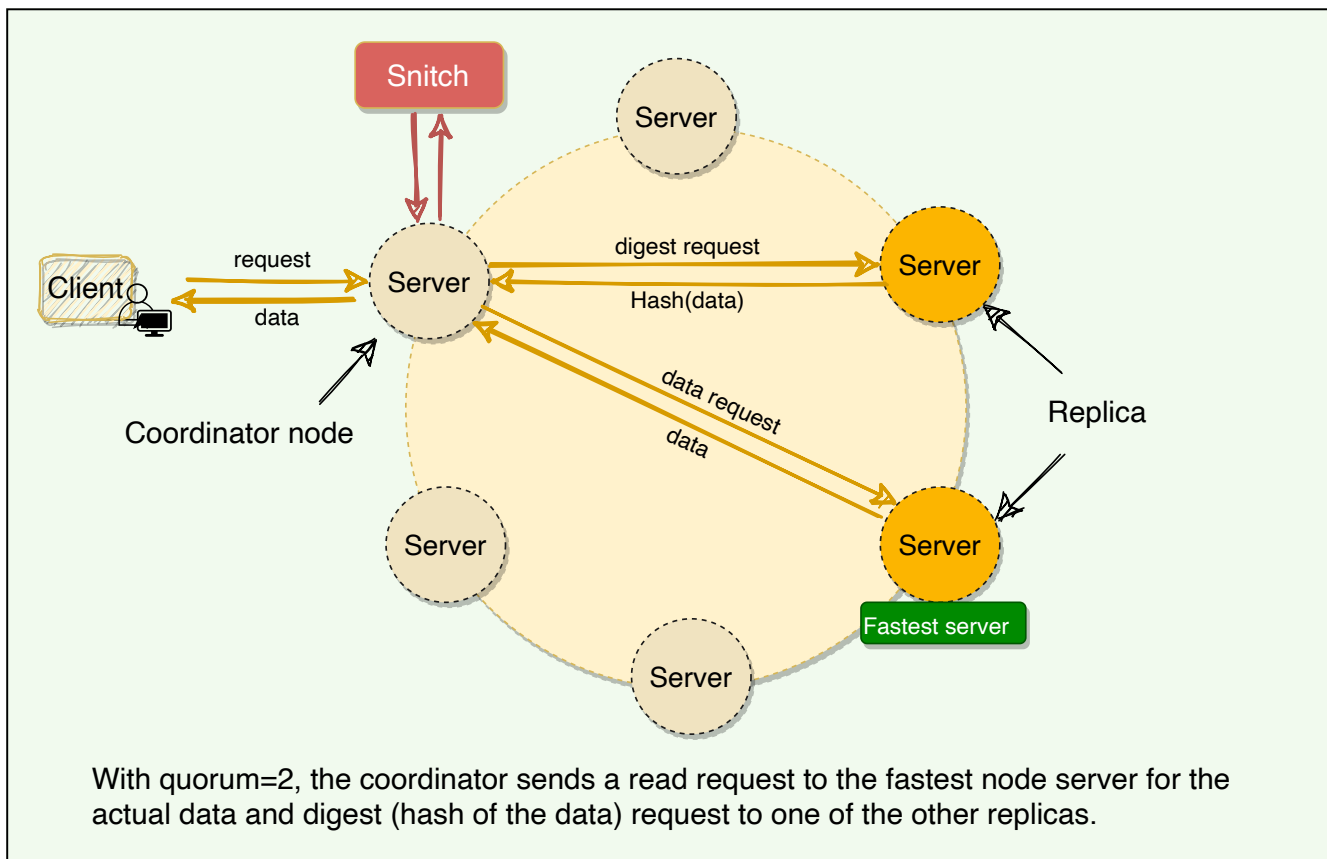
Snitch: The Snitch is an application that determines the proximity of nodes within the ring and also tells which nodes are faster. Cassandra nodes use this information to route read/write requests efficiently. We will discuss this in detail later.



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.

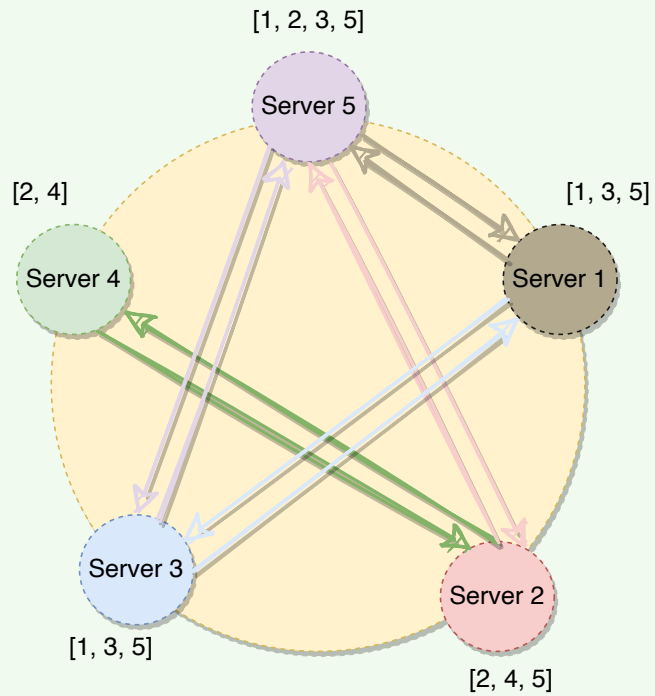
Each gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

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

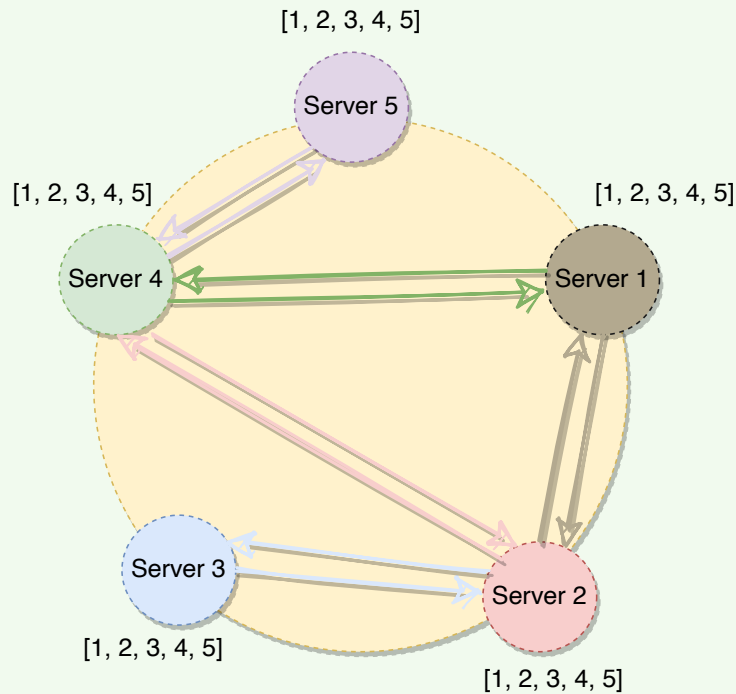
distinguish the current state of a node from its state before a restart. The generation number remains the same while the node is alive and is incremented each time the node restarts. The node receiving the gossip message can compare the generation number it knows and the gossip message's generation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

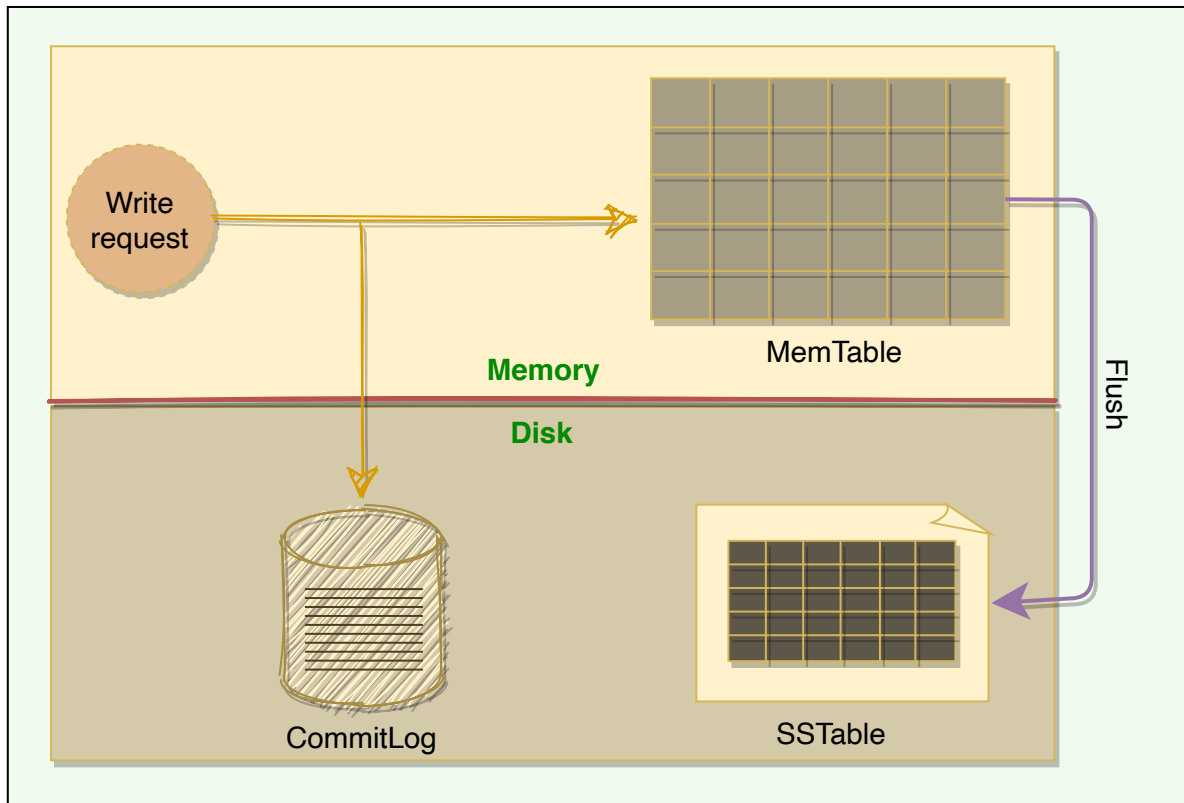
1. Each write is appended to a **commit log**, which is stored on disk.
2. Then it is written to **MemTable** in memory.
3. Periodically, MemTables are flushed to **SSTables** on the disk.
4. Periodically, compaction runs to merge SSTables.

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.



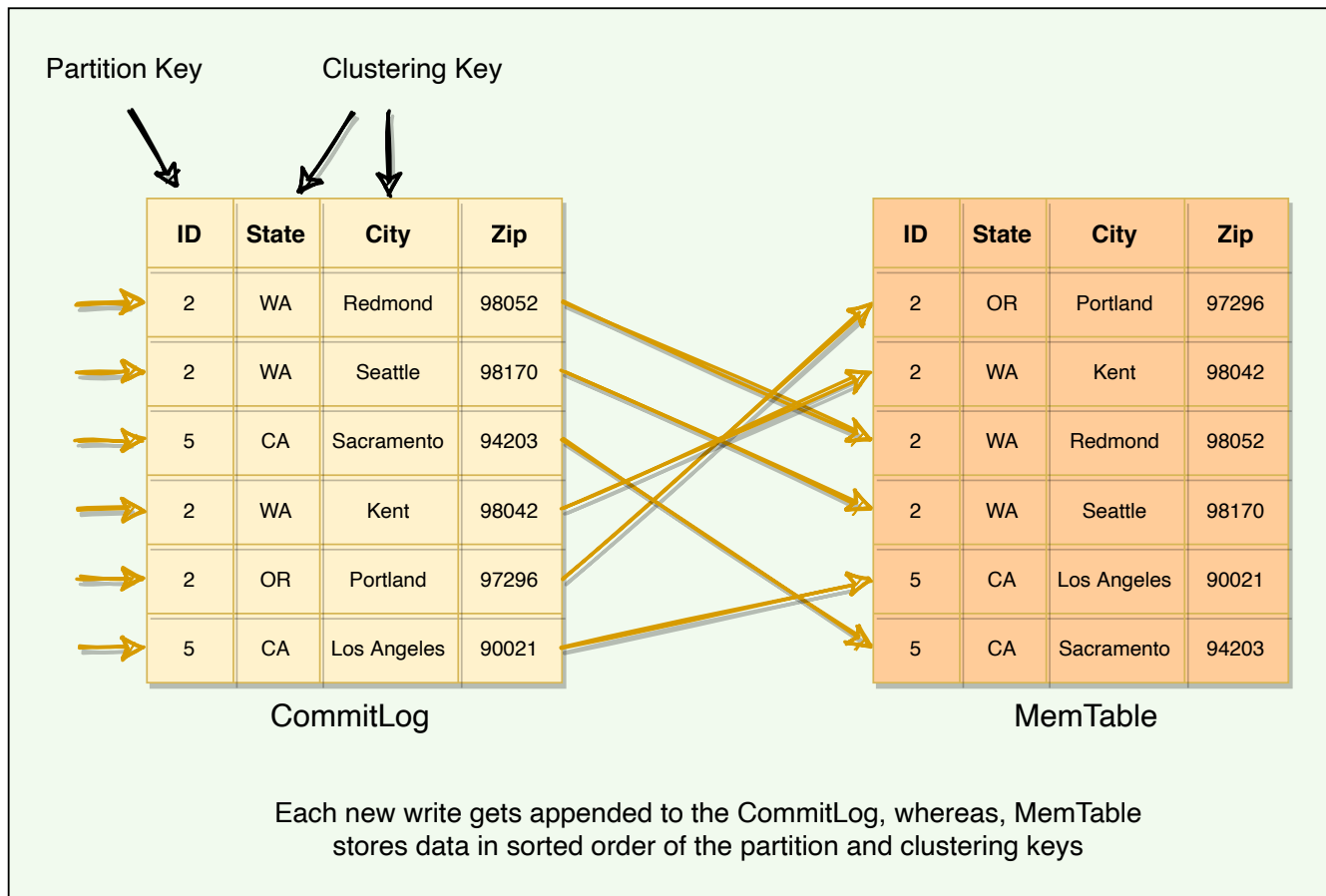
Cassandra's write path

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.

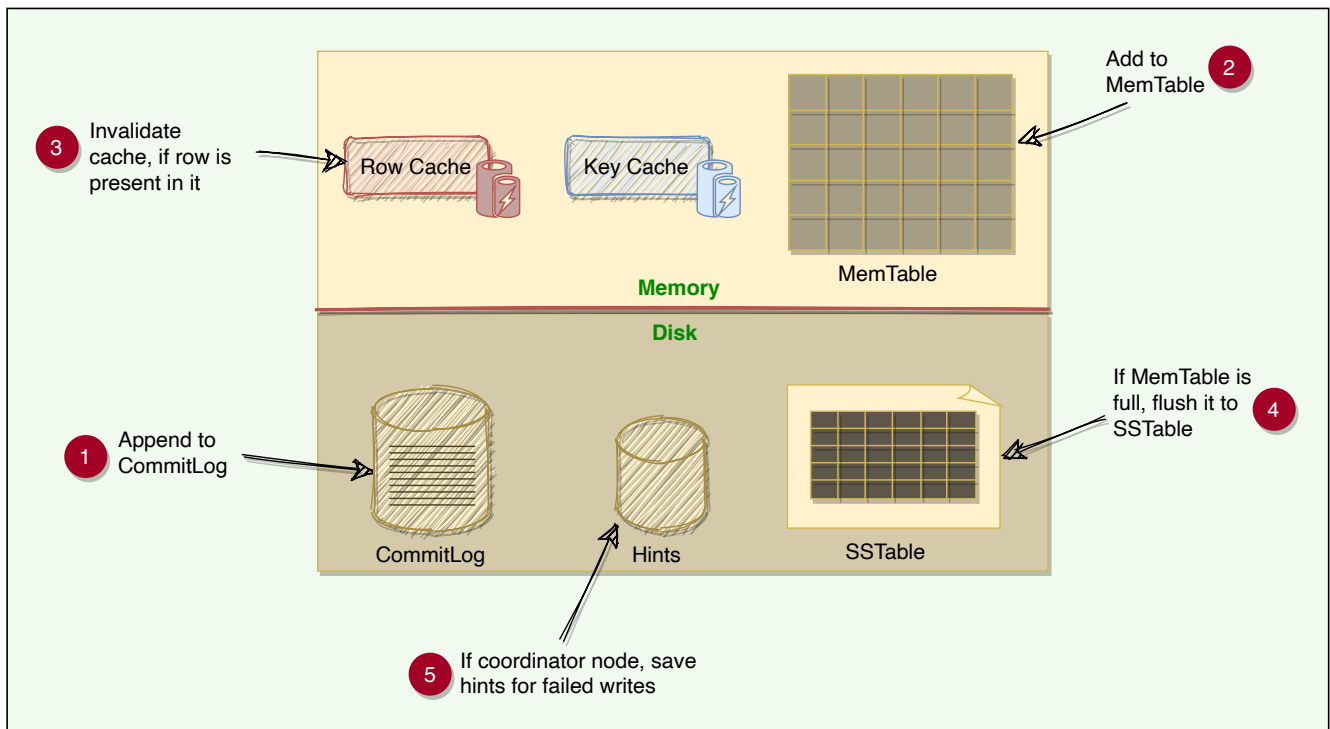
When the MemTable is flushed to SSTables, corresponding entries in the Commit Log are removed.

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.

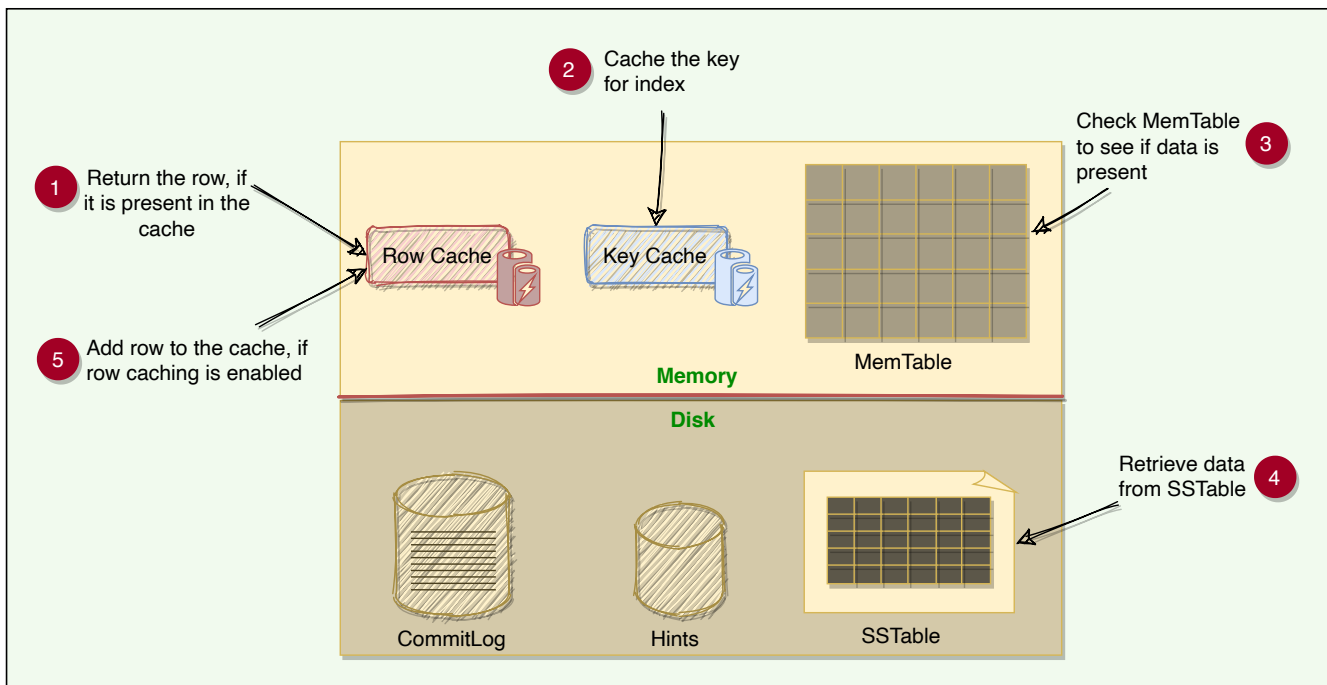
Reading from MemTable

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
Partition key = 2	2	OR	Portland	97296
	2	WA	Kent	98042
	2	WA	Redmond	98052
	2	WA	Seattle	98170
Partition key = 5	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

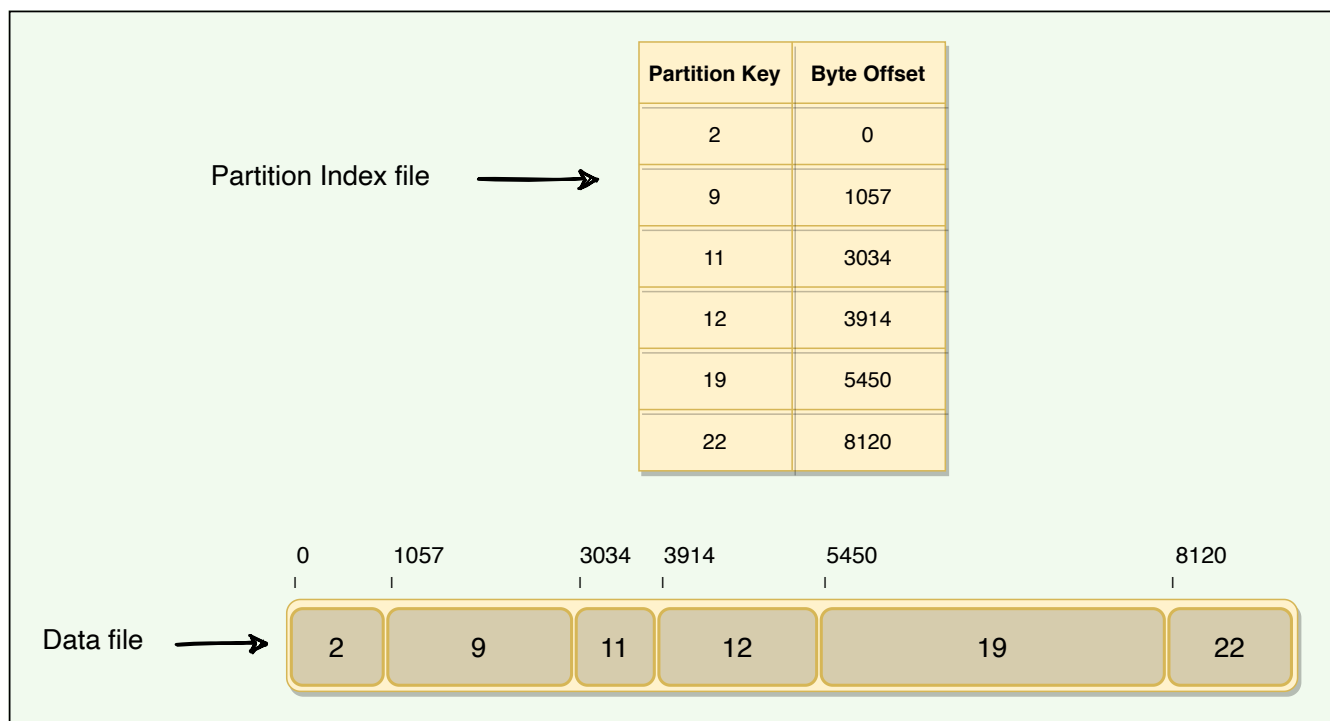
Each SStable has a Bloom filter associated with it, which tells if a particular key is present in it or not. Bloom filters are used to boost the performance of read operations. Bloom filters are very fast, non-deterministic algorithms for testing whether an element is a member of a set. They are non-deterministic 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:

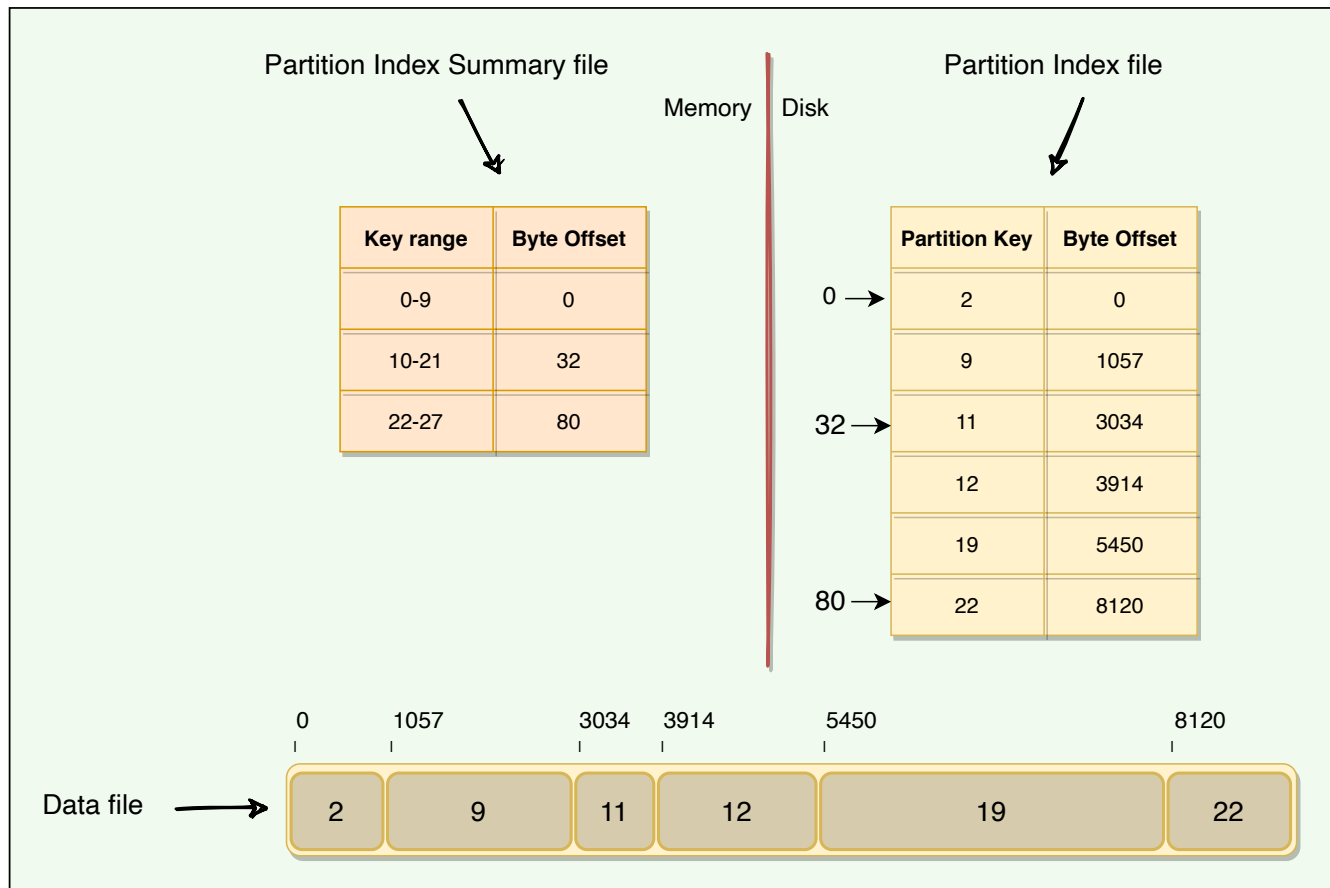
1. **Data File:** Actual data is stored in a data file. It has partitions and rows associated with those partitions. The partitions are in sorted order.
2. **Partition Index file:** Stored on disk, partition index file stores the sorted partition keys mapped to their SSTable offsets. It enables locating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



Reading from an SSTable

Partition index summary file

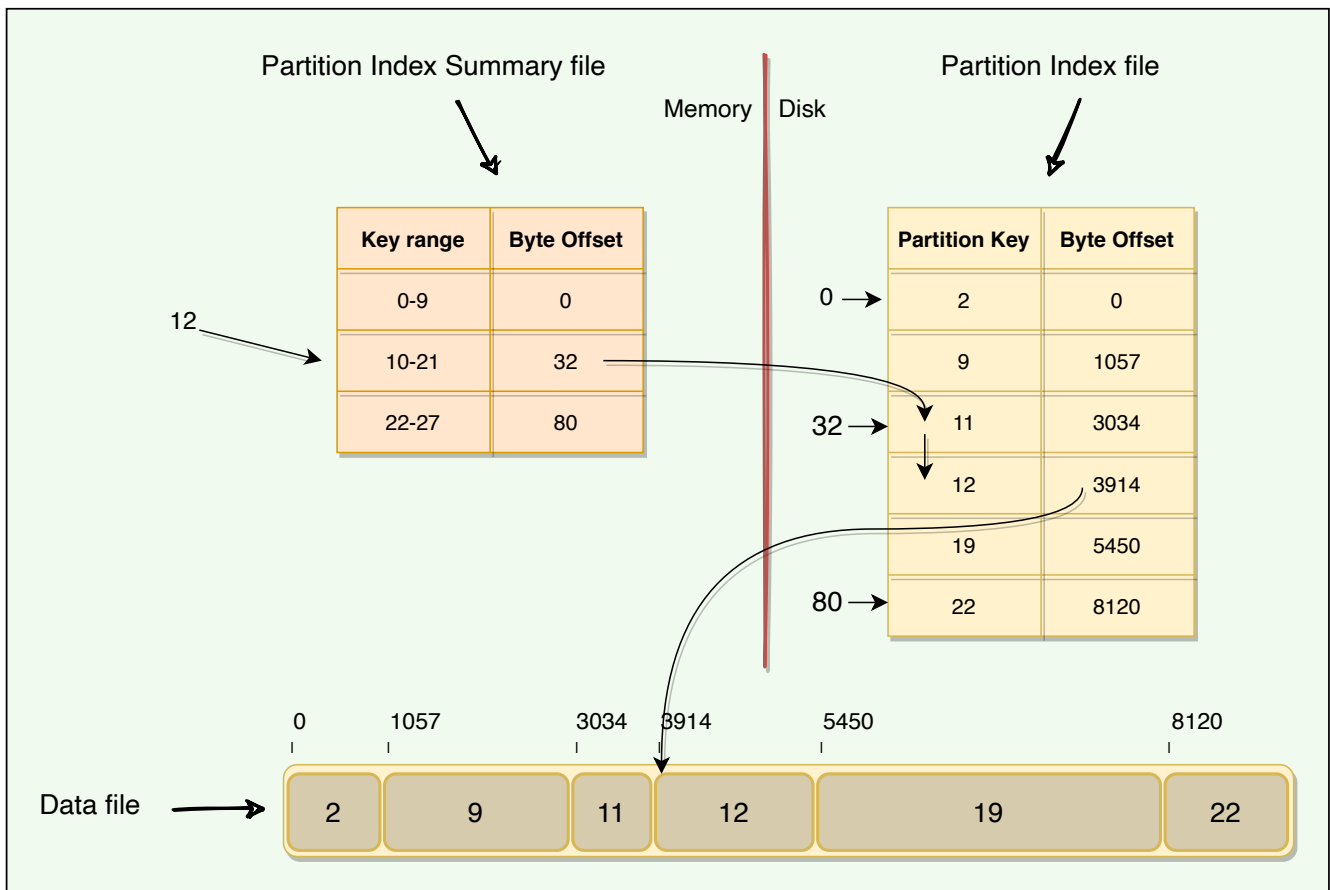
Stored in memory, the Partition Index Summary file stores the summary of the Partition Index file. This is done for performance improvement.



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):

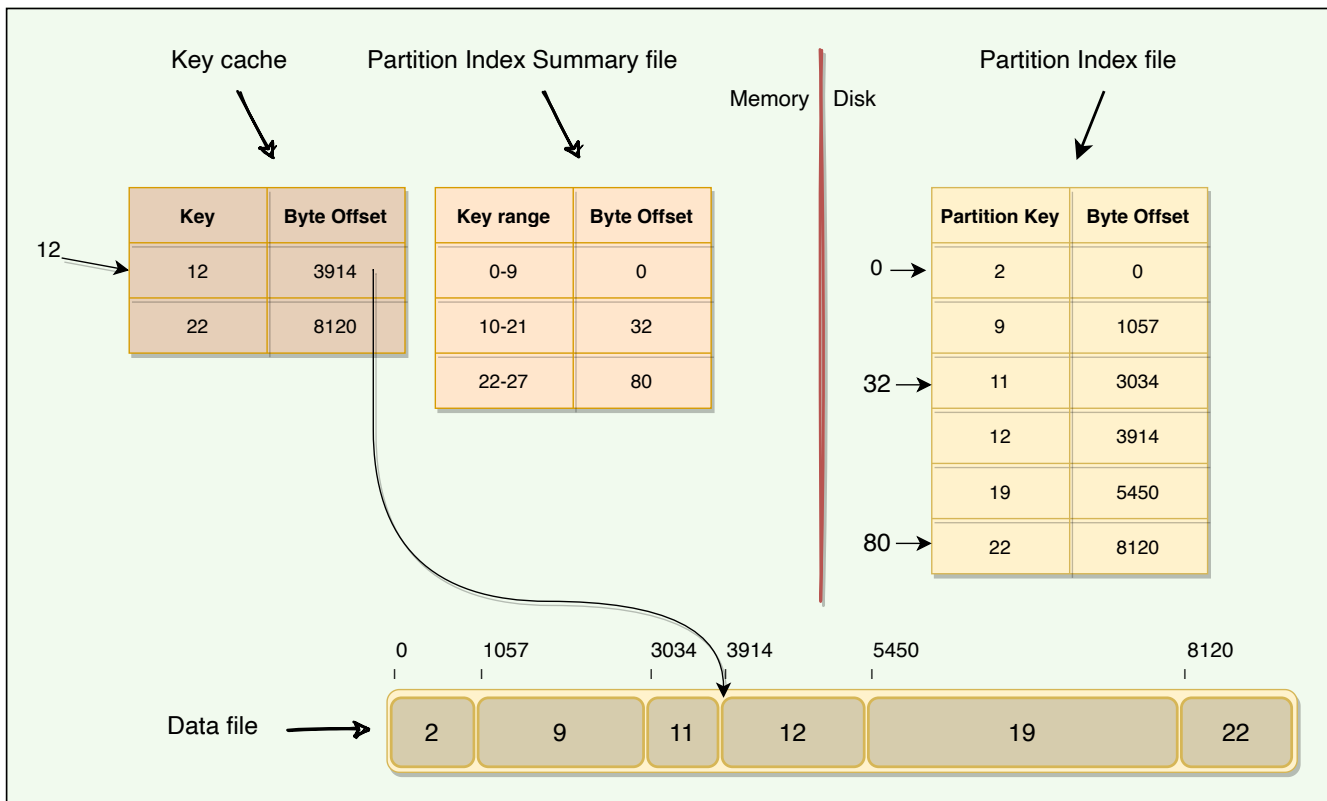
1. In the Partition Index Summary file, find the key range in which the key=12 lies. This will give us offset (=32) into the Partition Index file.
2. Jump to offset 32 in the Partition Index file to search for the offset of key=12. This will give us offset (=3914) into the SSTable file.
3. Jump to SSTable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



Reading from partition index summary file

Reading SSTable through key cache

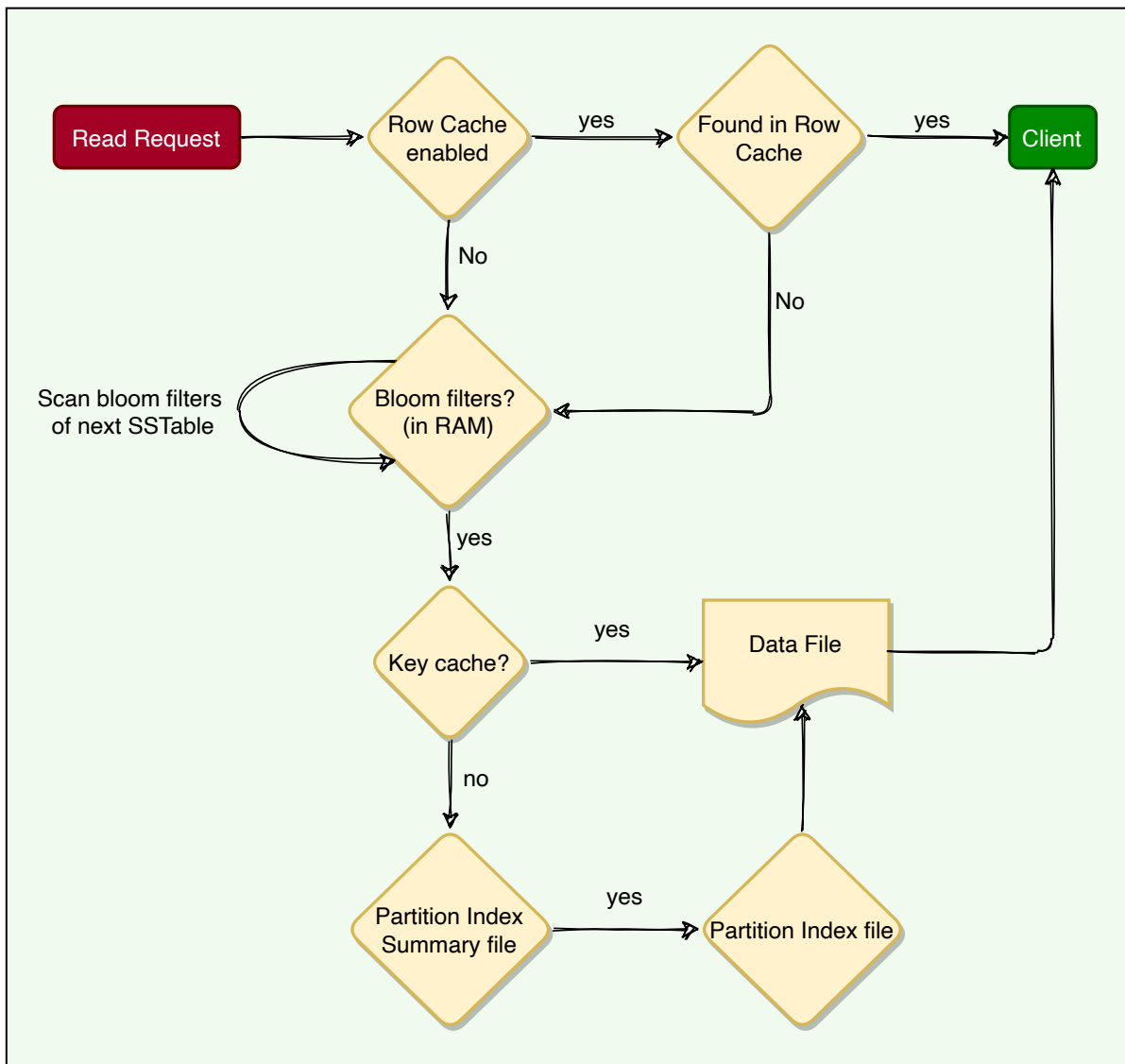
As the Key Cache stores a map of recently read partition keys to their SSTable offsets, it is the fastest way to find the required row in the SSTable.



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

Next →

Anatomy of Cassandra's Write Operat...

Compaction

Compaction

Let's explore how Cassandra handles compaction.

We'll cover the following

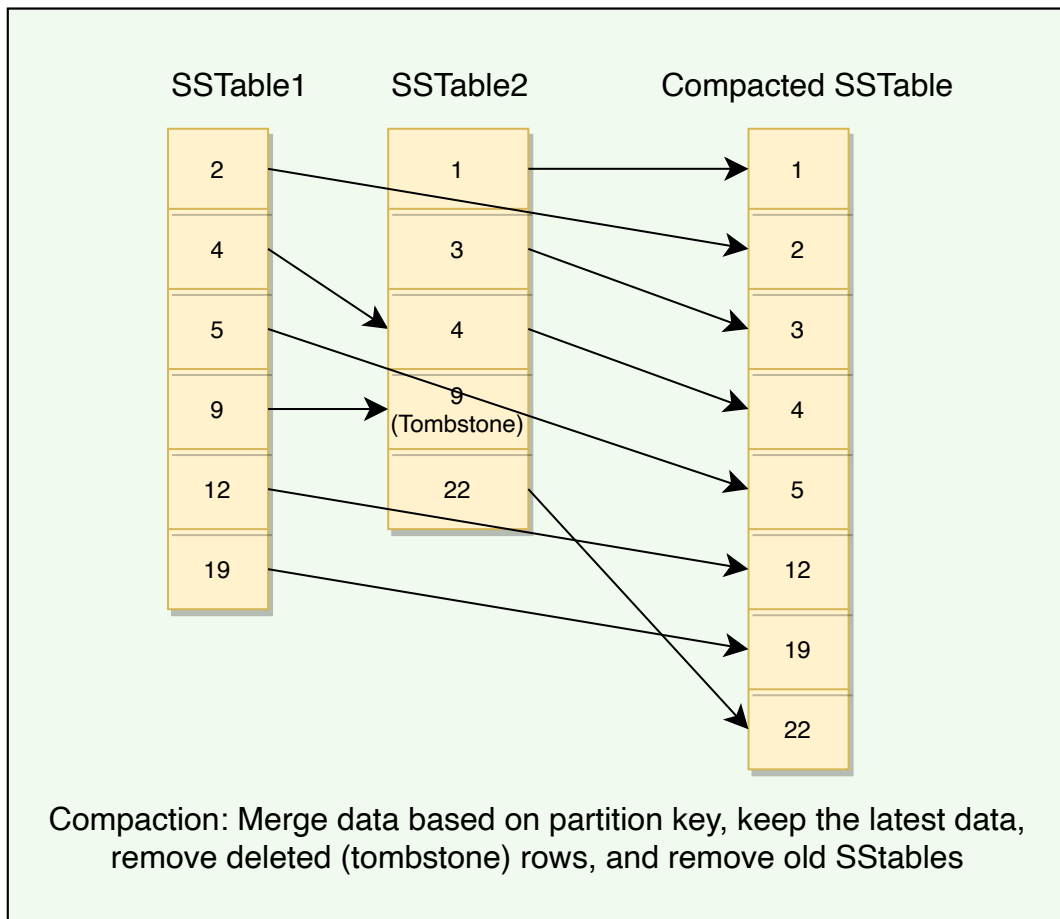


- How does compaction work in Cassandra?
- Compaction strategies
- Sequential writes

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](#)[Next >](#)

Anatomy of Cassandra's Read Operati...

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](#)[Next >](#)

Compaction

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

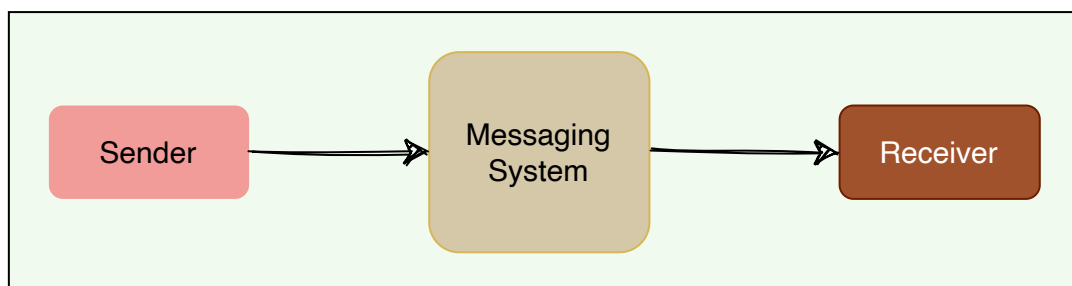
One of the common challenges among distributed systems is handling a continuous influx of data from multiple sources. Imagine a log aggregation service that is receiving hundreds of log entries per second from different sources. The function of this log aggregation service is to store these logs on disk 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?

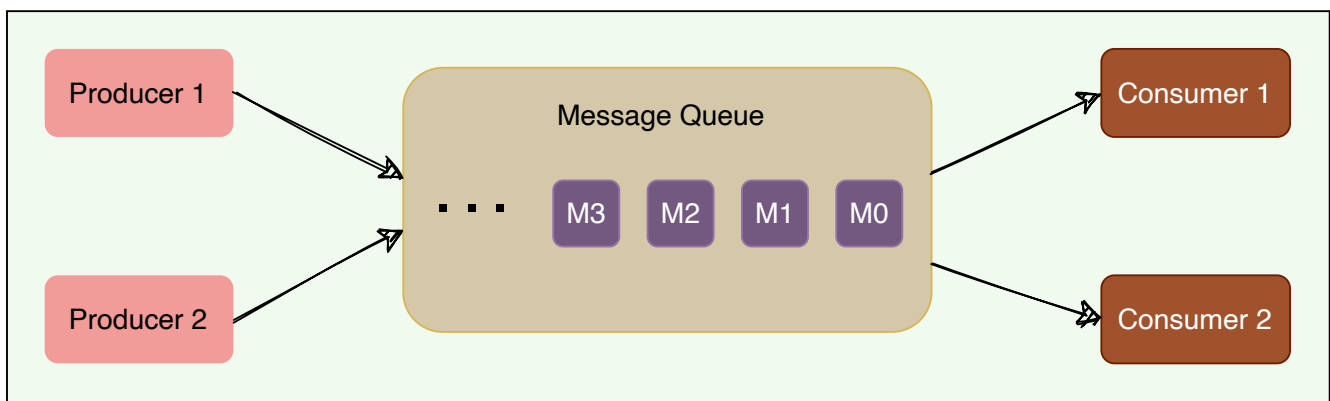
A **messaging system** is responsible for transferring data among services, applications, processes, or servers. Such a system helps **decouple** different parts of a distributed system by providing an **asynchronous** way of transferring messaging between the sender and the receiver. Hence, all senders (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**.

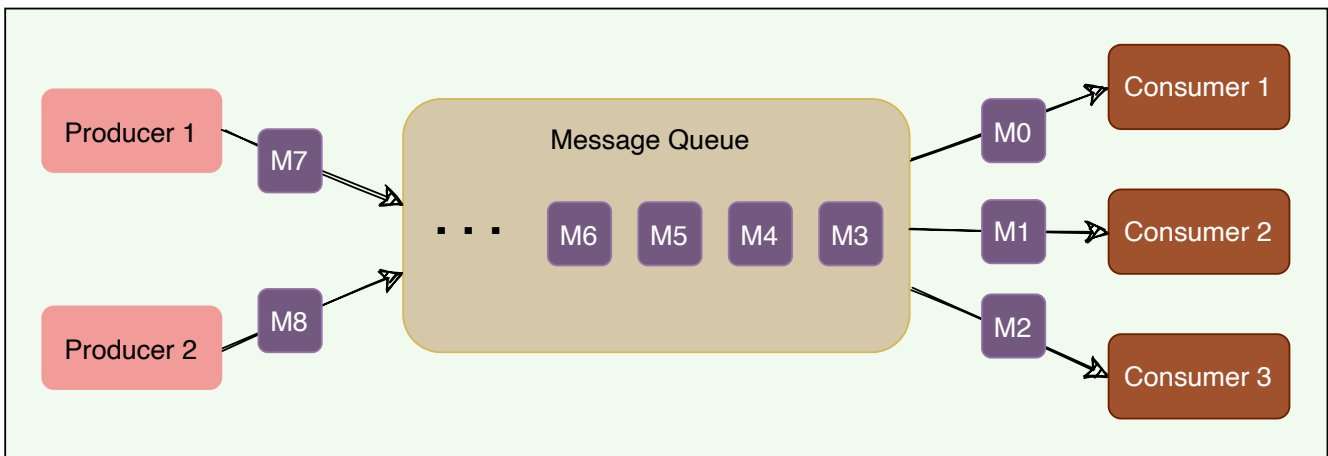
Queue

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.



Message queue

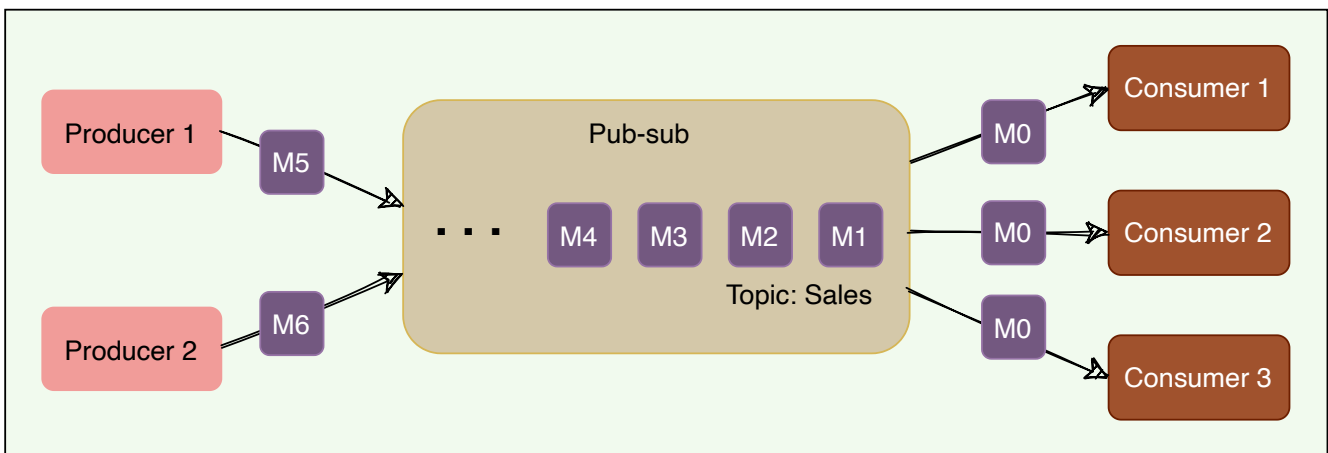
A particular message can be consumed by a maximum of one consumer only. Once a consumer grabs a message, it is removed from the queue such that the next consumer will get the next message. This is a great model for distributing message-processing among multiple consumers. But this also limits the system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



Message consumption in a message queue

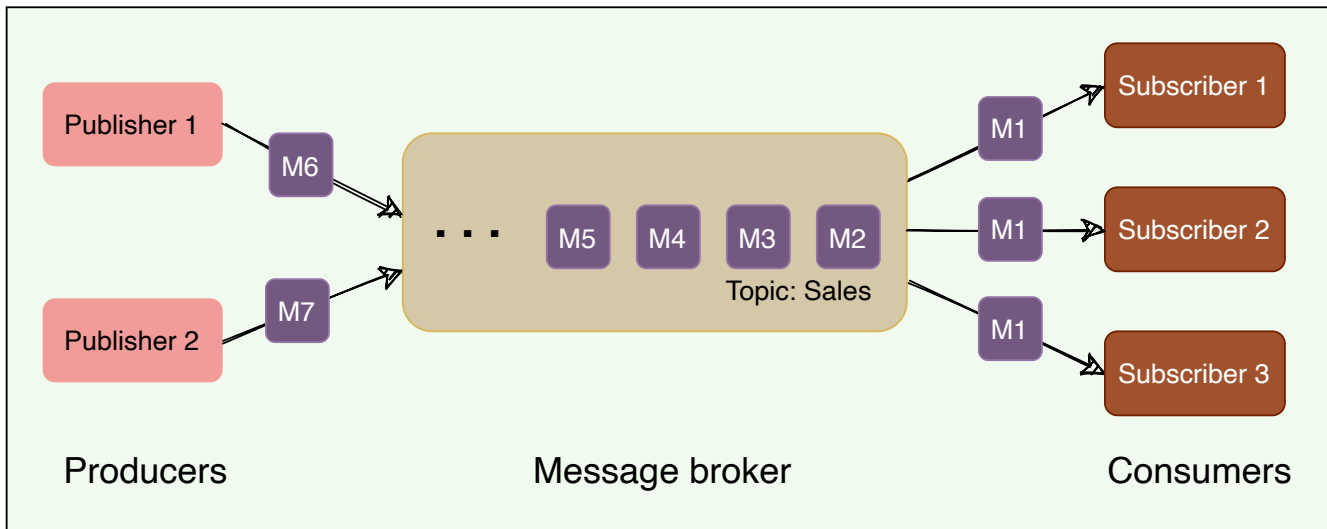
Publish-subscribe messaging system

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.



Message broker

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.

← Back

Mock Interview: Cassandra

Next →

Kafka: Introduction

Kafka: Introduction

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

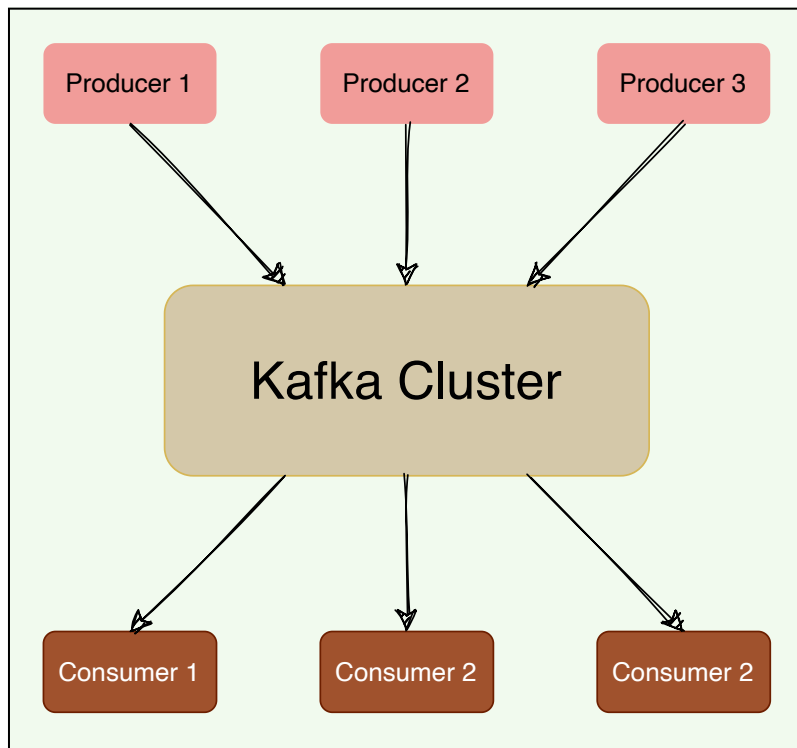
We'll cover the following



- 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

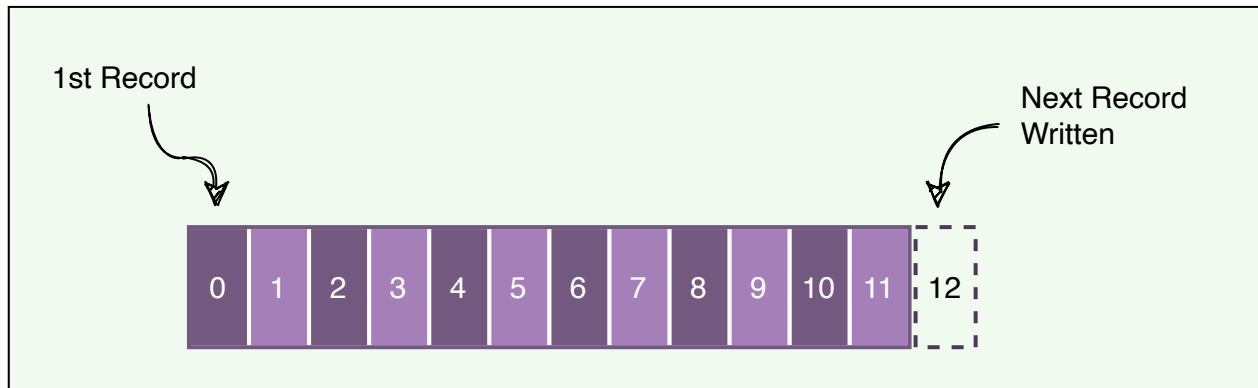
Background

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/>), 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.

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

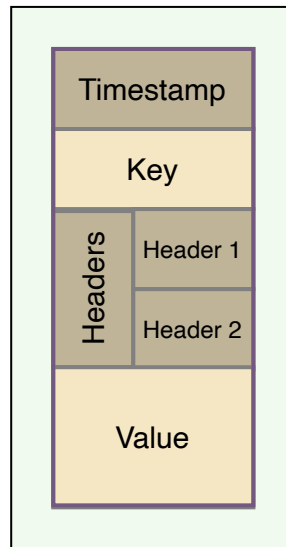
Before digging deep into Kafka's architecture, let's first go through some of its common terms.

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

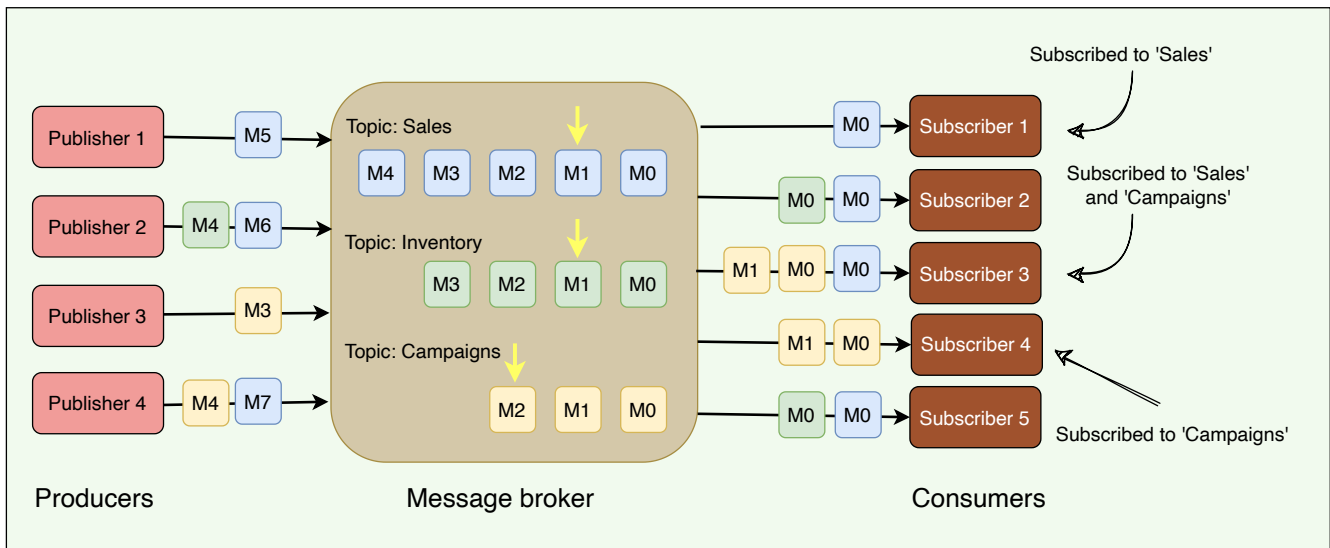
Topics

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.



Kafka topics

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

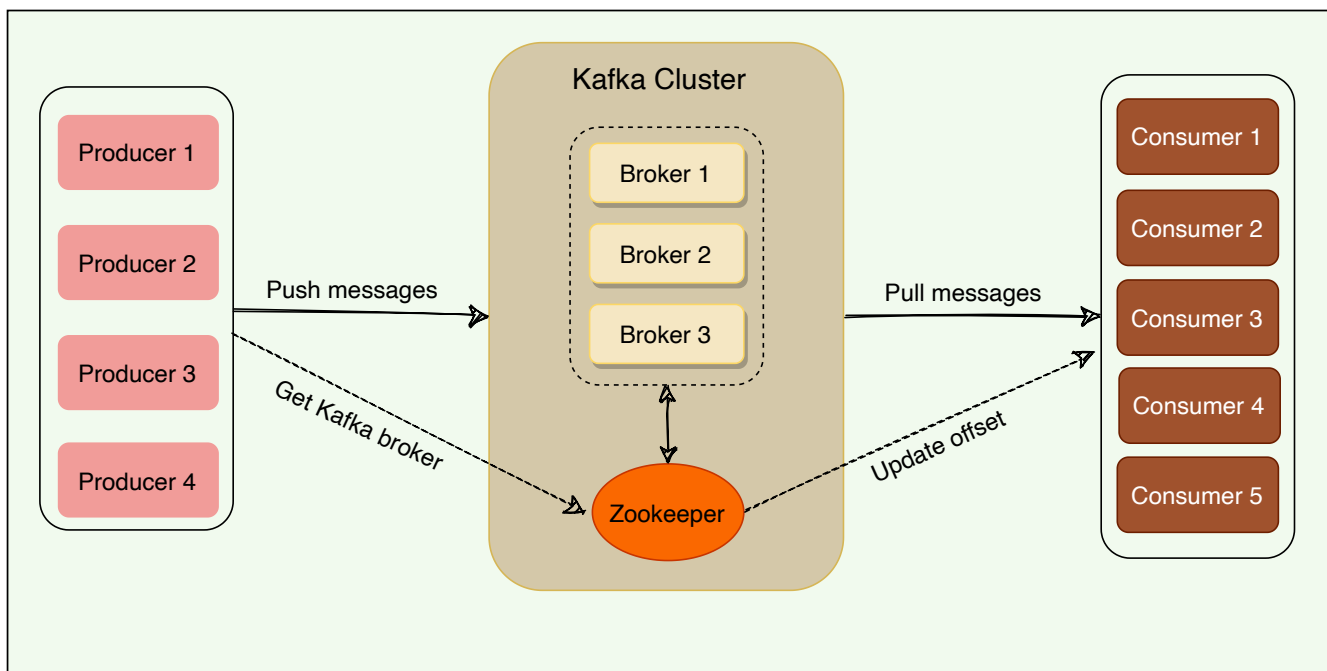
At a high level, applications (producers) send messages to a Kafka broker, and these messages are read by other applications called consumers. Messages get stored in a topic, and consumers subscribe to the topic to receive new messages.

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

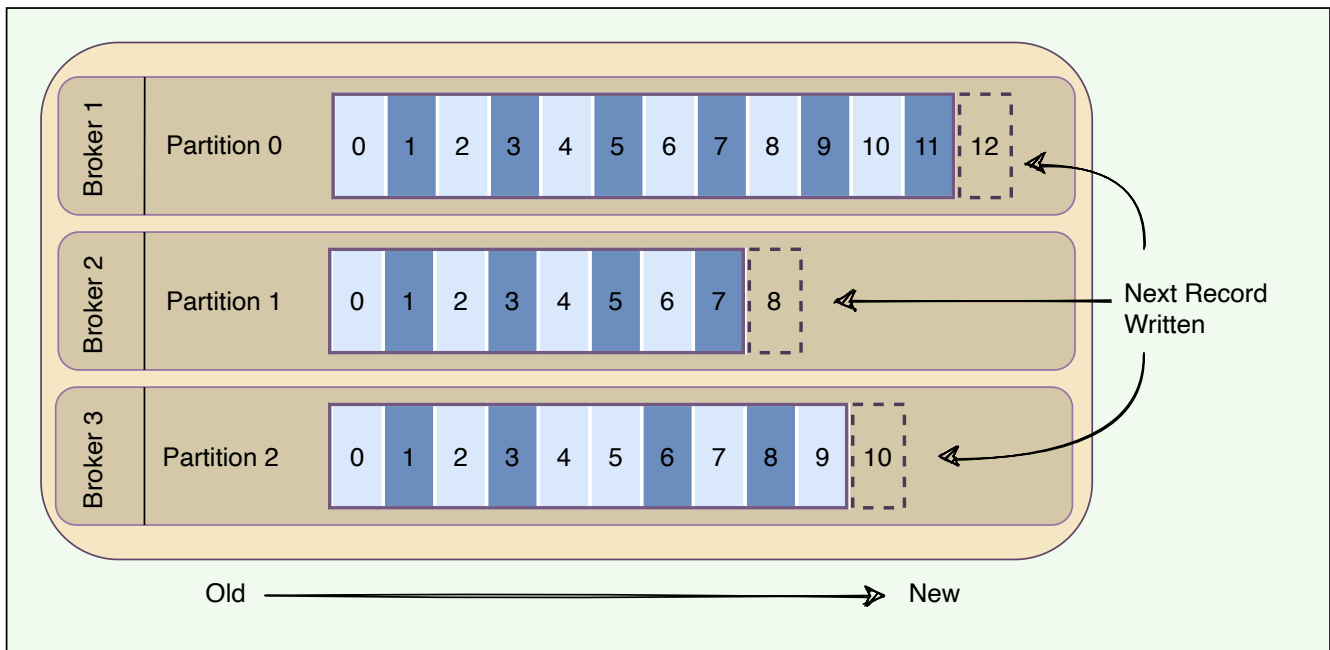
Kafka is simply a **collection of topics**. As topics can get quite big, they are **split into partitions** of a smaller size for better performance and scalability.

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.

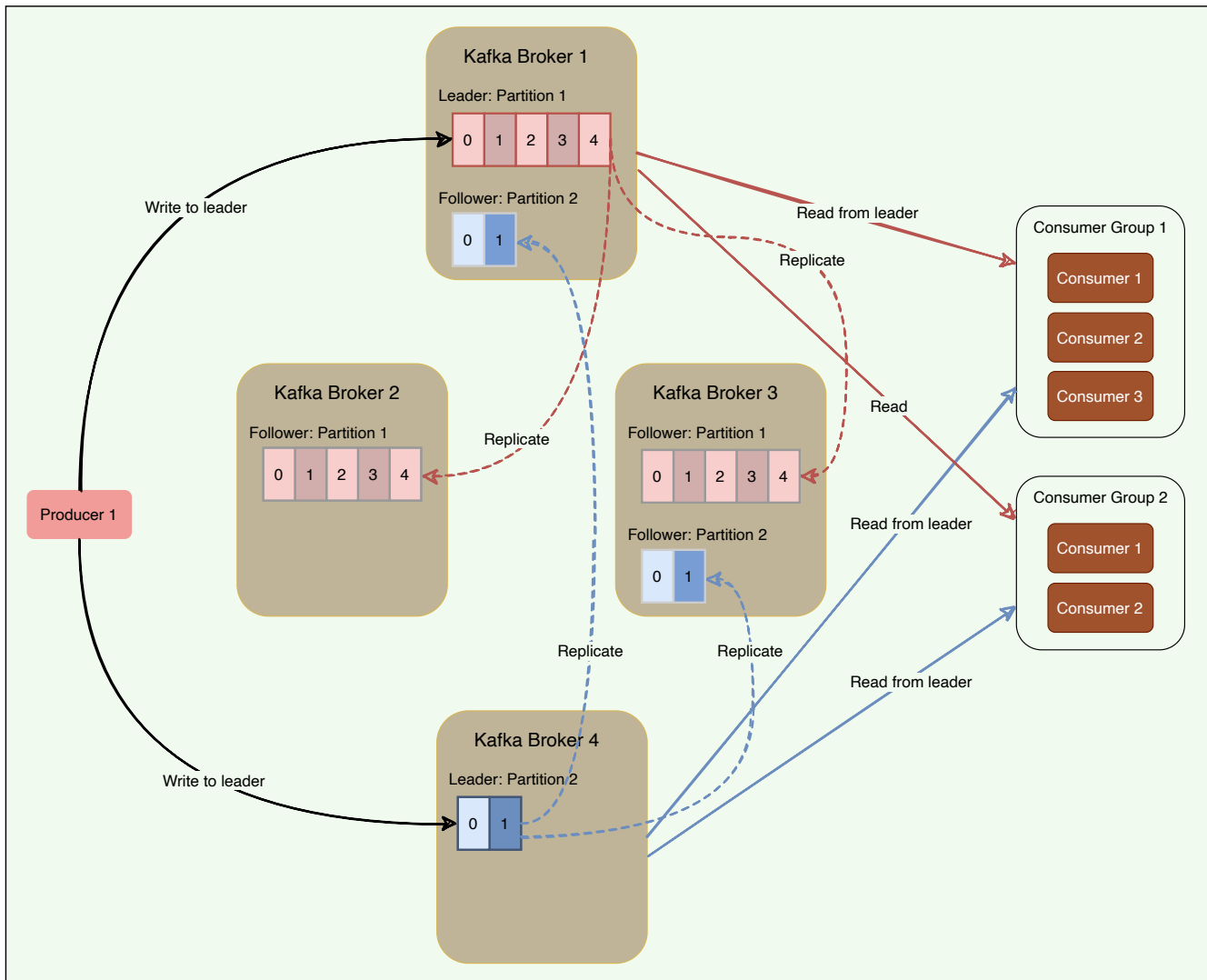
Leader

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

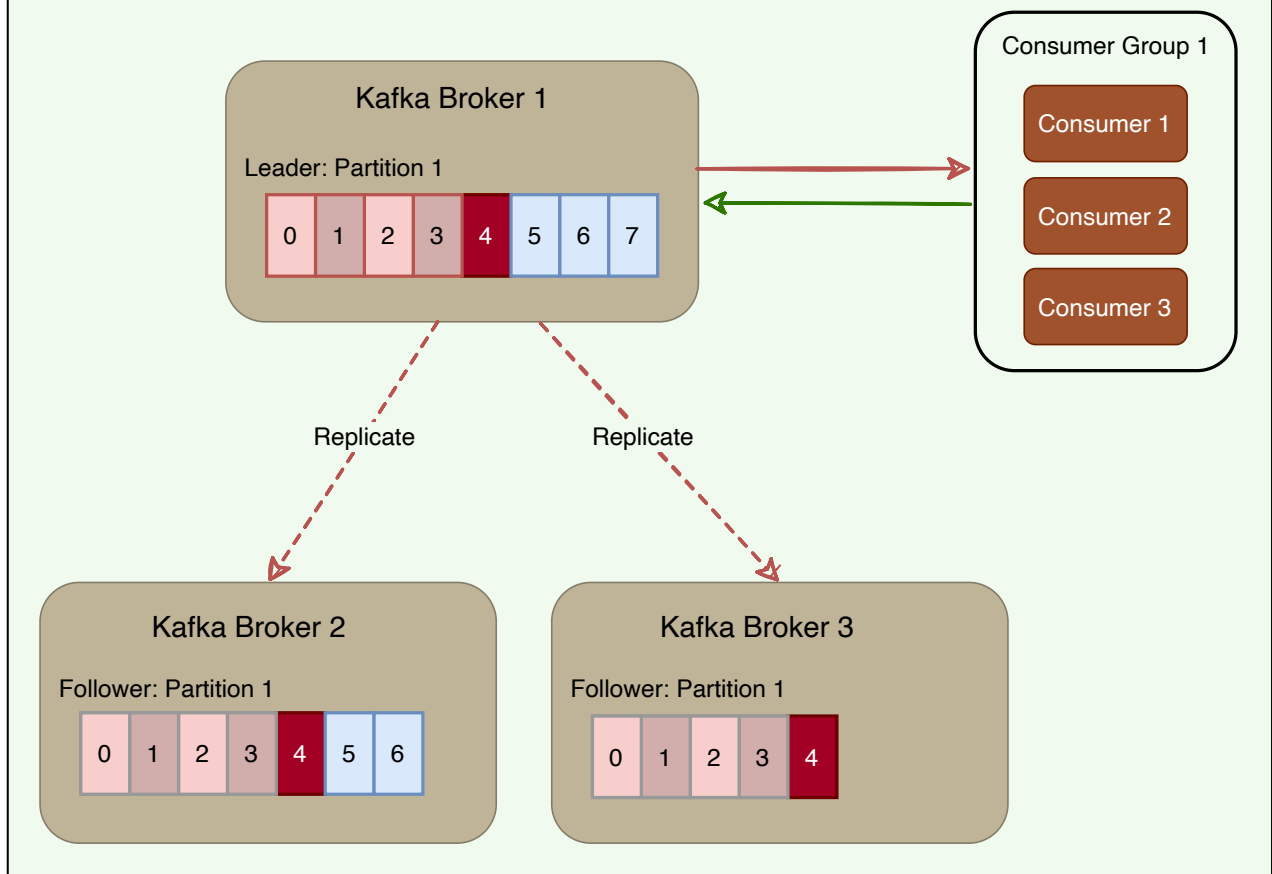
An in-sync replica (ISR) is a broker that has the latest data for a given partition. A leader is always an in-sync replica. A follower is an in-sync replica only if it has fully caught up to the partition it is following. In other words, ISRs cannot be behind on the latest records for a given partition.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



High-water mark offset

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

High-level Architecture

Next →

Consumer Groups

Consumer Groups

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

We'll cover the following



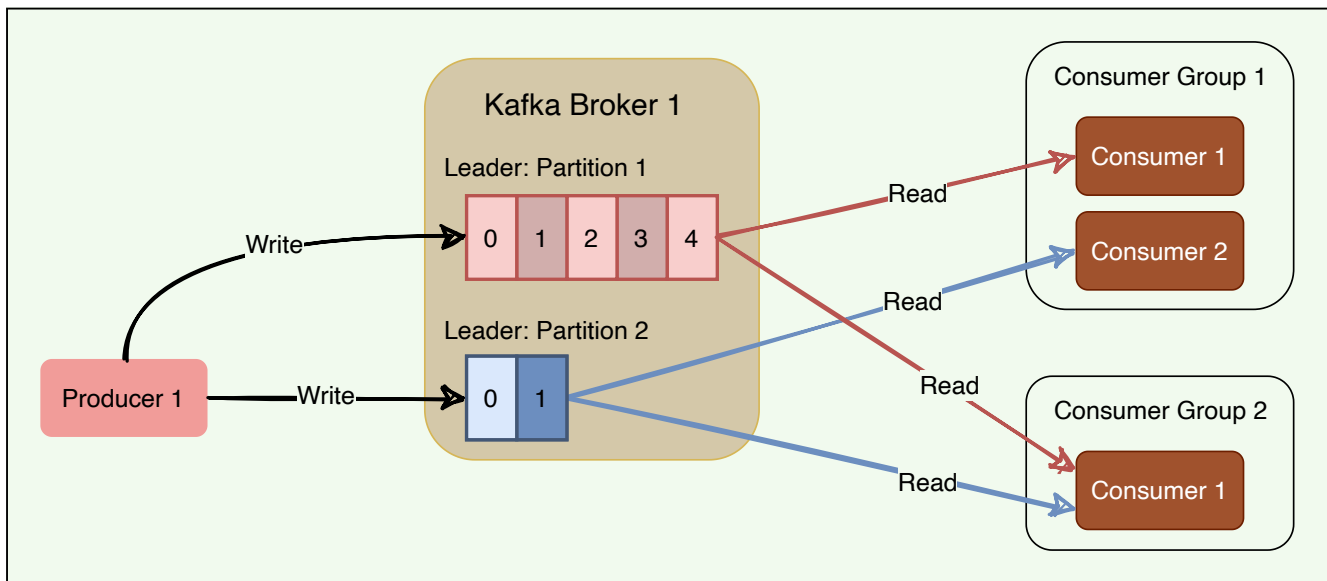
- What is a consumer group?
 - Distributing partitions to consumers within a consumer group

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.

← Back

Kafka: Deep Dive

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.

← Back

Consumer Groups

Next →

Role of ZooKeeper

Role of ZooKeeper

Let's delve into how Kafka interacts with ZooKeeper.

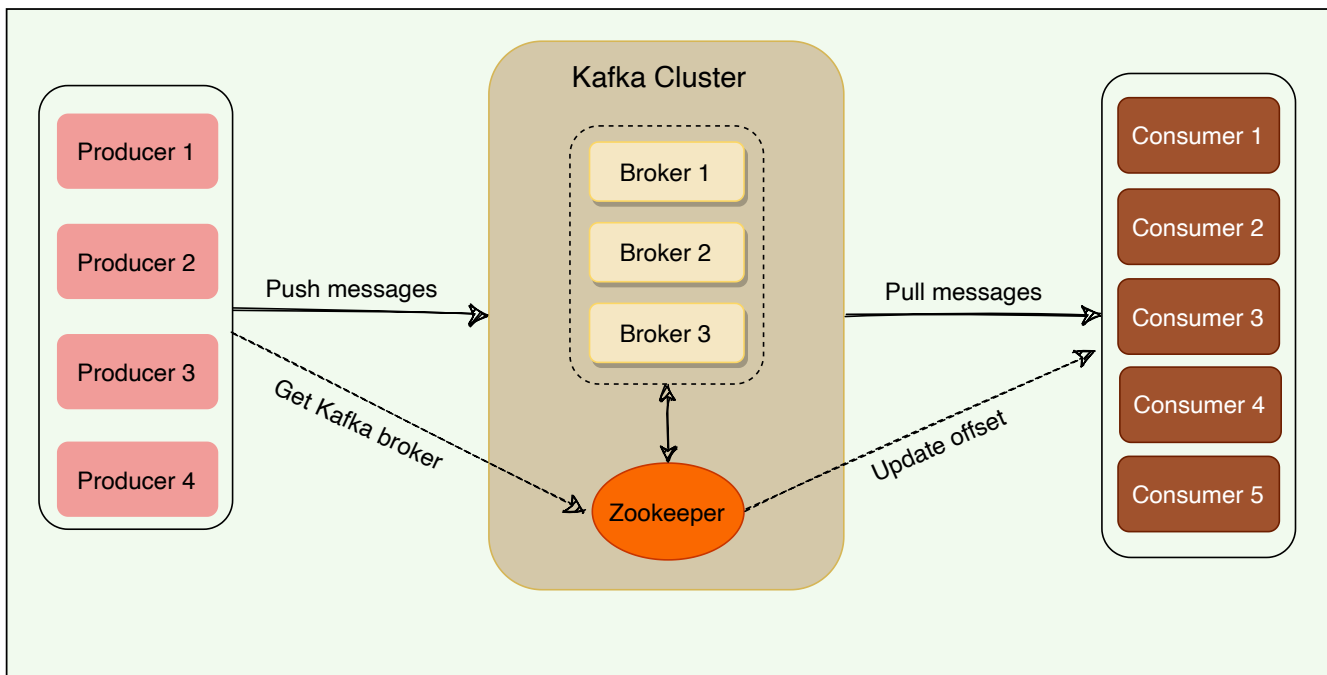
We'll cover the following



- 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.

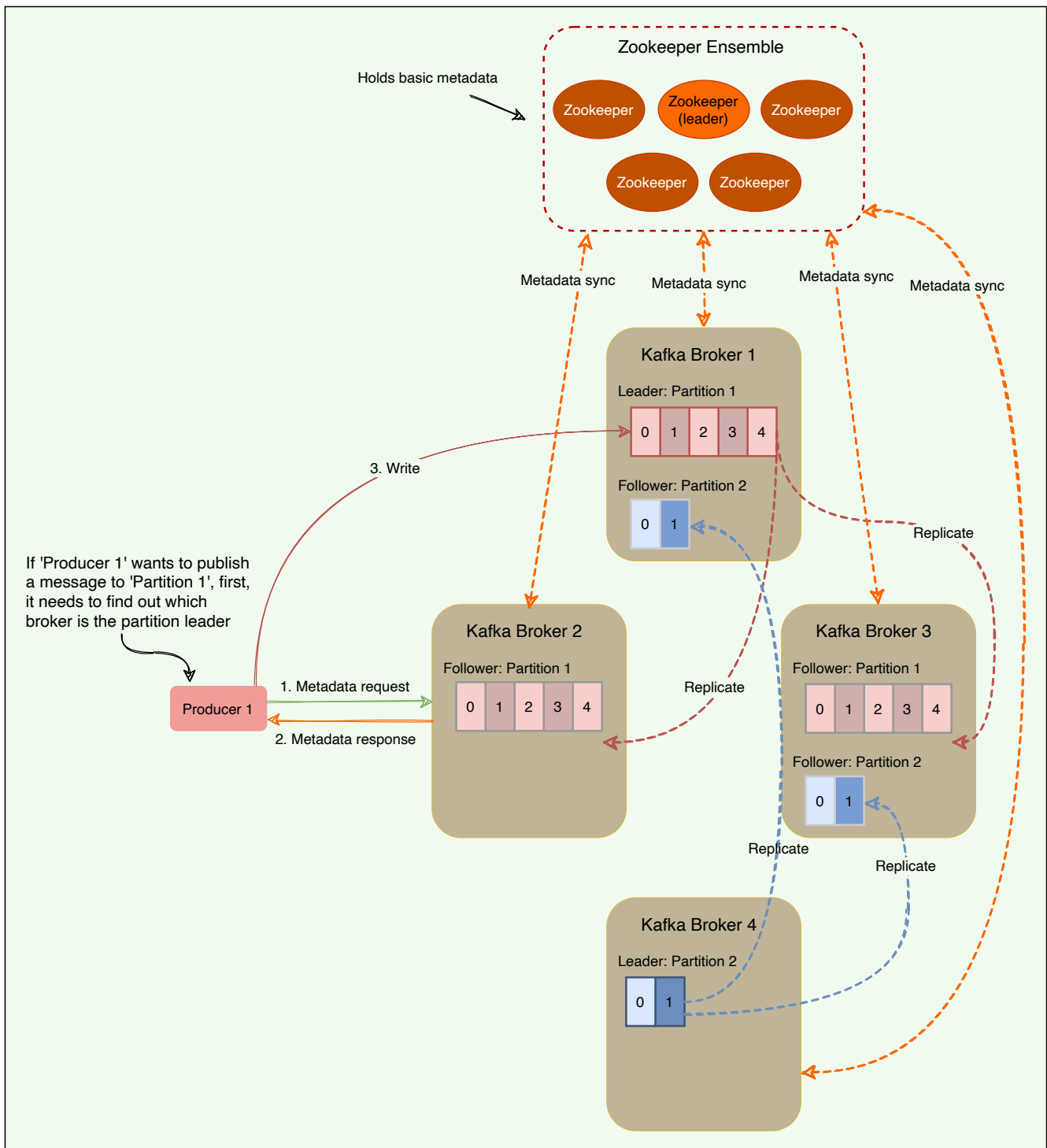
ZooKeeper is used for storing all sorts of metadata about the Kafka cluster:

- 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](#)[Next >](#)

Kafka Delivery Semantics

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:

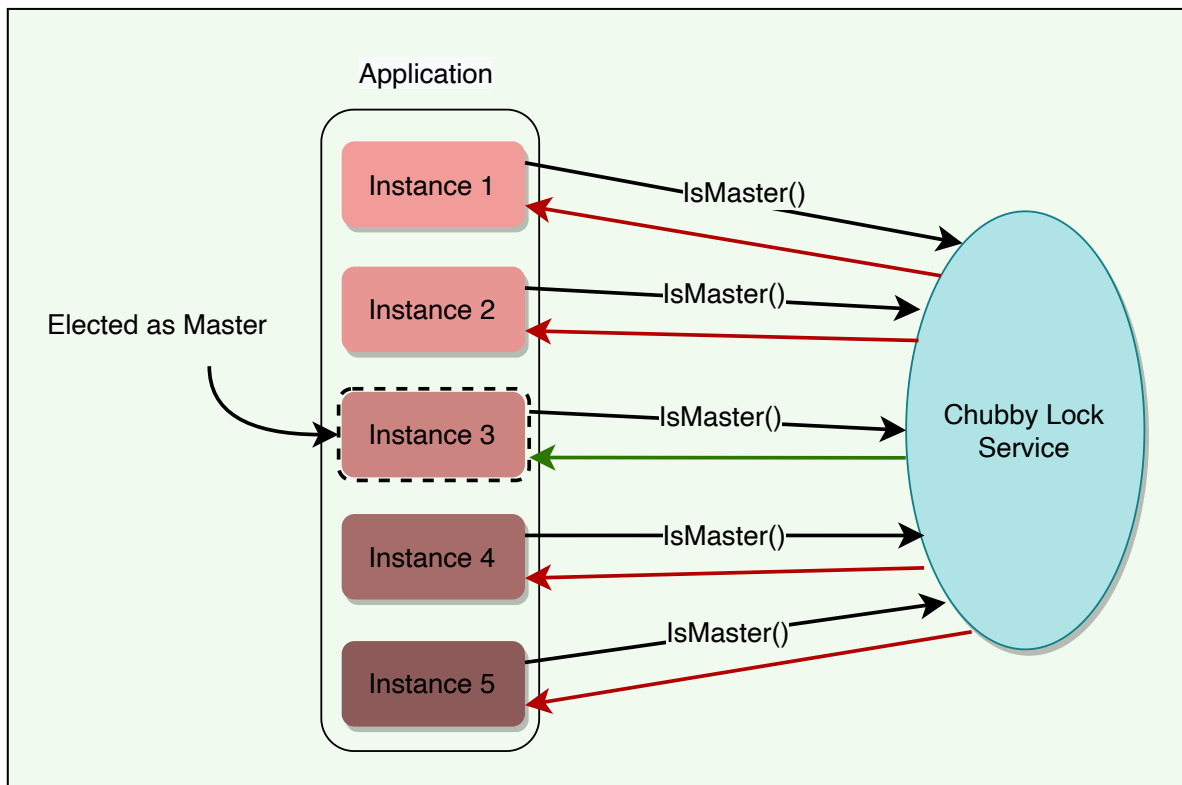
- Leader/master election
- Naming service (like DNS)
- Storage (small objects that rarely change)
- Distributed locking mechanism

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

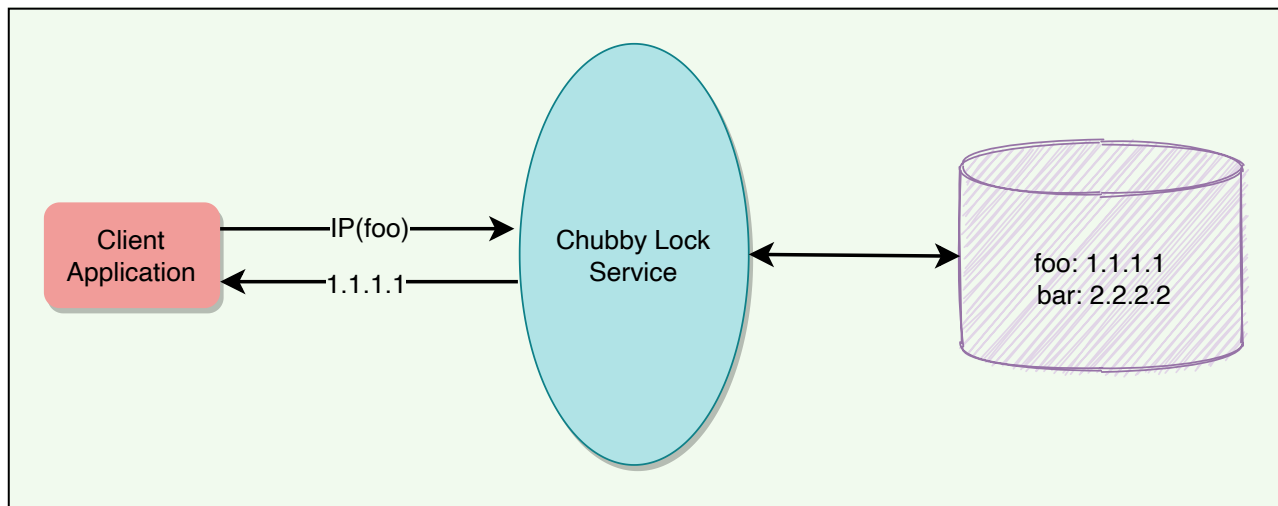
running and wants one of them to be chosen as the leader. Chubby can be used for leader election among a set of replicas, e.g., the leader/master for GFS and BigTable.



Chubby being used for leader election

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.

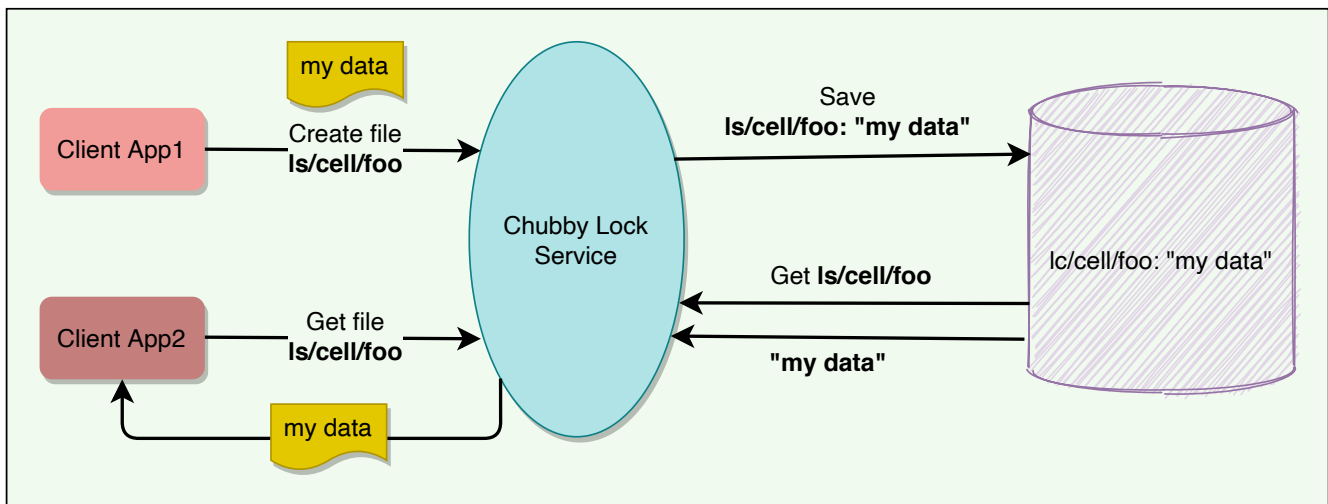


Chubby as DNS

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).

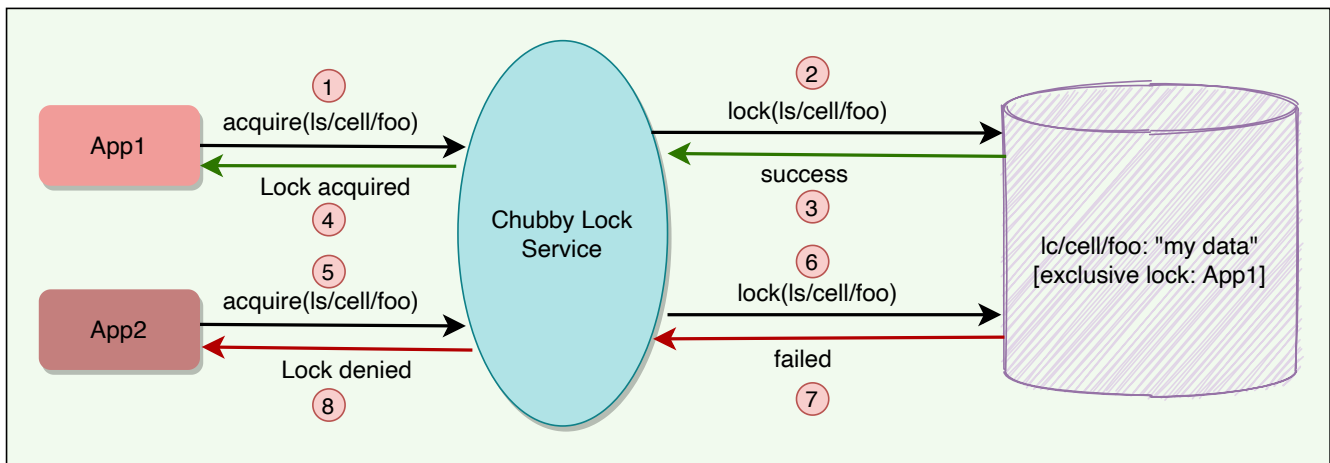
Applications can then use these files for any usage like DNS, configs, etc. GFS and Bigtable store their metadata in Chubby. Some services use Chubby to store ACL files.



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:

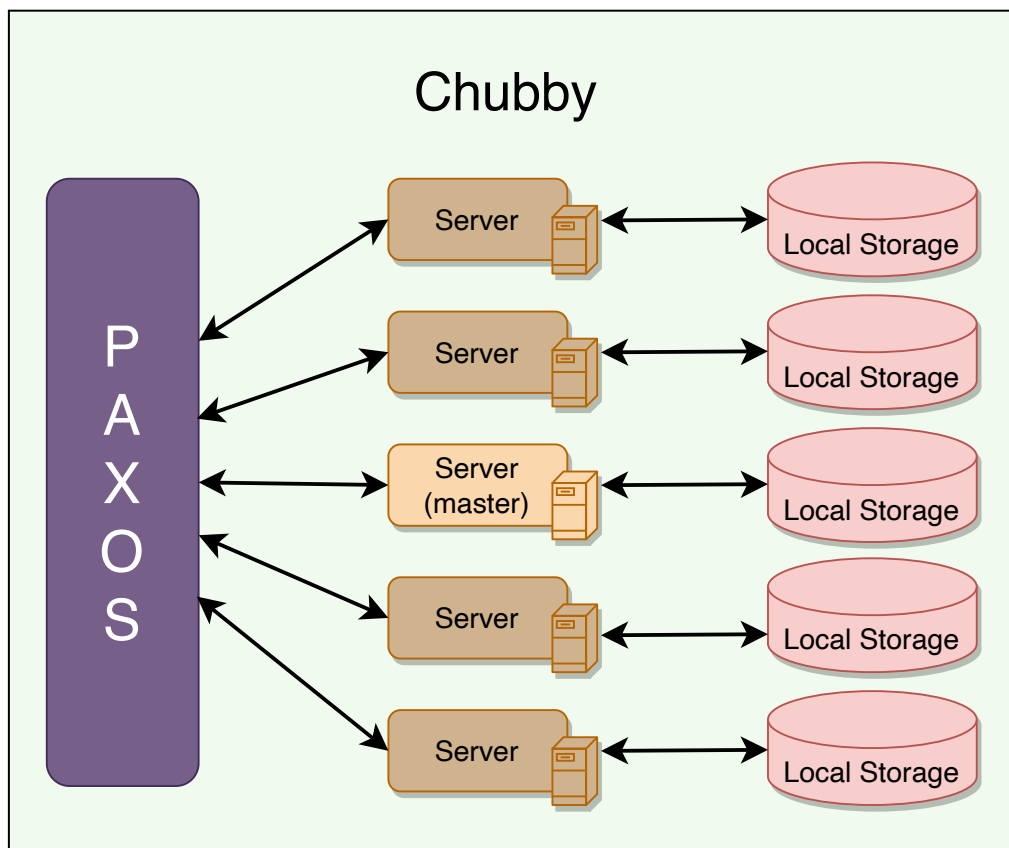
- Bulk storage is needed.
- Data update rate is high.
- Locks are acquired/released frequently.
- Usage is more like a publish/subscribe model.

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 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

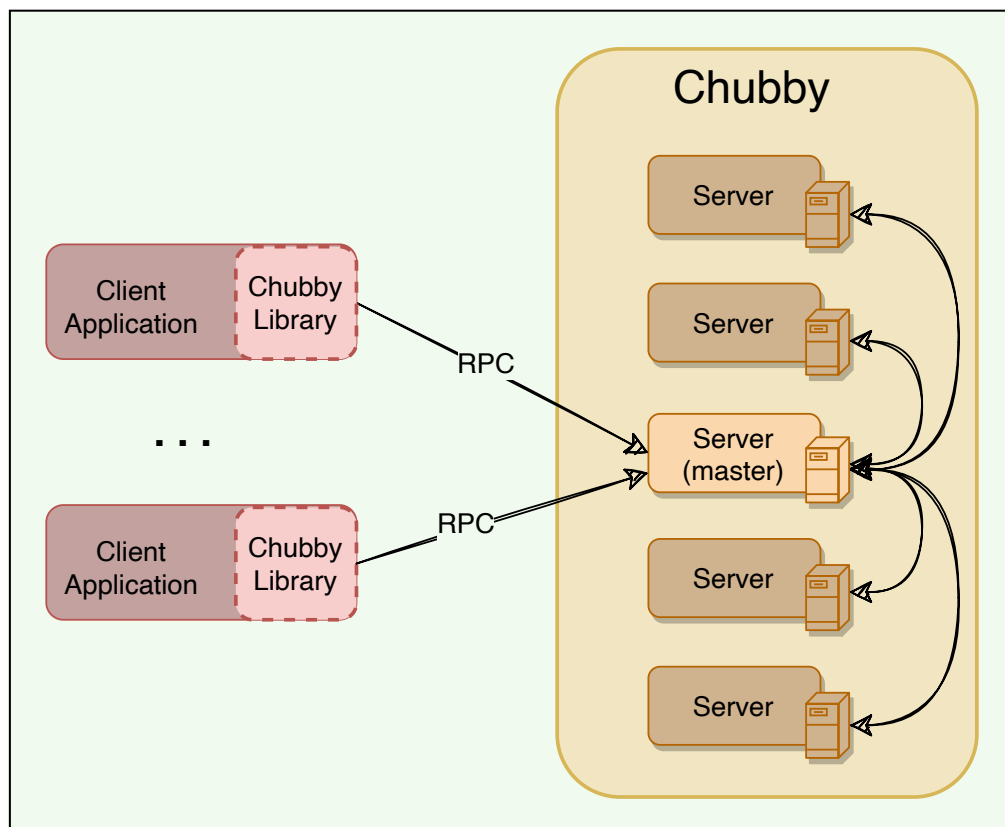
A Chubby Cell basically refers to a Chubby cluster. Most Chubby cells are confined to a single data center or machine room, though there can be a Chubby cell whose replicas are separated by thousands of kilometers. A single Chubby cell has two main components, server and client, that communicate via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

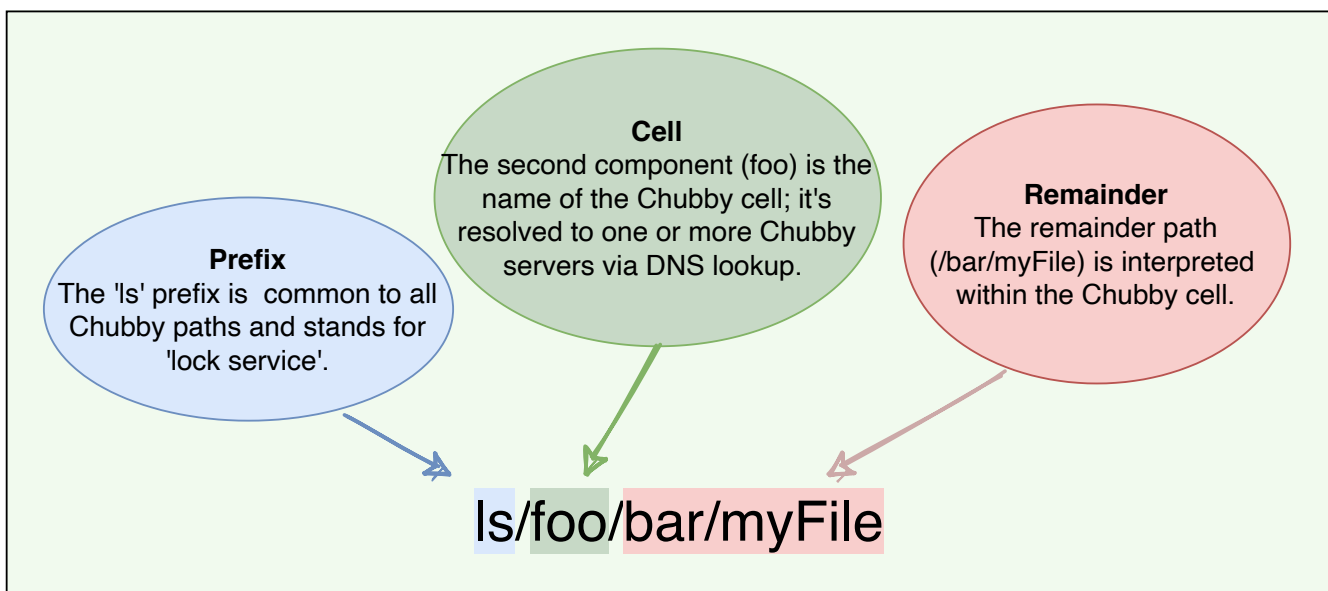
Not having a separate service for sharing the results reduces the number of servers that clients depend on. Chubby's storage requirements are really simple. i.e., store a small amount of data (KBs) and limited operation support (i.e., create/delete).

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

High-level Architecture

Next →

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:

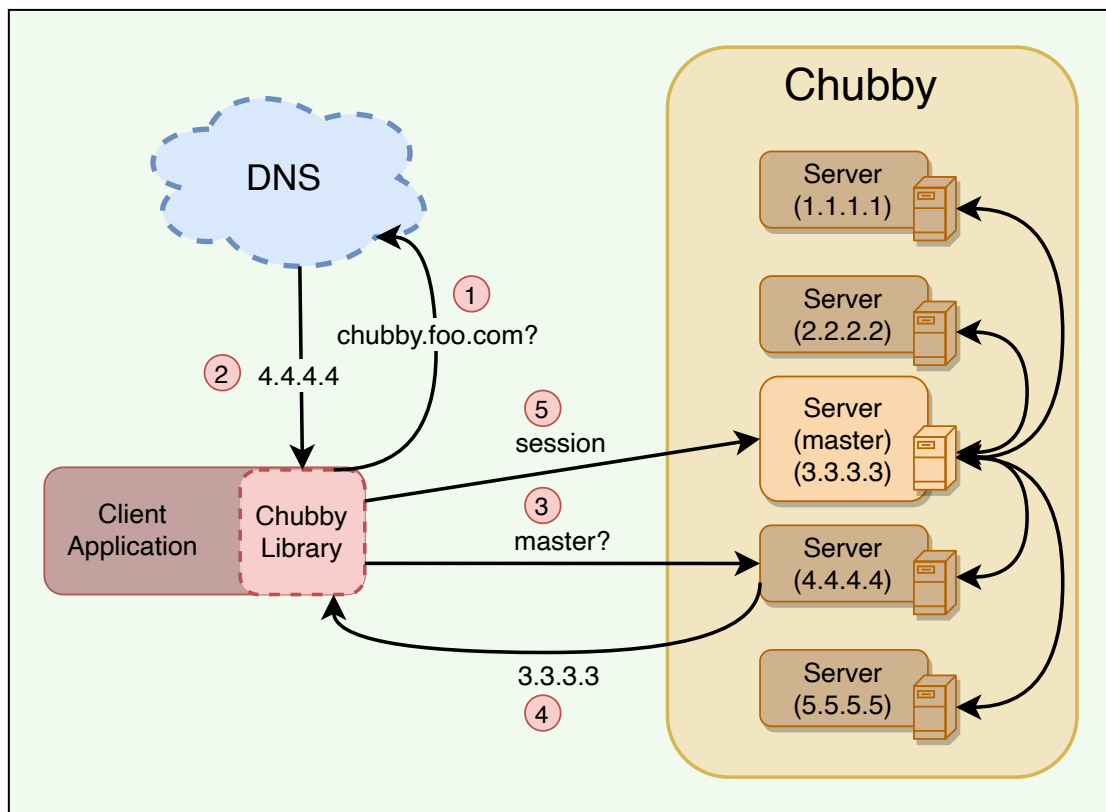
- A master is chosen among Chubby replicas using Paxos.
- Current master information is persisted in storage, and all replicas become aware of the master.

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.



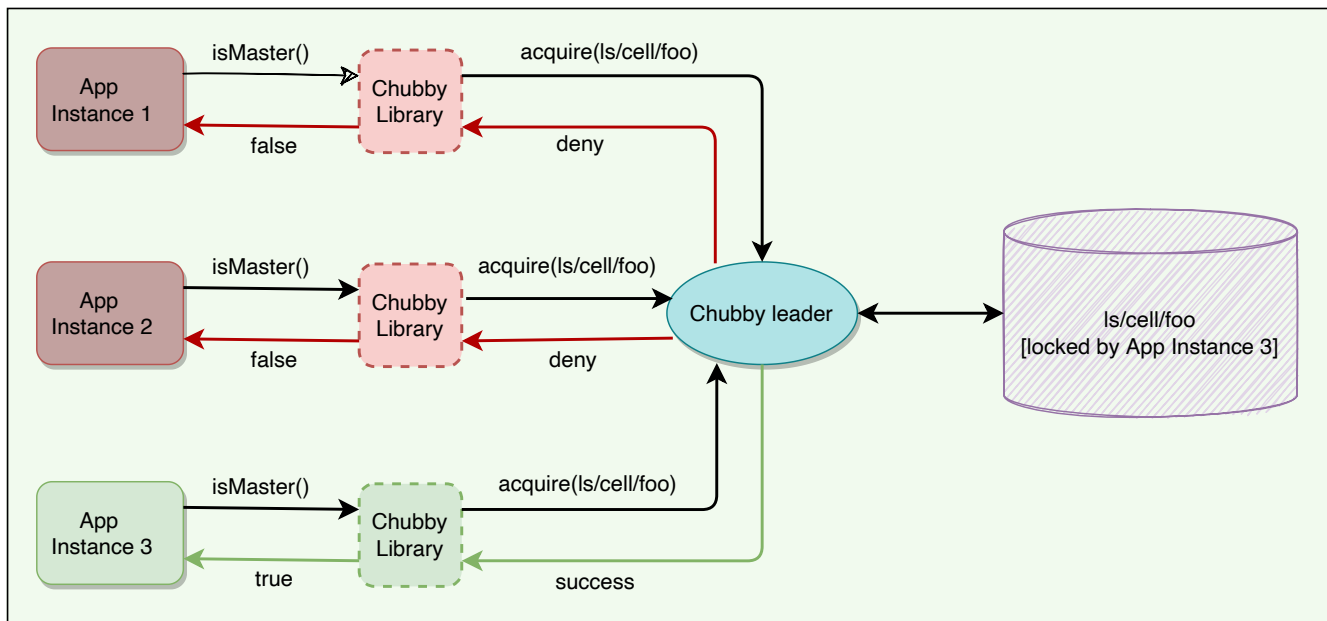
Chubby high-level architecture

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](#)

[Next >](#)

Design Rationale

File, Directories, and Handles

File, Directories, and Handles

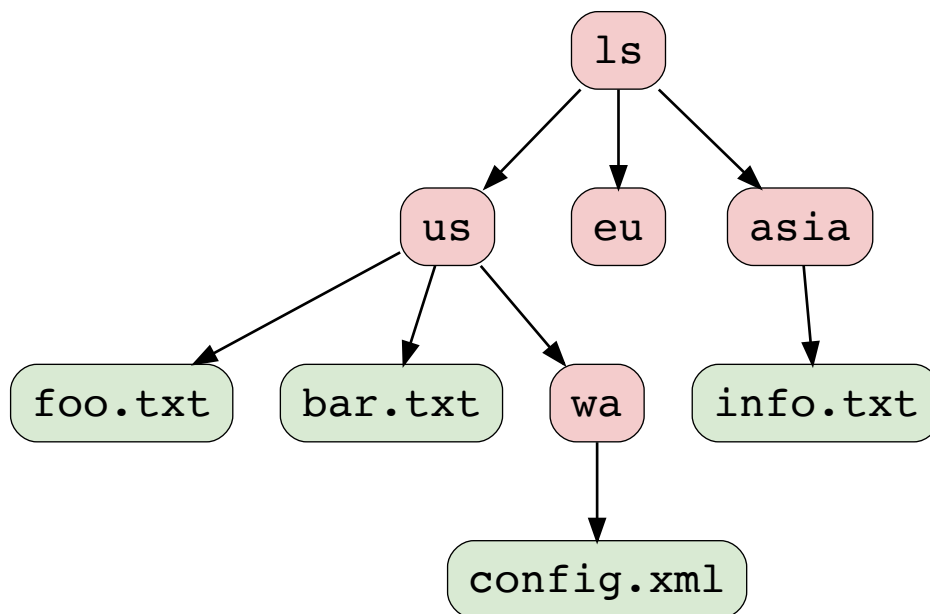
Let's explore how Chubby works with files, directories, and handles.

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:

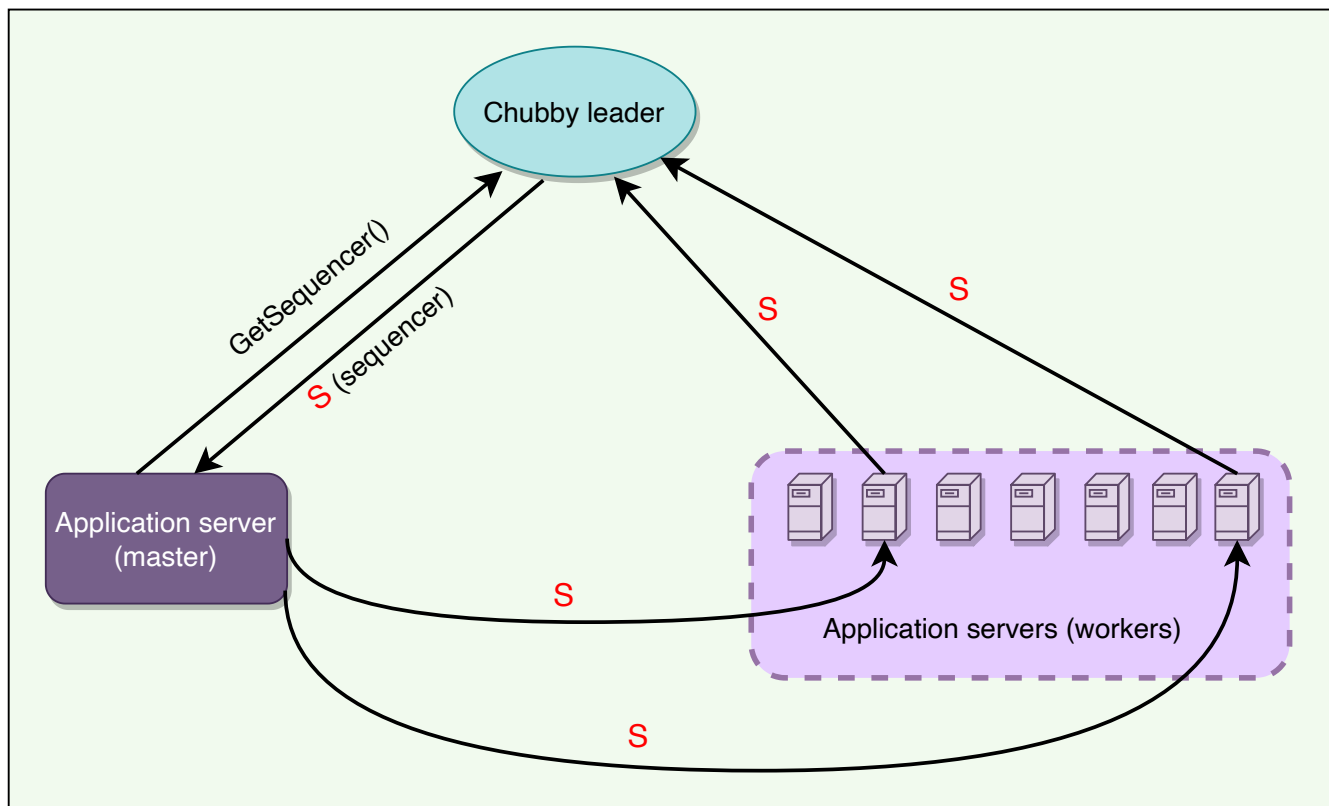
- **Exclusive:** One client may hold the lock in exclusive (write) mode.
- **Shared:** Any number of clients may hold the lock in shared (reader) mode.

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.

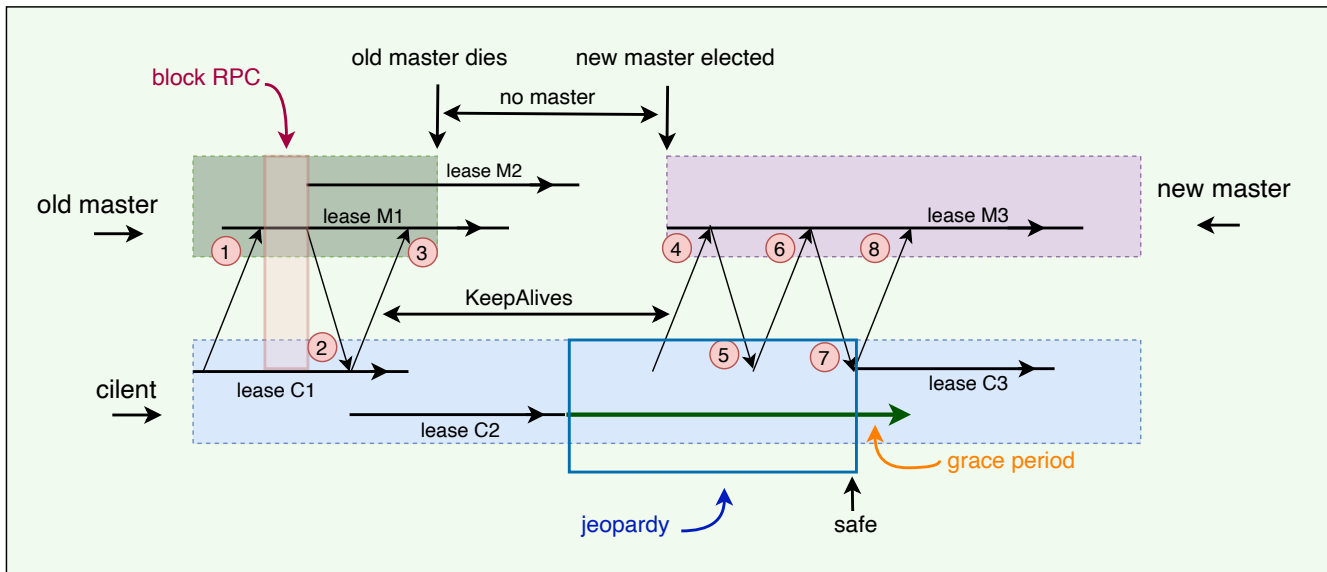
- A session ends if the client ends it explicitly or it has been idle. A session is considered idle if there are no open handles and calls for a minute.
- Each session has an associate lease, which is a time interval during which the master guarantees to not terminate the session unilaterally. The end 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,

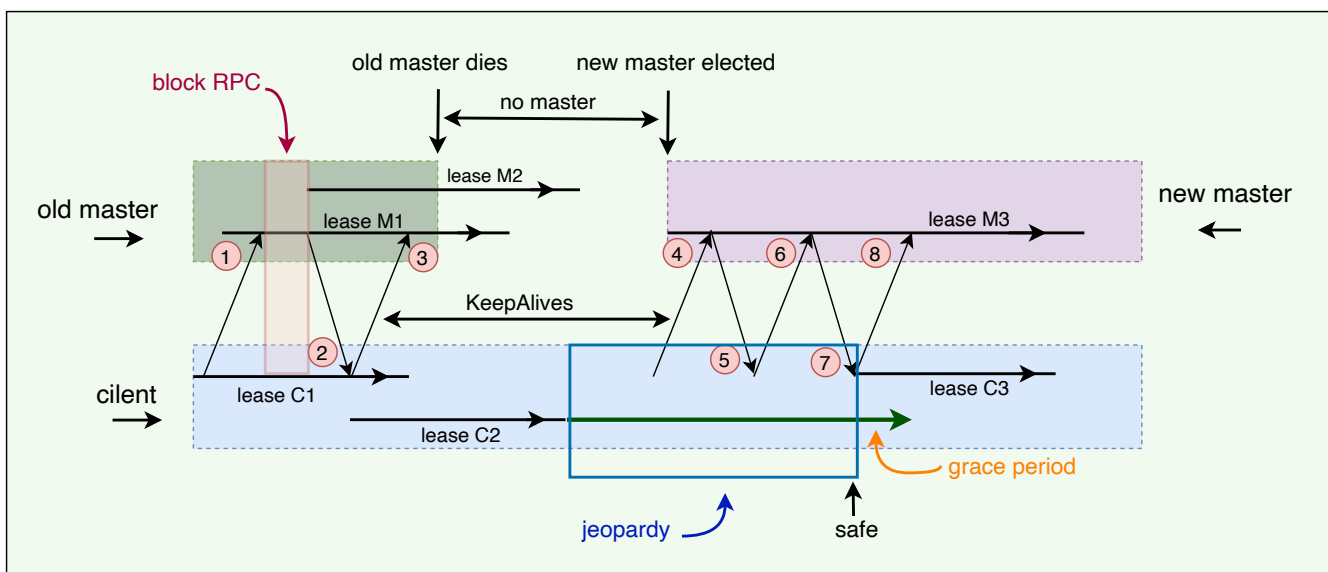
the client enables its cache once more. Otherwise, the client assumes that the session has expired.

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.

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

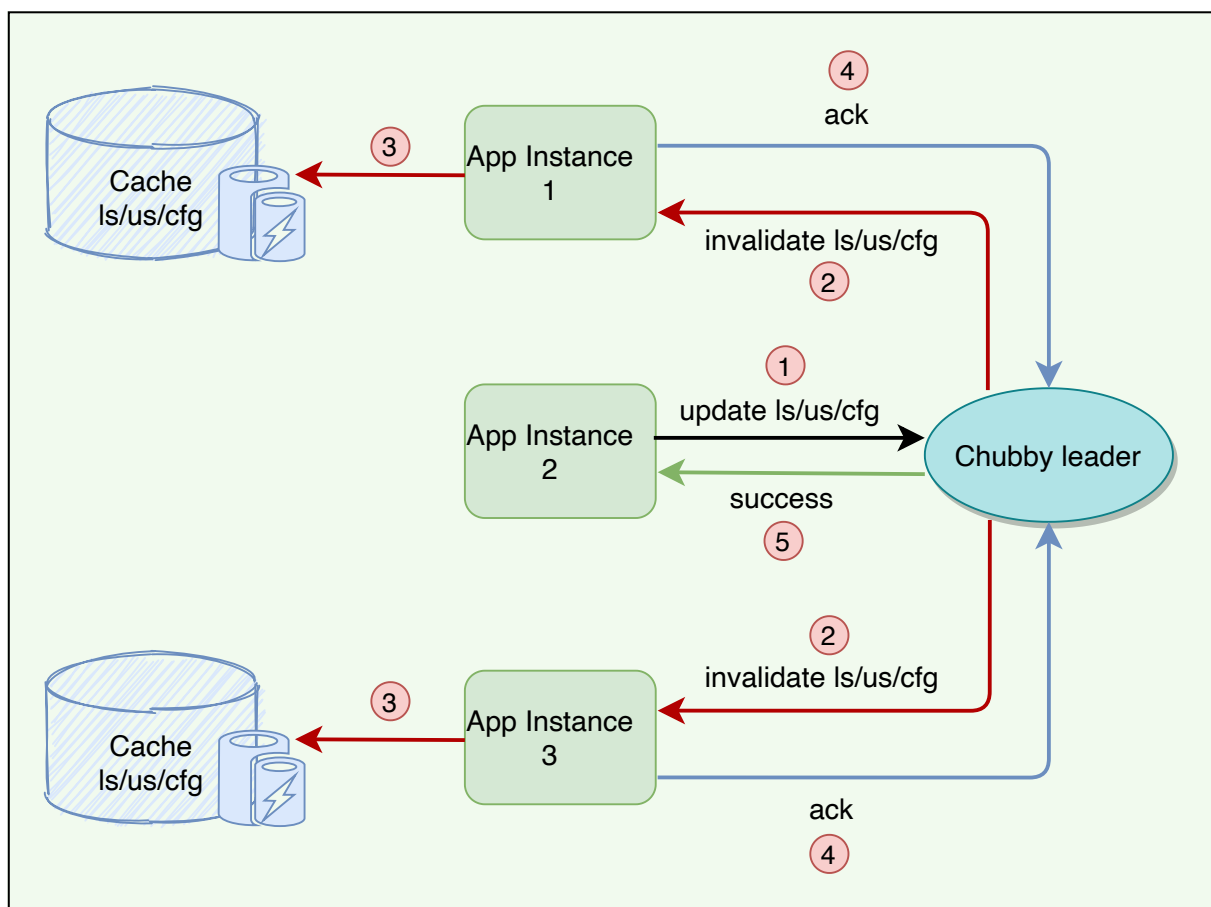
In Chubby, caching plays an important role, as read requests greatly outnumber write requests. To reduce read traffic, Chubby clients cache file contents, node metadata, and information on open handles in a consistent, write-through cache in the client's memory. Because of this caching, Chubby must maintain 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.



Cache invalidation

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



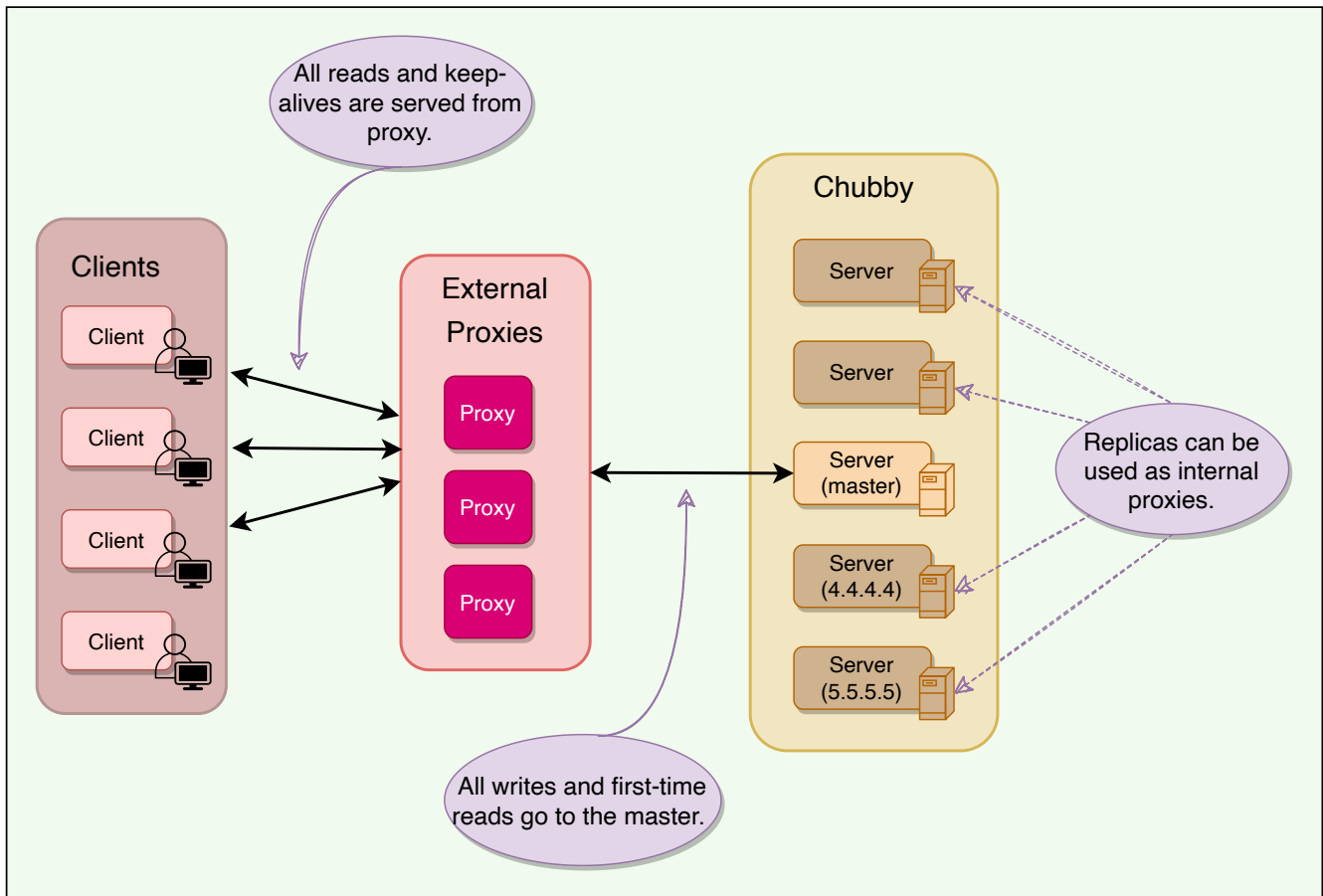
- Proxies
- Partitioning
- Learnings

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.

Google built GFS keeping the following goals in mind:

- **Scalable:** GFS should run reliably on a very large system built from commodity hardware.
- **Fault-tolerant:** The design must be sufficiently tolerant of hardware and software failures to enable application-level services to continue their operation in the face of any likely combination of failure conditions.
- **Large files:** Files stored in GFS will be huge. Multi-GB files are common.
- **Large sequential and small random reads:** The workloads primarily consist of two kinds of reads: large, streaming reads and small, random reads.
- **Sequential writes:** The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again.
- **Not optimized for small data:** Small, random reads and writes do occur and are supported, but the system is not optimized for such cases.
- **Concurrent access:** The level of concurrent access will also be high, with large numbers of concurrent appends being particularly prevalent, often accompanied by concurrent reads.
- **High throughput:** GFS should be optimized for high and sustained throughput in reading the data, and this is prioritized over latency. This is not to say that latency is unimportant; rather, GFS needs to be optimized for high-performance reading and appending large volumes of data for the correct operation of the system.

GFS use cases

- GFS is a distributed file system built for large, distributed data-intensive applications like **Gmail** or **YouTube**.

- Originally, it was built to store data generated by Google's large **crawling and indexing system**.
- Google's **BigTable** uses the distributed Google File System to store log and data files.

APIs

GFS does not provide standard POSIX-like APIs; instead, user-level APIs are provided. In GFS, files are organized hierarchically in directories and identified by their pathnames. GFS supports the usual file system operations:

`create` – To create a new instance of a file.

`delete` – To delete an instance of a file.

`open` – To open a named file and return a handle.

`close` – To close a given file specified by a handle.

`read` – To read data from a specified file and offset.

`write` – To write data to a specified file and offset.

In addition, GFS supports two special operations:

- **Snapshot:** A snapshot is an efficient way of creating a copy of the current instance of a file or directory tree.
- **Append:** An append operation allows multiple clients to append data to the same file concurrently while guaranteeing atomicity. It is useful for implementing multi-way merge results and producer-consumer queues that many clients can simultaneously append to without additional locking.

[< Back](#)

[Next >](#)

High-level Architecture

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

We'll cover the following



- Chunks
- Chunk handle
- Cluster
- ChunkServer
- Master
- Client

A GFS cluster consists of a single master and multiple ChunkServers and is accessed by multiple clients.

Chunks

As files stored in GFS tend to be very large, GFS breaks files into multiple fixed-size chunks where each chunk is 64 megabytes in size.

Chunk handle

Each chunk is identified by an immutable and globally unique 64-bit ID number called chunk handle. This allows for 2^{64} unique chunks. If each chunk is 64 MB, total storage space would be more than 10^9 exa-bytes.

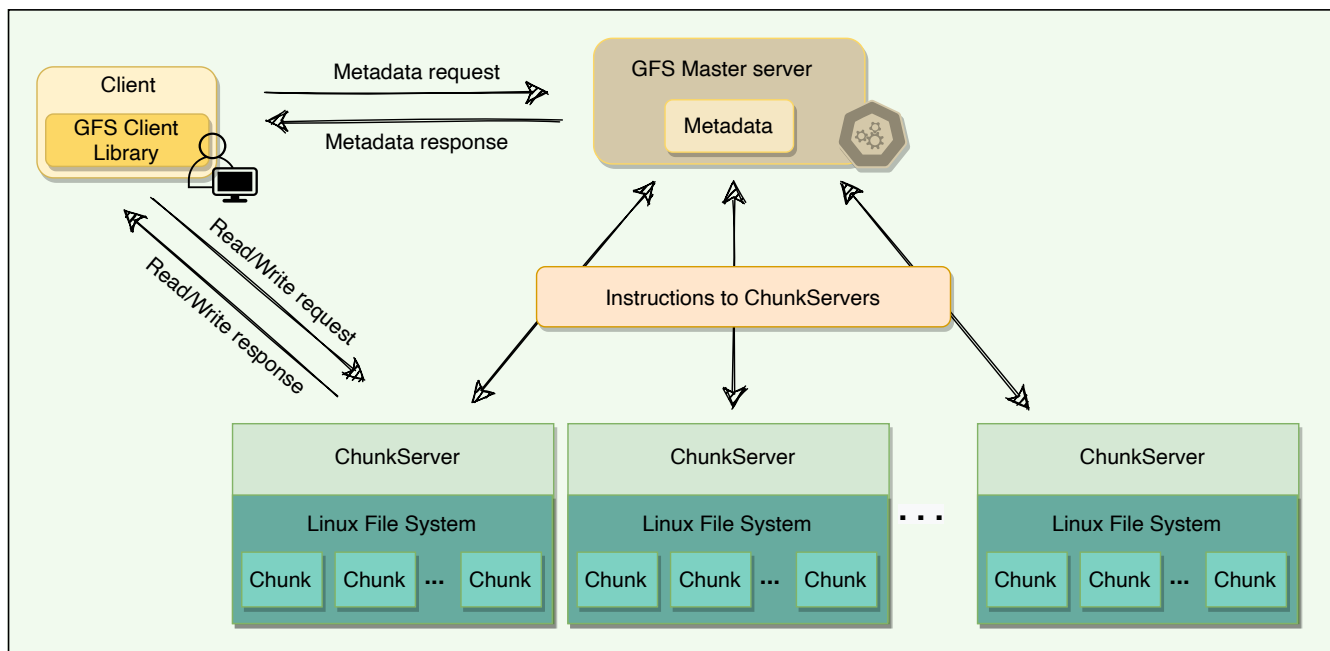
As files are split into chunks, therefore, the job of GFS is to provide a mapping from files to chunks, and then to support standard operations on files, mapping down to operations on individual chunks.

Cluster

GFS is organized into a simple network of computers called a cluster. All GFS clusters contain three kinds of entities:

1. A single master server
2. Multiple ChunkServers
3. Many clients

The master stores all metadata about the system, while the ChunkServers store the real file data.

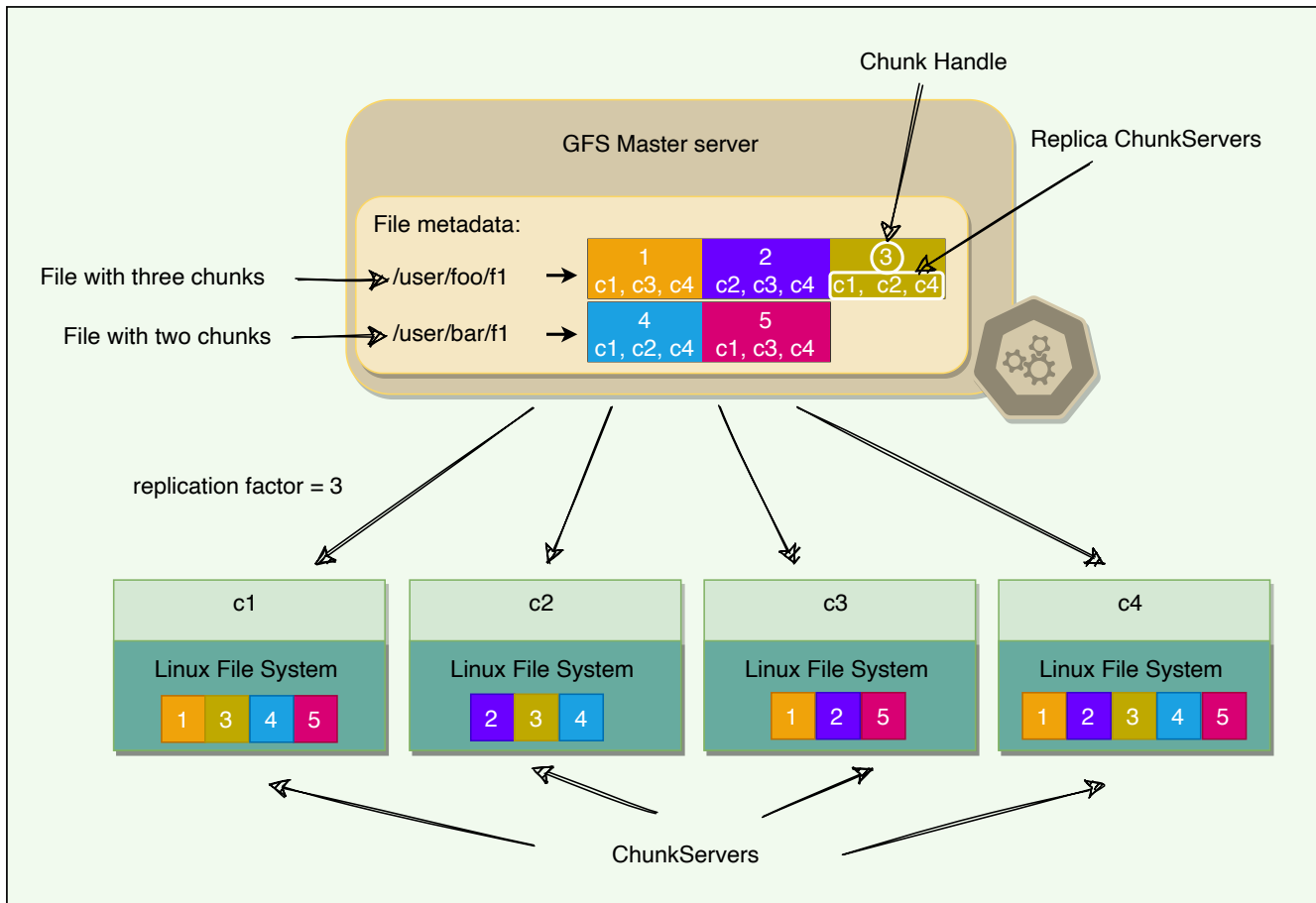


GFS high-level architecture

ChunkServer

ChunkServers store chunks on local disks as regular Linux files and read or write chunk data specified by a chunk handle and byte-range.

For reliability, each chunk is replicated to multiple ChunkServers. By default, GFS stores three replicas, though different replication factors can be specified on a per-file basis.



Chunk replication

Master

Master server is the coordinator of a GFS cluster and is responsible for keeping track of filesystem metadata:

1. The metadata stored at the master includes:
 - Name and directory of each file
 - Mapping of each file to its chunks
 - Current locations of chunks
 - Access control information
2. The master also controls system-wide activities such as chunk lease management (locks on chunks with expiration), garbage collection of orphaned chunks, and chunk migration between ChunkServers. Master assigns chunk handle to chunks at time of chunk creation.
3. The master periodically communicates with each ChunkServer in HeartBeat messages to give it instructions and collect its state.
4. For performance and fast random access, all metadata is stored in the master's main memory. This includes the entire filesystem namespace as well as all the name-to-chunk mappings.
5. For fault tolerance and to handle a master crash, all metadata changes are written to the disk onto an operation log. This operation log is also replicated onto remote machines. The operation log is similar to a journal. Every operation to the file system is logged into this file.
6. The master is a single point of failure, hence, it replicates its data onto several remote machines so that the master can be readily restored on failure.
7. The benefit of having a single, centralized master is that it has a global view of the file system, and hence, it can make optimum management decisions, for example, related to chunk placement.

Client

Client is an entity that makes a read or write request to GSF. GFS client library is linked into each application that uses GFS. This library communicates with the master for all metadata-related operations like

creating or deleting files, looking up files, etc. To read or write data, the client interacts directly with the ChunkServers that hold the data.

Neither the client nor the ChunkServer caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. ChunkServers rely on the buffer cache in Linux to maintain frequently accessed data in memory.

[< Back](#)

Google File System: Introduction

[Next >](#)

Single Master and Large Chunk Size

Single Master and Large Chunk Size

This lesson will explain why Chubby has a single master and a large chunk size.

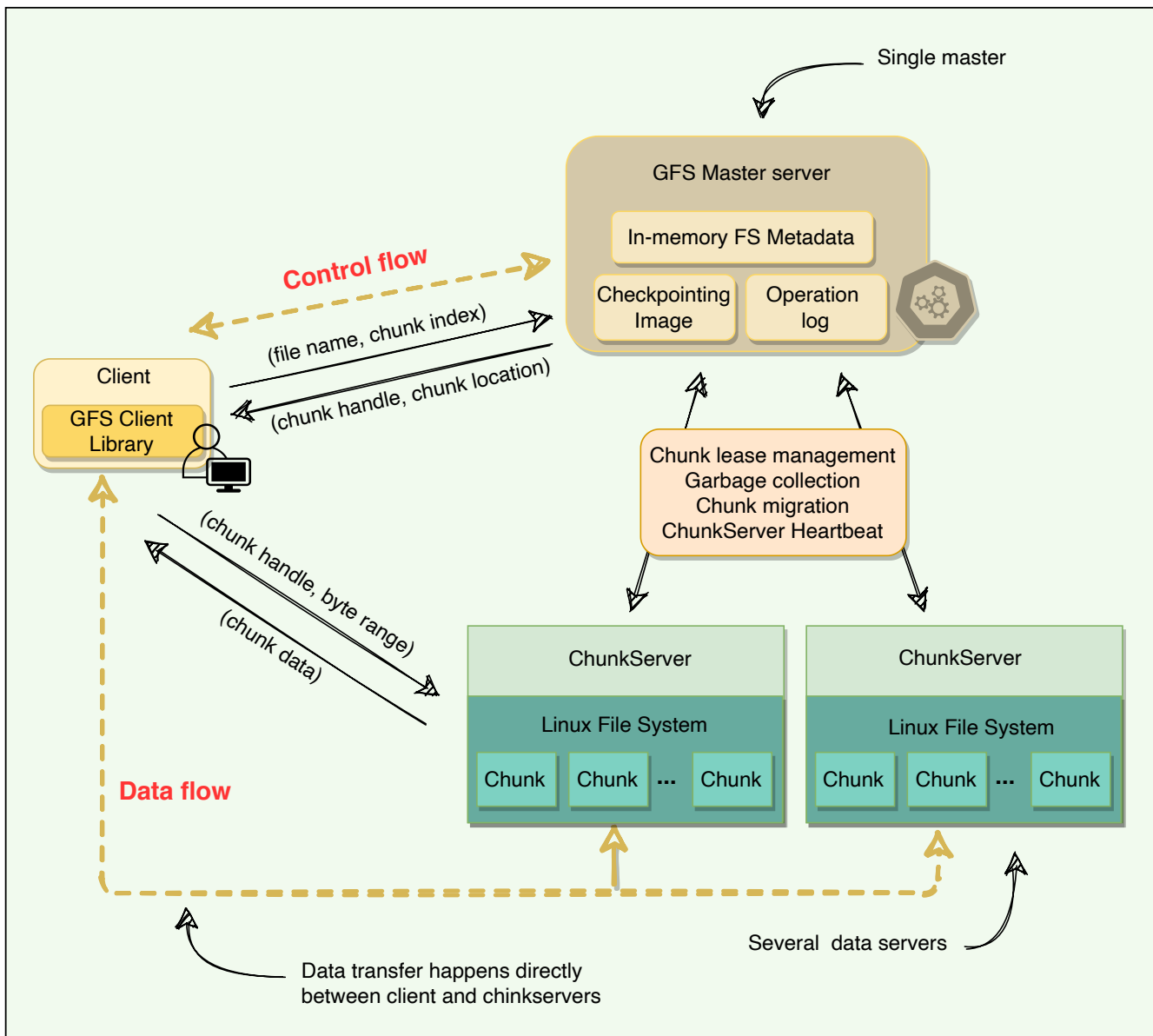
We'll cover the following



- Single master
- Chunk size
 - Lazy space allocation

Single master

Having a single master vastly simplifies GFS design and enables the master to make sophisticated chunk placement and replication decisions using global knowledge. However, GFS minimizes the master's involvement in reads and writes, so that it does not become a bottleneck. Clients never read or write file data through the master. Instead, a client asks the master which ChunkServers it should contact. The client caches this information for a limited time and interacts with the ChunkServers directly for many subsequent operations.



GFS's high-level architecture

Chunk size

Chunk size is one of the key design parameters. GFS has chosen 64 MB, which is much larger than typical filesystem block sizes (which are often around 4KB). Here are the advantages of using a large chunk size:

1. Since GFS was designed to handle huge files, small chunk sizes would not make a lot of sense, as each file would then have a map of a huge number of chunks.

2. As the master holds the metadata and manages file distribution, it is involved whenever chunks are read, modified, or deleted. A small chunk size would significantly increase the amount of data a master would need to manage, and also, increase the amount of data that would need to be communicated to a client, resulting in extra network traffic.
3. A large chunk size reduces the size of the metadata stored on the master, which enables the master to keep all the metadata in memory, thus significantly decreasing the latency for control operations.
4. By using a large chunk size, GFS reduces the need for frequent communication with the master to get chunk location information. It becomes feasible for a client to cache all information related to chunk locations of a large file. Client metadata caches have timeouts to reduce the risk of caching stale data.
5. A large chunk size also makes it possible to keep a TCP connection open to a ChunkServer for an extended time, amortizing the time of setting up a TCP connection.
6. A large chunk size simplifies ChunkServer management, i.e., to check which ChunkServers are near capacity or which are overloaded.
7. Large chunk size provides highly efficient sequential reads and appends of large amounts of data.

Lazy space allocation

Each chunk replica is stored as a plain Linux file on a ChunkServer. GFS does not allocate the whole 64MB of disk space when creating a chunk. Instead, as the client appends data, the ChunkServer, lazily extends the chunk. This lazy space allocation avoids wasting space due to internal fragmentation. Internal fragmentation refers to having unused portions of the 64 MB chunk. For example, if we allocate a 64 MB chunk and only fill up 20MB, the remaining space is unused.

One disadvantage of having a large chunk size is the handling of small files. Since a small file will have one or a few chunks, the ChunkServers storing those chunks can become hotspots if a lot of clients access the same file. To handle this scenario, GFS stores such files with a higher replication factor and also adds a random delay in the start times of the applications accessing these files.

[!\[\]\(d0a1791f26d167e866e44ebbf83efebe_img.jpg\) Back](#)

High-level Architecture

[Next !\[\]\(eafc244b53721dd1ec133f0772f70fc7_img.jpg\)](#)

Metadata

Metadata

Let's explore how GFS manages the filesystem metadata.

We'll cover the following



- Storing metadata in memory
- Chunk location
- Operation log
 - Checkpointing

The master stores three types of metadata:

1. The file and chunk namespaces (i.e., directory hierarchy).
2. The mapping from files to chunks.
3. The locations of each chunk's replicas.

There are three aspects of how master manages the metadata:

1. Master keeps all this metadata in memory.
2. The first two types (i.e., namespaces and file-to-chunk mapping) are also persisted on the master's local disk.
3. The third (i.e., chunk replicas' locations) is not persisted.

Let's discuss these aspects one by one.

Storing metadata in memory

Since metadata is stored in memory, the master operates very quickly. Additionally, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement three functions:

1. Chunk garbage collection
2. Re-replication in the case of ChunkServer failures
3. Chunk migration to balance load and disk-space usage across ChunkServers

As discussed above, one potential concern for this memory-only approach is that the number of chunks, and hence the capacity of the whole system, is limited by how much memory the master has. This is not a serious problem in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less than 64 bytes per file because the master stores file names compactly using **prefix compression**.

If the need for supporting an even larger file system arises, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility gained by storing the metadata in memory.

Chunk location

The master does not keep a persistent record of which ChunkServers have a replica of a given chunk; instead, the master asks each chunk server about its chunks at master startup, and whenever a ChunkServer joins the cluster. The master can keep itself up-to-date after that because it controls all chunk placements and monitors ChunkServer status with regular HeartBeat messages.

By having the ChunkServer as the ultimate source of truth of each chunk's location, GFS eliminates the problem of keeping the master and ChunkServers in sync. It is not beneficial to maintain a consistent view of chunk locations on the master, because errors on a ChunkServer may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled, or ChunkServer is renamed or failed, etc.) In a cluster with hundreds of servers, these events happen all too often.

Operation log

The master maintains an operation log that contains the namespace and file-to-chunk mappings and stores it on the local disk. Specifically, this log stores a historical record of all the metadata changes. Operation log is very important to GFS. It contains the persistent record of metadata and serves as a logical timeline that defines the order of concurrent operations.

For fault tolerance and reliability, this operation log is replicated on multiple remote machines, and changes to the metadata are not made visible to clients until they have been persisted on all replicas. The master batches several log records together before flushing, thereby reducing the impact of flushing and replicating on overall system throughput.

Upon restart, the master can restore its file-system state by replaying the operation log. This log must be kept small to minimize the startup time, and that is achieved by periodically checkpointing it.

Checkpointing

Master's state is periodically serialized to disk and then replicated, so that on recovery, a master may load the checkpoint into memory, replay any subsequent operations from the operation log, and be available again very

quickly. To further speed up the recovery and improve availability, GFS stores the checkpoint in a compact B-tree like format that can be directly mapped into memory and used for namespace lookup without extra parsing.

The checkpoint process can take time, therefore, to avoid delaying incoming mutations, the master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch.

[< Back](#)[Next >](#)

Single Master and Large Chunk Size

Master Operations

Master Operations

Let's learn the different operations performed by the master.

We'll cover the following



- Namespace management and locking
- Replica placement
 - Replica creation and re-replication
 - Replica rebalancing
- Stale replica detection

The master executes all namespace operations. Furthermore, it manages chunk replicas throughout the system. It is responsible for:

- Making replica placement decisions
- Creating new chunks and hence replicas
- Making sure that chunks are fully replicated according to the replication factor
- Balancing the load across all the ChunkServers
- Reclaim unused storage

Namespace management and locking

#

The master acquires locks over a namespace region to ensure proper serialization and to allow multiple operations at the master. GFS does not have an i-node like tree structure for directories and files. Instead, it has a hash-map that maps a filename to its metadata, and reader-writer locks are applied on each node of the hash table for synchronization.

- Each absolute file name or absolute directory name has an associated read-write lock.
- Each master operation acquires a set of locks before it runs.
- To make operation on `/dir1/dir2/leaf`, it first needs the following locks:
 - Reader lock on `/dir1`
 - Reader lock on `/dir1/dir2`
 - Reader or Writer lock on `/dir1/dir2/leaf`
- Following this scheme, concurrent writes on the same leaf are prevented right away. However, at the same time, concurrent modifications in the same directory are allowed.
- File creation does not require write-lock on the parent directory; a read-lock on its name is sufficient to protect the parent directory from deletion, rename, or snapshot.
- Write-lock on a file name stops attempts to create multiple files with the same name.
- Locks are acquired in a consistent order to prevent deadlock:
 - First ordered by level in the namespace tree
 - Lexicographically ordered within the same level

Replica placement

To ensure maximum data availability and integrity, the master distributes replicas on different racks, so that clients can still read or write in case of a rack failure. As the in and out bandwidth of a rack may be less than the sum of the bandwidths of individual machines, placing the data in various racks can help clients exploit reads from multiple racks. For ‘write’ operations, multiple racks are actually disadvantageous as data has to travel longer distances. It is an intentional tradeoff that GFS made.

Replica creation and re-replication

The goals of a master are to place replicas on servers with less-than-average disk utilization, spread replicas across racks, and reduce the number of ‘recent’ creations on each ChunkServer (even though writes are cheap, they are followed by heavy write traffic) which might create additional load.

Chunks need to be re-replicated as soon as the number of available replicas falls (due to data corruption on a server or a replica being unavailable) below the user-specified replication factor. Instead of re-replicating all of such chunks at once, the master prioritizes re-replication to prevent these cloning operations from becoming bottlenecks. Restrictions are placed on the bandwidth of each server for re-replication so that client requests are not compromised.

How are chunks prioritized for re-replication?

- A chunk is prioritized based on how far it is from its replication goal. For example, a chunk that has lost two replicas will be given priority on a chunk that has lost only one replica.
- GFS prioritizes chunks of live files as opposed to chunks that belong to recently deleted files (more on this when we discuss Garbage Collection (<https://www.educative.io/collection/page/5668639101419520/5559029852536832/5335676130689024>)). Deleted files are not removed immediately; instead, they are renamed temporarily and garbage-

collected after a few days. Replicas of deleted files can exist for a few days as well.

Replica rebalancing

Master rebalances replicas regularly to achieve load balancing and better disk space usage. It may move replicas from one ChunkServer to another to bring disk usage in a server closer to the average. Any new ChunkServer added to the cluster is filled up gradually by the master rather than flooding it with a heavy traffic of write operations.

Stale replica detection

Chunk replicas may become stale if a ChunkServer fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a chunk Version Number to distinguish between up-to-date and stale replicas. The master increments the chunk version every time it grants a lease (more on this later) and informs all up-to-date replicas. The master and these replicas all record the new version number in their persistent state. If the ChunkServer hosting a chunk replica is down during a mutation, the chunk replica will become stale and will have an older version number. The master will detect this when the ChunkServer restarts and reports its set of chunks and their associated version numbers. Master removes stale replicas during regular garbage collection.

Stale replicas are not given to clients when they ask the master for a chunk location, and they are not involved in mutations either. However, because a client caches a chunk's location, it may read from a stale replica before the data is resynced. The impact of this is low due to the fact that most operations to a chunk are append-only. This means that a stale replica usually returns a premature end of a chunk rather than outdated data for a value.

← Back

Metadata

Next →

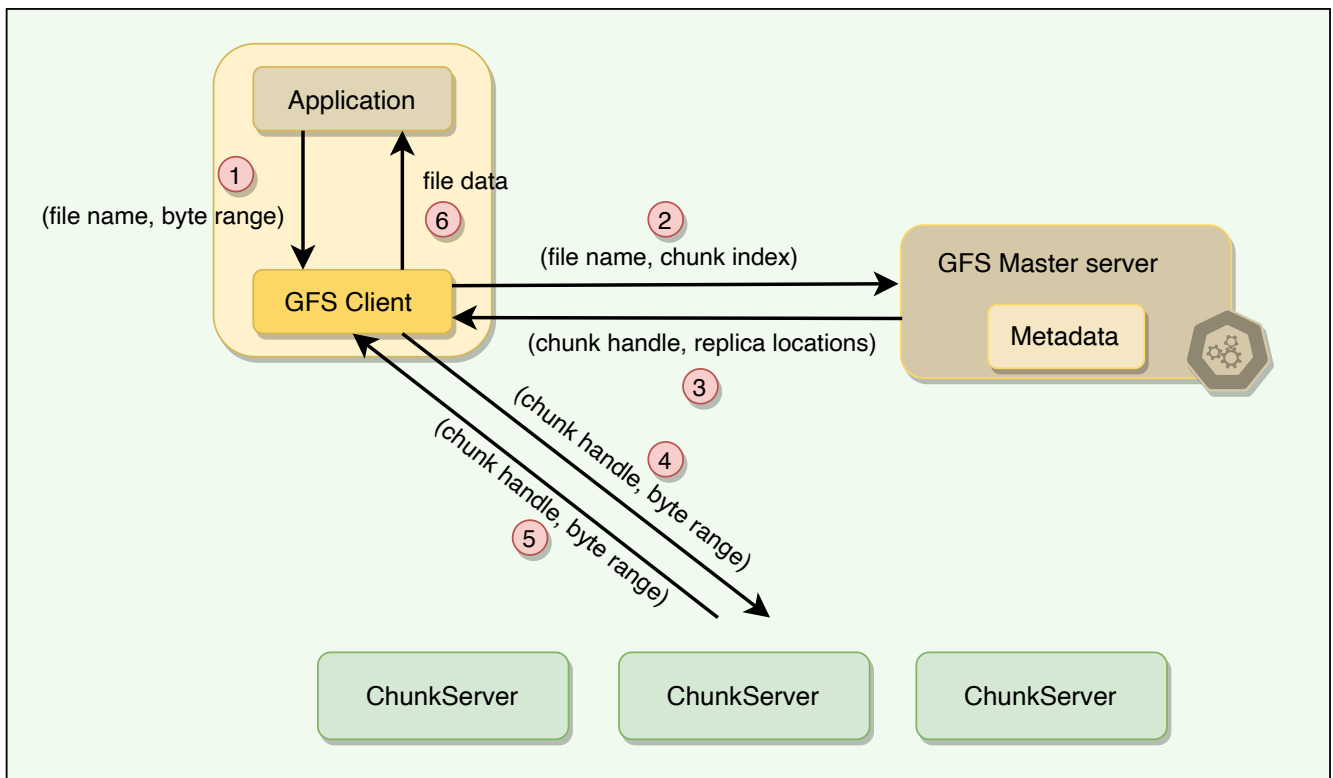
Anatomy of a Read Operation

Anatomy of a Read Operation

Let's learn how GFS handles a read operation.

A typical read interaction with a GFS cluster by a client application goes like this:

1. First, the client translates the file name and byte offset specified by the application into a chunk index within the file. Given the fixed chunk size, this can be computed easily.
2. The client then sends the master an RPC request containing the file name and chunk index.
3. The master replies with the chunk handle and the location of replicas holding the chunk. The client caches this metadata using the file name and chunk-index as the key. This information is subsequently used to access the data.
4. The client then sends a request to one of the replicas (the closest one). The request specifies the chunk handle and a byte range within that chunk.
 - Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened.
 - In fact, the client typically asks for multiple chunks in the same request, and the master can also include the information for chunks immediately following those requested.
5. The replica ChunkServer replies with the requested data.
6. As evident from the above workflow, the master is involved at the start and is then completely out of the loop, implementing a separation of control and data flows – a separation that is crucial for maintaining high performance of file accesses.



The anatomy of a read operation

← Back

Next →

Master Operations

Anatomy of a Write Operation

Anatomy of a Write Operation

Let's learn how GFS handles a write operation.

We'll cover the following ^

- What is a chunk lease?
- Data writing

What is a chunk lease?

To safeguard against concurrent writes at two different replicas of a chunk, GFS makes use of chunk lease. When a mutation (i.e., a write, append or delete operation) is requested for a chunk, the master finds the ChunkServers which hold that chunk and grants a chunk lease (for 60 seconds) to one of them. The server with the lease is called the primary and is responsible for providing a serial order for all the currently pending concurrent mutations to that chunk. There is only one lease per chunk at any time, so that if two write requests go to the master, both see the same lease denoting the same primary.

Thus, a global ordering is provided by the ordering of the chunk leases combined with the order determined by that primary. The primary can request lease extensions if needed. When the master grants the lease, it increments the chunk version number and informs all replicas containing that chunk of the new version number.

Data writing

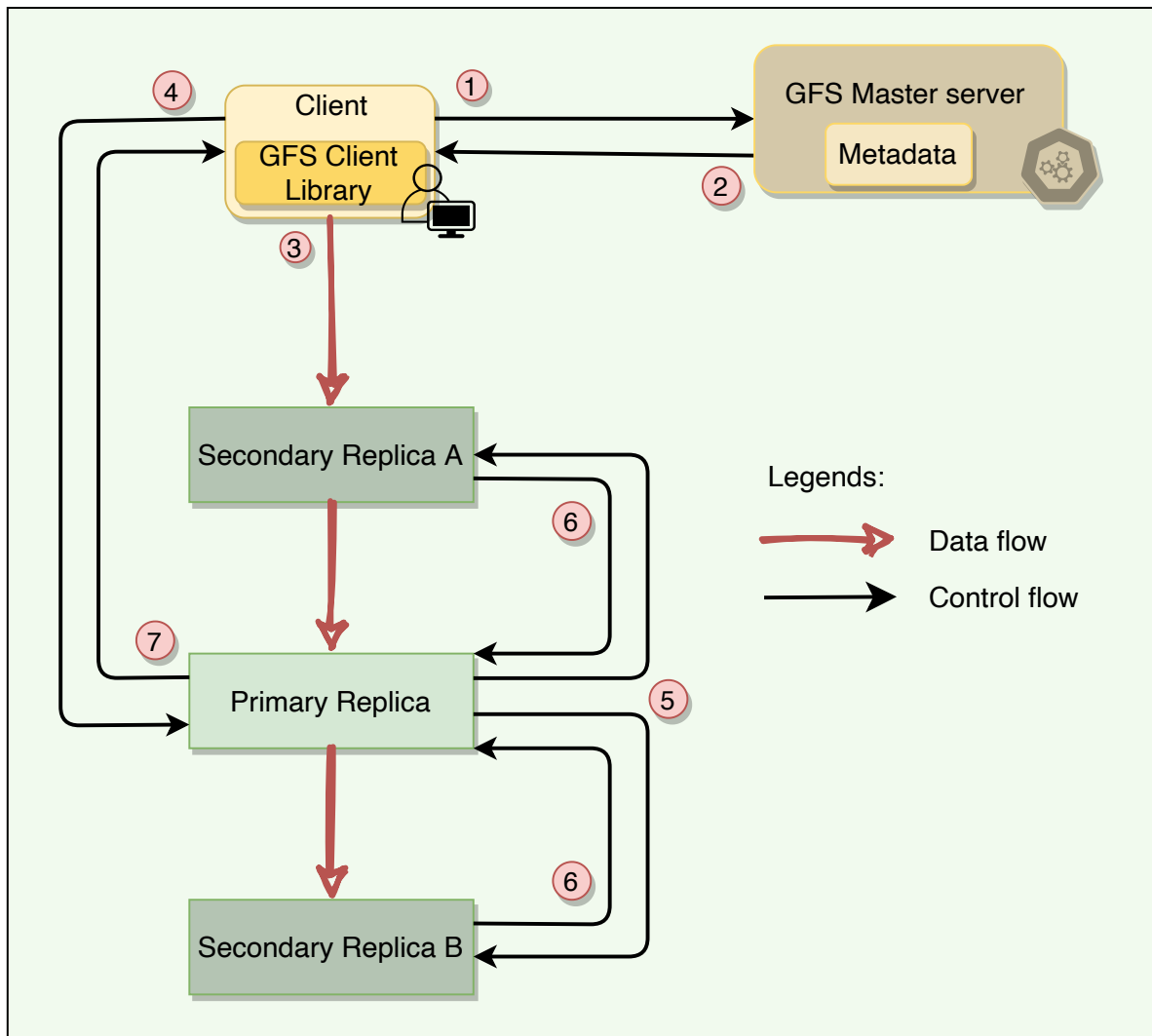
The actual writing of data is split into two phases:

- **Sending:** First, the client is given a list of replicas that identifies the primary ChunkServer and secondaries. The client sends the data to the closest replica. Then replicas send the data in chain to all other replicas to maximize bandwidth and throughput. Eventually, all the replicas get the data, which is not yet written to a file but sits in a cache.
- **Writing:** When the client gets an acknowledgment from all replicas that the data has been received, it then sends a write request to the primary, identifying the data that was sent in the previous phase. The primary is responsible for the serialization of writes. It assigns consecutive serial numbers to all write requests that it has received, applies the writes to the file in serial-number order, and forwards the write requests in that order to the secondaries. Once the primary gets acknowledgments from all the secondaries, the primary responds back to the client, and the write operation is complete. Any errors at any stage in this process are met with retries and eventual failure. On failure, an error is returned to the client.

Following is the stepwise breakdown of the data transfer:

1. Client asks master which chunk server holds the current lease of chunk and locations of other replicas.
2. Master replies with the identity of primary and locations of the secondary replicas.
3. Client pushes data to the closest replica. Then replicas send the data in chain to all other replicas.
4. Once all replicas have acknowledged receiving the data, the client sends the write request to the primary. The primary assigns consecutive serial numbers to all the mutations it receives, providing serialization. It applies mutations in serial number order.

5. Primary forwards the write request to all secondary replicas. They apply mutations in the same serial number order.
6. Secondary replicas reply to primary indicating they have completed operation.
7. Primary replies to the client with success or error message



The anatomy of a write operation

The key point to note is that the data flow is different from the control flow. The data flows from the client to a ChunkServer and then from that ChunkServer to another ChunkServer, until all ChunkServers that store replicas for that chunk have received the data. The control (the write request) flow goes from the client to the primary ChunkServer for that

chunk. The primary then forwards the request to all the secondaries. This ensures that the primary controls the order of writes even if it receives multiple concurrent write requests. All replicas will have data written in the same sequence. Chunk version numbers are used to detect if any replica has stale data which has not been updated because that ChunkServer was down during some update.

[< Back](#)[Next >](#)

Anatomy of a Read Operation

Anatomy of an Append Operation

Anatomy of an Append Operation

Let's learn how GFS handles an append operation.

Record append operation is optimized in a unique way that distinguishes GFS from other distributed file systems. In a normal write, the client specifies the offset at which data is to be written. Concurrent writes to the same region can experience race conditions, and the region may end up containing data fragments from multiple clients. In a record append, however, the client specifies only the data. GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client. This process is similar to the append operation on a file opened with `O_APPEND` (<https://man7.org/linux/man-pages/man2/open.2.html>) mode on a POSIX-compliant file system but without the race conditions when multiple writers do so concurrently.

Record Append is a kind of mutation that changes the contents of the metadata of a chunk. When an application tries to append data on a chunk by sending a request to the client, the client pushes the data to all replicas of the last chunk of the file just like the write operation. When the client forwards the request to the master, the primary checks whether appending the record to the existing chunk will increase the chunk's size more than its limit (maximum size of a chunk is 64MB). If this happens, it pads the chunk to the maximum limit, commands the secondary to do the same, and requests the clients to try to append to the next chunk. If the record fits within the maximum size, the primary appends the data to its replica, tells the secondary to write the data at the exact offset where it has, and finally replies success to the client.

If an append operation fails at any replica, the client retries the operation. Due to this reason, replicas of the same chunk may contain different data, possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are byte-wise identical; instead, it only ensures that the data is written at-least-once as an atomic unit.

[< Back](#)[Next >](#)

Anatomy of a Write Operation

GFS Consistency Model and Snapshot...

GFS Consistency Model and Snapshotting

This lesson will explain how GFS handles the consistency of its operations and data. Additionally, we will look into how GFS implements a snapshotting operation.

We'll cover the following



- GFS consistency model
- Snapshotting

GFS consistency model

To keep things simple and efficient, GFS has a relaxed consistency model.

Metadata operations (e.g., file creation) are atomic. They are handled exclusively by the master. Namespace locking guarantees atomicity and correctness, whereas the master's operation log defines a global total order of these operations.

In data mutations, there is an important distinction between `write` and `append` operations. Write operations specify an offset at which mutations should occur, whereas `appends` are always applied at the end of the file. This means that for the `write` operation, the offset in the chunk is predetermined, whereas for `append`, the system decides. Concurrent writes to the same location are not serializable and may result in corrupted regions of the file. With `append` operations, GFS guarantees the append will happen at-least-once and atomically (that is, as a contiguous sequence of bytes). The system does not guarantee that all copies of the chunk will be identical (some may have duplicate data).

Snapshotting

A snapshot is a copy of some subtree of the global namespace as it exists at a given point in time. GFS clients use snapshotting to efficiently branch two versions of the same data. Snapshots in GFS are initially **zero-copy**. This means that data copies are made only when clients make a request to modify the chunks. This scheme is known as **copy-on-write**.

When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files to snapshot. It waits for leases to be revoked or expired and logs the snapshot operation to the operation log. The snapshot is then made by duplicating the metadata for the source directory tree. Newly created snapshot files still point to the original chunks.

When a client makes a request to write to one of these chunks, the master detects that it is a copy-on-write chunk by examining its reference count (which will be more than one). At this point, the master asks each ChunkServer holding the replica to make a copy of the chunk and store it locally. These local copies are made to avoid copying the chunk over the network. Once the copy is complete, the master issues a lease for the new copy, and the write proceeds.

[< Back](#)

Anatomy of an Append Operation

[Next >](#)

Fault Tolerance, High Availability, and ...

Fault Tolerance, High Availability, and Data Integrity

Let's learn how GFS implements fault tolerance, high availability, and data integrity.

We'll cover the following



- Fault tolerance
- High availability through Chunk replication
- Data integrity through checksum

Fault tolerance

To make the system fault-tolerant and available, GFS makes use of two simple strategies:

1. **Fast recovery** in case of component failures.
2. **Replication** for high availability.

Let's first see how GFS recovers from master or replica failure:

- **On master failure:** The Master being a single point of failure, can make the entire system unavailable in a short time. To handle this, all operations applied on master are saved in an **operation log**. This log is checkpointed and replicated on multiple remote machines, so that on recovery, a master may load the checkpoint into memory, replay any subsequent operations from the operation log, and be available again in a short amount of time. GFS relies on an external monitoring

infrastructure to detect the master failure and switch the traffic to the backup master server.

- **Shadow masters** are replicas of master and provide read-only access to the file system even when the primary is down. All shadow masters keep themselves updated by applying the same sequence of updates exactly as the primary master does by reading its operation log. Shadow masters may lag the primary slightly, but they enhance read availability for files that are not being actively changed or applications that do not mind getting slightly stale metadata. Since file contents are read from the ChunkServers, applications do not observe stale file contents.
- **On primary replica failure:** If an active primary replica fails (or there is a network partition), the master detects this failure (as there will be no heartbeat), and waits for the current lease to expire (in case the primary replica is still serving traffic from clients directly), and then assigns the lease to a new node. When the old primary replica recovers, the master will detect it as 'stale' by checking the version number of the chunks. The master node will pick new nodes to replace the stale node and garbage-collect it before it can join the group again.
- **On secondary replica failure:** If there is a replica failure, all client operations will start failing on it. When this happens, the client retries a few times; if all of the retries fail, it reports failure to the master. This can leave the secondary replica inconsistent because it misses some mutations. As described above, stale nodes will be replaced by new nodes picked by the master, and eventually garbage-collected.

 **Note:** Stale replicas might be exposed to clients. It depends on the application programmer to deal with these stale reads. GFS does not guarantee strong consistency on chunk reads.

High availability through Chunk replication

As discussed earlier, each chunk is replicated on multiple ChunkServers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones the existing replicas to keep each chunk fully replicated as ChunkServers go offline or when the master detects corrupted replicas through checksum verification.

A chunk is lost irreversibly only if all its replicas are lost before GFS can react. Even in this case, the data becomes unavailable, not corrupted, which means applications receive clear errors rather than corrupt data.

Data integrity through checksum

Checksumming is used by each ChunkServer to detect the corruption of stored data. The chunk is broken down into 64 KB blocks. Each has a corresponding 32-bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

1. **For reads**, the ChunkServer verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another ChunkServer. Therefore, ChunkServers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the ChunkServer returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, and the master will clone the chunk from another replica. After a valid new replica is in place, the

master instructs the ChunkServer that reported the mismatch to delete its replica.

2. **For writes**, ChunkServer verifies the checksum of first and last data blocks that overlap the write range before performing the write. Then, it computes and records the new checksums. For a corrupted block, the ChunkServer returns an error to the requestor and reports the mismatch to the master.
3. **For appends**, checksum computation is optimized as there is no checksum verification on the last block; instead, just incrementally update the checksum for the last partial block and compute new checksums for any brand-new blocks filed by the append. This way, if the last partial block is already corrupted (and GFS fails to detect it now), the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

During idle periods, ChunkServers can scan and verify the contents of inactive chunks (prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk).

Checksumming has little effect on read performance for the following reasons:

- Since most of the reads span at least a few blocks, GFS needs to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries.
- Checksum lookups and comparisons on the ChunkServer are done without any I/O.
- Checksum calculation can often be overlapped with I/Os.

Garbage Collection

Let's learn how GFS implements garbage collection.

We'll cover the following



- Garbage collection through lazy deletion
- Advantages of lazy deletion
- Disadvantages of lazy deletion

Garbage collection through lazy deletion

When a file is deleted, GFS does not immediately reclaim the physical space used by that file. Instead, it **follows a lazy garbage collection strategy**.

When the client issues a delete file operation, GFS does two things:

1. The master logs the deletion operation just like other changes.
2. The deleted file is renamed to a hidden name that also includes a deletion timestamp.

The file can still be read under the new, special name and can also be undeleted by renaming it back to normal. To reclaim the physical storage, the master, while performing regular scans of the file system, removes any such hidden files if they have existed for more than three days (this interval is configurable) and also deletes its in-memory metadata. This lazy deletion scheme provides a window of opportunity to a user who deleted a file by mistake to recover the file.

The master, while performing regular scans of chunk namespace, deletes the metadata of all chunks that are not part of any file. Also, during the exchange of regular HeartBeat messages with the master, each ChunkServer reports a subset of the chunks it has, and the master replies with a list of chunks from that subset that are no longer present in the master's database; such chunks are then deleted from the ChunkServer.

Advantages of lazy deletion

Here are the advantages of lazy deletion.

- Simple and reliable. If the chunk deletion message is lost, the master does not have to retry. The ChunkServer can perform the garbage collection with the subsequent heartbeat messages.
- GFS merges storage reclamation into regular background activities of the master, such as the regular scans of the filesystem or the exchange of HeartBeat messages. Thus, it is done in batches, and the cost is amortized.
- Garbage collection takes place when the master is relatively free.
- Lazy deletion provides safety against accidental, irreversible deletions.

Disadvantages of lazy deletion

As we know, after deletion, storage space does not become available immediately. Applications that frequently create and delete files may not be able to reuse the storage right away. To overcome this, GFS provides following options:

- If a client deletes a deleted file again, GFS expedites the storage reclamation.
- Users can specify directories that are to be stored without replication.

- Users can also specify directories where deletion takes place immediately.

← Back

Fault Tolerance, High Availability, and ...

Next →

Criticism on GFS

Criticism on GFS

Here is the summary of criticism on GFS's architecture.

We'll cover the following



- Problems associated with single master
- Problems associated with large chunk size

Problems associated with single master

As GFS has grown in usage, Google has started to see the following problems with the centralized master scheme:

- Despite the separation of control flow (i.e., metadata operations) and data flow, the master is emerging as a bottleneck in the design. As the number of clients grows, a single master could not serve them because it does not have enough CPU power.
- Despite the reduced amount of metadata (because of the large chunk size), the amount of metadata stored by the master is increasing to a level where it is getting difficult to keep all the metadata in the main memory.

Problems associated with large chunk size

Large chunk size (64MB) in GFS has its disadvantages while reading. Since a small file will have one or a few chunks, the ChunkServers storing those chunks can become hotspots if a lot of clients are accessing the same file. As a workaround for this problem, GFS stores extra copies of small files for distributing the load to multiple ChunkServers. Furthermore, GFS adds a random delay in the start times of the applications accessing such files.

[!\[\]\(1d3a1175dd4902218e694b9c098adb83_img.jpg\) Back](#)[Next !\[\]\(c507f772dba2b921f86777f01218e570_img.jpg\)](#)

Garbage Collection

Summary: GFS

Summary: GFS

Here is a quick summary of Google File System for you!

We'll cover the following



- Summary
- System design patterns
- References and further reading

Summary

- GFS is a scalable distributed file storage system for large data-intensive applications.
- GFS uses commodity hardware to reduce infrastructure costs.
- GFS was designed with the understanding that system/hardware failures can and do occur.
- Reading workload consists of large streaming reads and small random reads. Writing workloads consists of many large, sequential writes that append data to files.
- GFS provides APIs for usual file operations like `create`, `delete`, `open`, `close`, `read`, and `write`. Additionally, GFS supports `snapshot` and `record append` operations. `Snapshot` creates a copy of the file or directory tree. `Record append` allows multiple clients to append data to the same file concurrently while guaranteeing atomicity.
- A GFS cluster consists of a **single master** and **multiple ChunkServers** and is accessed by multiple clients.

- **Chunk:** Files are broken into fixed-size chunks where each chunk is 64 megabytes in size. Each chunk is identified by an immutable and globally unique **64-bit chunk handle** assigned by the master at the time of chunk creation.
- ChunkServers store chunks on the local disk as Linux files.
- For reliability, each chunk is replicated on multiple ChunkServers.
- Master server is the coordinator of a GFS cluster and is responsible for keeping track of all the filesystem metadata. This includes namespace, authorization, mapping of files to chunks, and the current location of chunks.
- Master keeps all metadata in memory for faster operations. For fault tolerance and to handle a master crash, all metadata changes are written to the disk onto an **operation log**. This operation log is also replicated onto remote machines.
- The master does not keep a persistent record of which ChunkServers have a replica of a given chunk. Instead, the master asks each ChunkServer about what chunks it holds at master startup or whenever a ChunkServer joins the cluster.
- **Checkpointing:** The master's state is periodically serialized to disk and then replicated so that on recovery, a master may load the checkpoint into memory, replay any subsequent operations from the operation log, and be available again very quickly.
- **HeartBeat:** The master communicates with each ChunkServer through Heartbeat messages to pass instructions to it and collects its state.
- **Client:** GFS client code which is linked into each application, implements filesystem APIs, and communicates with the cluster. Clients interact with the master for metadata, but all data transfers happen directly between the client and ChunkServers.
- **Data Integrity:** Each ChunkServer uses checksumming to detect the corruption of stored data.

- **Garbage Collection:** After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels.
- **Consistency:** Master guarantees data consistency by ensuring the order of mutations on all replicas and using **chunk version numbers**. If a replica has an incorrect version, it is garbage collected.
- GFS guarantees **at-least-once writes** for writers. This means that records could be written more than once as well (although rarely). It is the responsibility of the readers to deal with these duplicate chunks. This is achieved by having checksums and serial numbers in the chunks, which help readers to filter and discard duplicate data.
- **Cache:** Neither the client nor the ChunkServer caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. However, clients do cache metadata.

System design patterns

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

- **Write-Ahead Log:** For fault-tolerance and in the event of a master crash, all metadata changes are written to the disk onto an operation log which is a write-ahead log.
- **HeartBeat:** The GFS master periodically communicates with each ChunkServer in HeartBeat messages to give it instructions and collect its state.
- **Checksum:** Each ChunkServer uses checksumming to detect the corruption of stored data.

References and further reading

- GFS paper (<https://research.google/pubs/pub51/>)
- Bigtable (<https://research.google/pubs/pub27898/>)
- GFS: Evolution on Fast-forward (<https://queue.acm.org/detail.cfm?id=1594206>)

← Back

Criticism on GFS

Next →

Quiz: GFS

HDFS: How to Design a Distributed File Storage System?

Hadoop Distributed File System: Introduction

This lesson gives a brief introduction to the Hadoop Distributed File System.

We'll cover the following



- Goal
- What is Hadoop Distributed File System (HDFS)?
- Background
- APIs

Goal

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

What is Hadoop Distributed File System (HDFS)?

HDFS is a distributed file system and was built to store unstructured data. It is designed to store huge files reliably and stream those files at high bandwidth to user applications.

HDFS is a variant and a simplified version of the Google File System (GFS). A lot of HDFS architectural decisions are inspired by GFS design. HDFS is built around the idea that the most efficient data processing pattern is a **write-**

once, read-many-times pattern.

Background

Apache Hadoop (<https://hadoop.apache.org/>) is a software framework that provides a distributed file storage system and distributed computing for analyzing and transforming very large data sets using the MapReduce (https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html) programming model. HDFS is the default file storage system in Hadoop. It is designed to be a **distributed, scalable, fault-tolerant** file system that primarily caters to the needs of the **MapReduce** paradigm.

Both HDFS and GFS were built to store very large files and scale to store petabytes of storage. Both were built for handling batch processing on huge data sets and were designed for data-intensive applications and not for end-users. Like GFS, HDFS is also not POSIX-compliant and is not a mountable file system on its own. It is typically accessed via HDFS clients or by using application programming interface (API) calls from the Hadoop libraries.

Given the current HDFS design, the following types of applications are not a good fit for HDFS:

1. **Low-latency data access:**

HDFS is optimized for high throughput (which may come at the expense of latency). Therefore, applications that need low-latency data access will not work well with HDFS.

2. **Lots of small files:**

HDFS has a central server called NameNode, which holds all the filesystem metadata in memory. This limits the number of files in the filesystem by the amount of memory on the NameNode. Although storing millions of files is feasible, billions are beyond the capability of the current hardware.

3. No concurrent writers and arbitrary file modifications:

Contrary to GFS, multiple writers cannot concurrently write to an HDFS file. Furthermore, writes are always made at the end of the file, in an append-only fashion; **there is no support for modifications at arbitrary offsets in a file.**

APIs

HDFS does not provide standard POSIX-like APIs. Instead, it exposes user-level APIs. In HDFS, files are organized hierarchically in directories and identified by their pathnames. HDFS supports the usual file system operations, e.g., files and directories can be **created, deleted, renamed, moved**, and **symbolic links** can be created. All **read** and **write** operations are done in an append-only fashion.

[< Back](#)[Next >](#)

Mock Interview: GFS

High-level Architecture

High-level Architecture

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

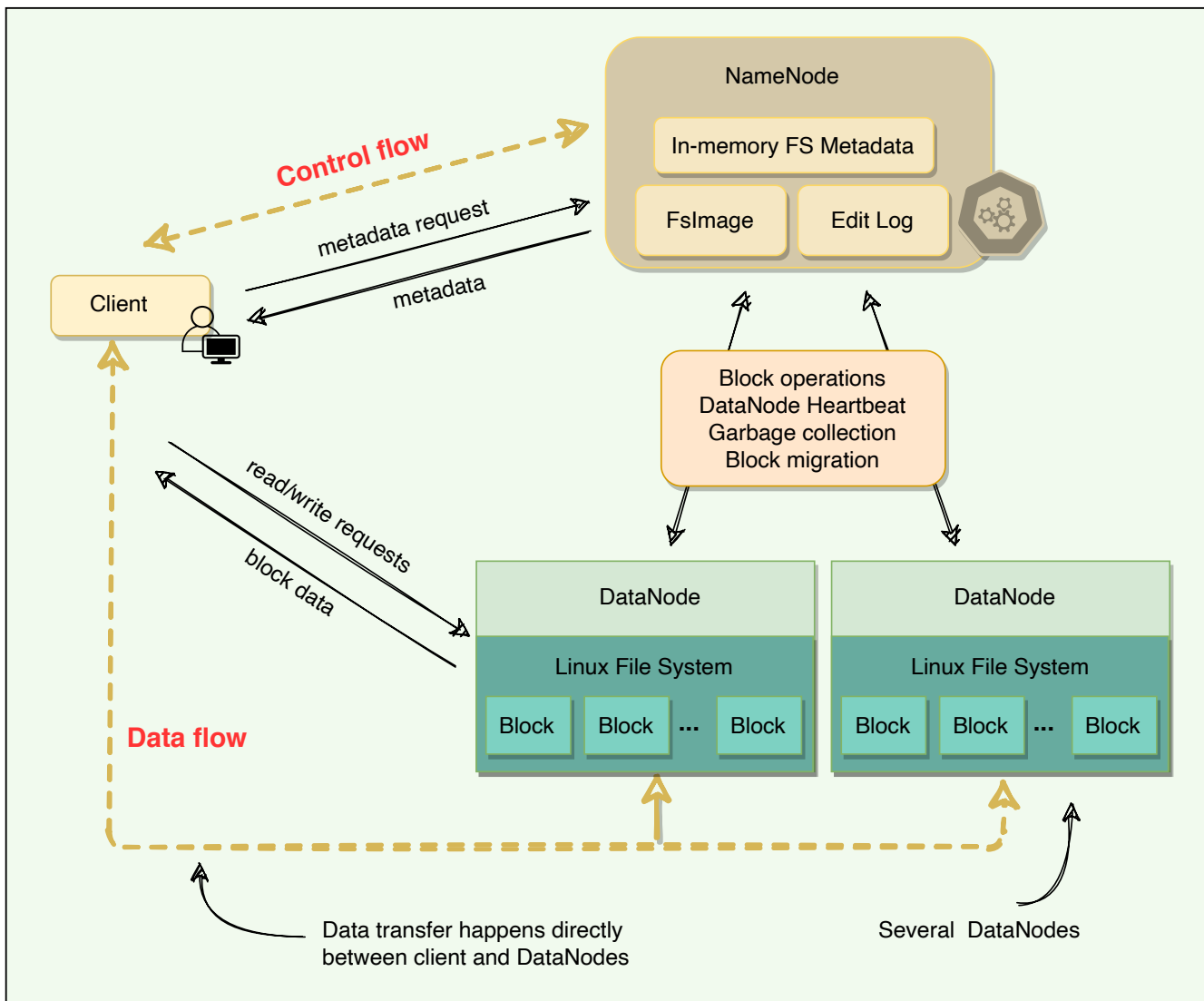
We'll cover the following



- HDFS architecture
- Comparison between GFS and HDFS

HDFS architecture

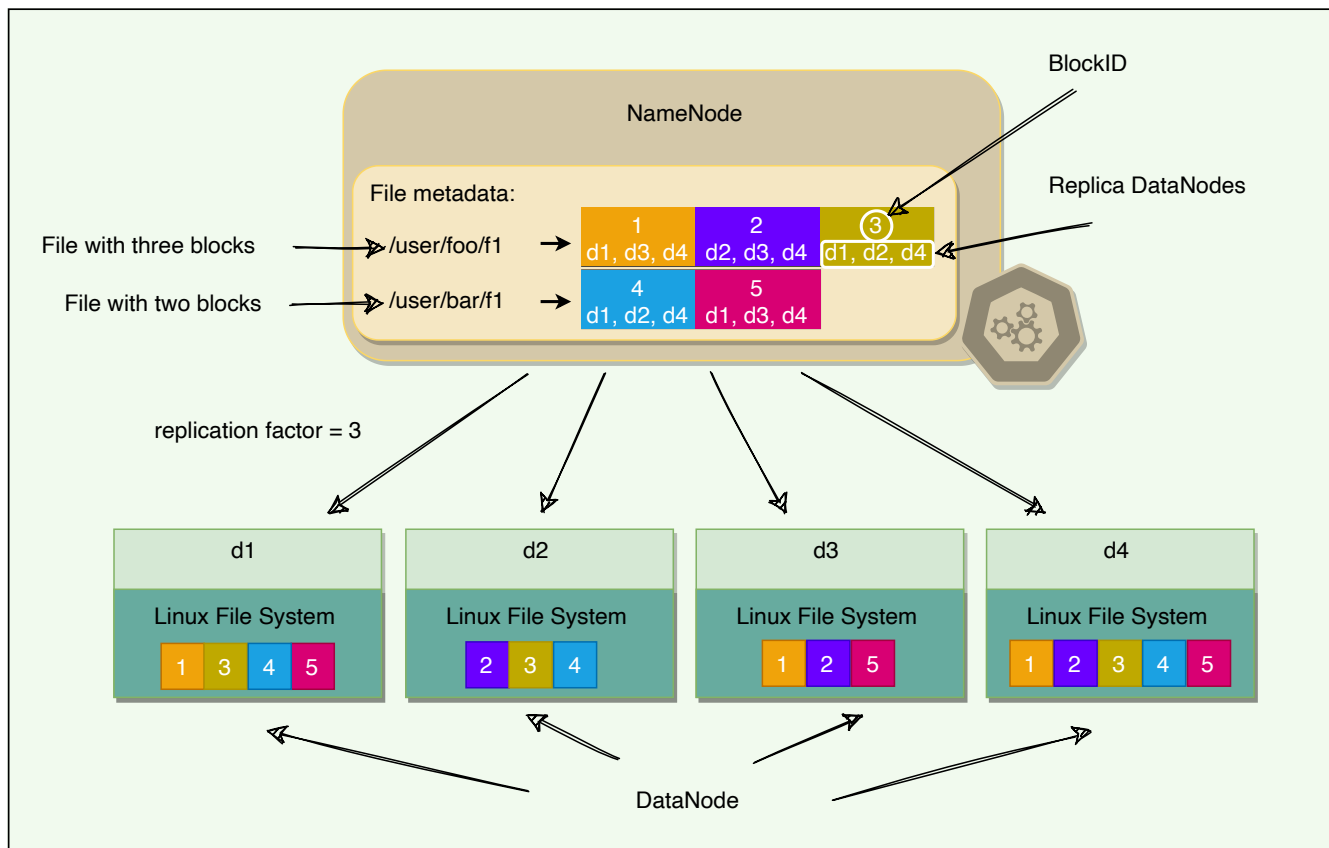
All files stored in HDFS are broken into multiple fixed-size blocks, where each block is 128 megabytes in size by default (configurable on a per-file basis). Each file stored in HDFS consists of two parts: the **actual file data** and the **metadata**, i.e., how many block parts the file has, their locations and the total file size, etc. HDFS cluster primarily consists of a **NameNode** that manages the file system metadata and **DataNodes** that store the actual data.



HDFS high-level architecture

- All blocks of a file are of the same size except the last one.
- HDFS uses **large block sizes** because it is designed to store extremely large files to enable MapReduce jobs to process them efficiently.
- Each block is identified by a unique 64-bit ID called **BlockID**.
- All read/write operations in HDFS operate at the block level.
- DataNodes store each block in a separate file on the local file system and provide read/write access.
- When a DataNode starts up, it scans through its local file system and sends the list of hosted data blocks (called BlockReport) to the NameNode.

- The NameNode maintains two on-disk data structures to store the file system's state: an **FsImage** file and an **EditLog**. FsImage is a checkpoint of the file system metadata at some point in time, while the EditLog is a log of all of the file system metadata transactions since the image file was last created. These two files help NameNode to recover from failure.
- User applications interact with HDFS through its client. HDFS Client interacts with NameNode for metadata, but all data transfers happen directly between the client and DataNodes.
- To achieve high-availability, HDFS creates multiple copies of the data and distributes them on nodes throughout the cluster.



HDFS block replication

Comparison between GFS and HDFS

#

HDFS architecture is similar to GFS, although there are differences in the terminology. Here is the comparison between the two file systems:

	GFS	HDFS
Storage node	ChunkServer	DataNode
File part	Chunk	Block
File part size	Default chunk size is 64MB (adjustable)	Default block size is 128MB (adjustable)
Metadata Checkpoint	Checkpoint image	FsImage
Write ahead log	Operation log	EditLog
Platform	Linux	Cross-Platform
Language	Developed in C++	Developed in Java
Available Implementation	Only used internally by Google	Opensource
Monitoring	Master receives HeartBeat from ChunkServers	NameNode receives HeartBeat from DataNodes
Concurrency	Follows multiple writers and multiple readers model.	Does not support multiple writ- ers. HDFS follows the write- once and read-many model.

File operations	Append and Random writes are possible.	Only append is possible.
Garbage Collection	Any deleted file is renamed into a particular folder to be garbage collected later.	Any deleted file is renamed to a hidden name to be garbage collected later.
Communication	RPC over TCP is used for communication with the master. To minimize latency, pipelining and streaming are used over TCP for data transfer.	RPC over TCP is used for communication with the NameNode. For data transfer, pipelining and streaming are used over TCP.
Cache Management	Clients cache metadata. Client or ChunkServer does not cache file data. ChunkServers rely on the buffer cache in Linux to maintain frequently accessed data in memory.	HDFS uses distributed cache. User-specified paths are cached explicitly in the DataNode's memory in an off-heap block cache. The cache could be private (belonging to one user) or public (belonging to all the users of the same node).

Replication Strategy	Chunk replicas are spread across the racks. Master automatically replicates the chunks. By default, three copies of each chunk are stored. User can specify a different replication factor. The master re-replicates a chunk replica as soon as the number of available replicas falls below a user-specified number.	The HDFS has an automatic rack-aware replication system. By default, two copies of each block are stored at two different DataNodes in the same rack, and a third copy is stored on a Data Node in a different rack (for better reliability). User can specify a different replication factor.
File system Namespace	Files are organized hierarchically in directories and identified by pathnames.	HDFS supports a traditional hierarchical file organization. Users or applications can create directories to store files inside. HDFS also supports third-party file systems such as Amazon Simple Storage Service (S3) and Cloud Store.
Database	Bigtable uses GFS as its storage engine.	HBase uses HDFS as its storage engine.

[< Back](#)

[Next >](#)

Deep Dive

Let's explore some of HDFS's design components.

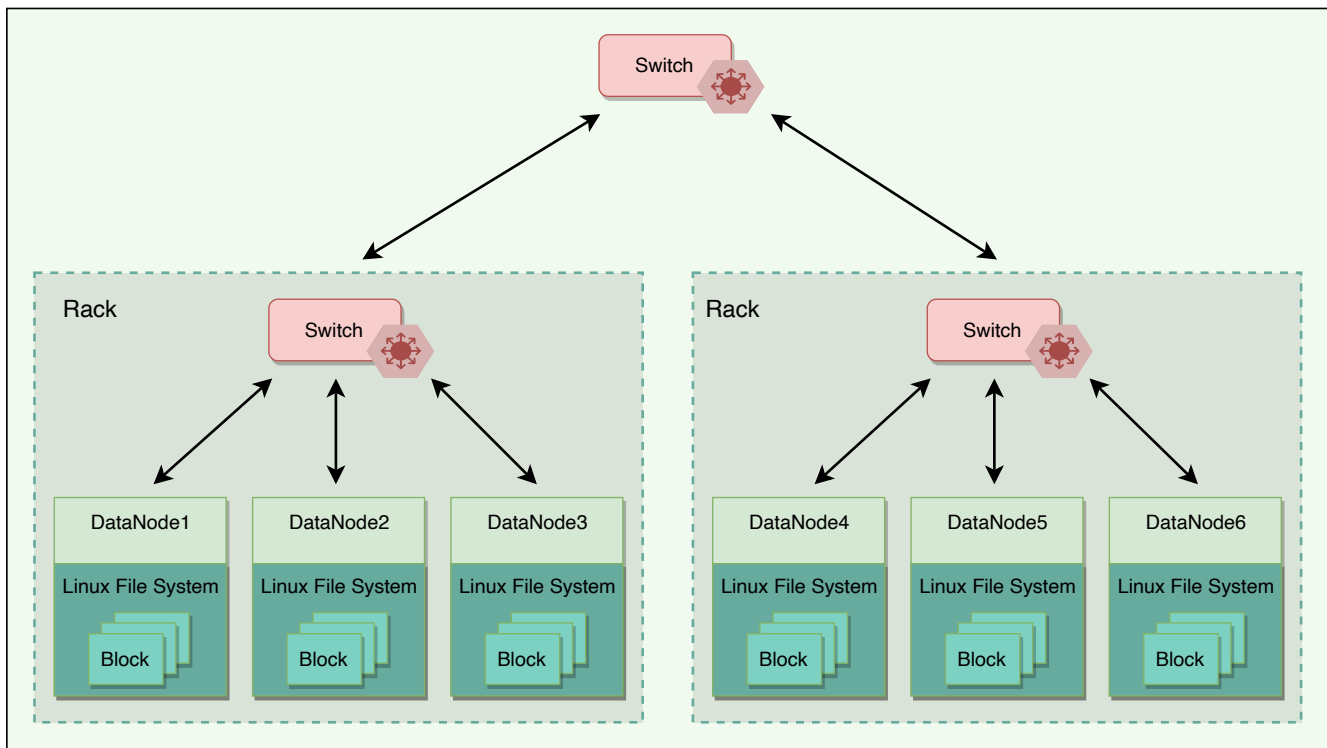
We'll cover the following



- Cluster topology
- Rack aware replication
- Synchronization semantics
- HDFS consistency model

Cluster topology

A typical data center contains many racks of servers connected using switches. A common configuration for Hadoop clusters is to have about 30 to 40 servers per rack. Each rack has a dedicated gigabit switch that connects all of its servers and an uplink to a core switch or router, whose bandwidth is shared by many racks in the data center, as shown in the following figure.



HDFS cluster topology

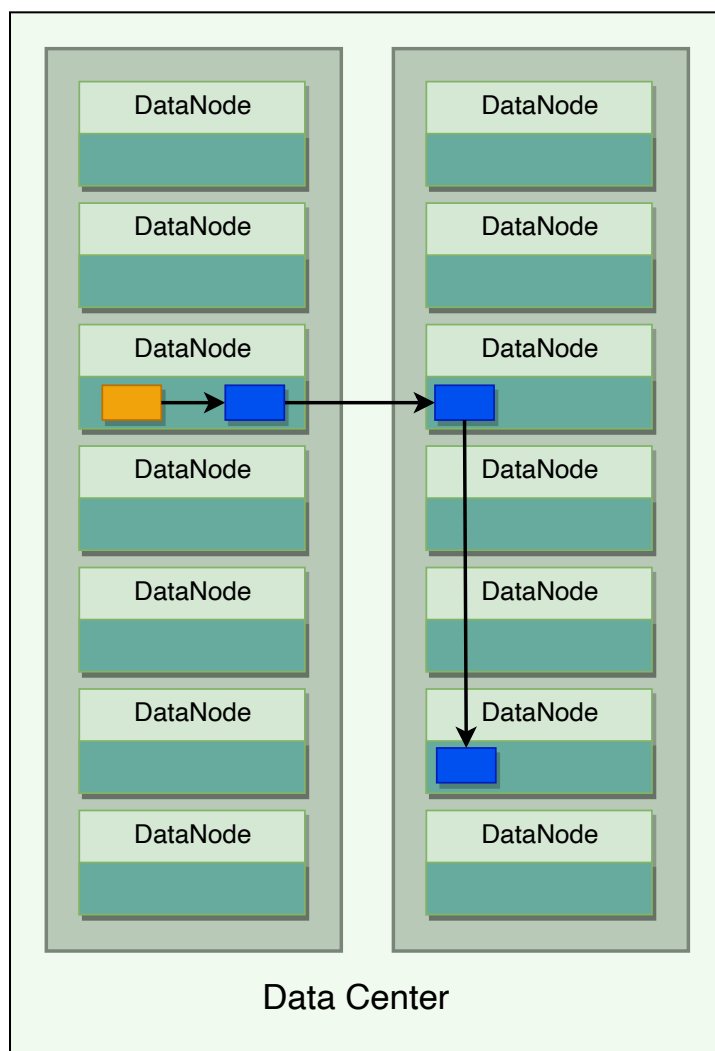
When HDFS is deployed on a cluster, each of its servers is configured and mapped to a particular rack. The network distance between servers is measured in hops, where one hop corresponds to one link in the topology. Hadoop assumes a tree-style topology, and the distance between two servers is the sum of their distances to their closest common ancestor.

In the above figure, the distance between Node 1 and itself is zero hops (the case when two processes are communicating on the same node). Node 1 and Node 2 are two hops away, while the distance between Node 3 and Node 4 is four hops.

Rack aware replication

The placement of replicas is critical to HDFS reliability and performance. HDFS employs a rack-aware replica placement policy to improve data reliability, availability, and network bandwidth utilization. If the replication factor is three, HDFS attempts to place the first replica on the same node as

the client writing the block. In case a client process is not running in the HDFS cluster, a node is chosen at random. The second replica is written to a node on a different rack from the first (i.e., off-rack replica). The third replica of the block is then written to another random node on the same rack as the second. Additional replicas are written to random nodes in the cluster, but the system tries to avoid placing too many replicas on the same rack. The figure below illustrates the replica placement for a triple-replicated block in HDFS. The idea behind HDFS's replica placement is to be able to tolerate node and rack failures. For example, when an entire rack goes offline due to power or networking problems, the requested block can still be located at a different rack.



Rack-aware replication

The default HDFS replica placement policy can be summarized as follows:

1. No DataNode will contain more than one replica of any block.
2. If there are enough racks available, no rack will contain more than two replicas of the same block.

Following this rack-aware replication scheme slows the write operation as the data needs to be replicated onto different racks, but this is an intentional tradeoff between reliability and performance that HDFS made.

Synchronization semantics

Early versions of HDFS followed strict **immutable semantics**. Once a file was written, it could never again be re-opened for writes; files could still be deleted. However, current versions of HDFS support append. This is still quite limited in the sense that existing binary data once written to HDFS cannot be modified in place.

This design choice in HDFS was made because some of the most common MapReduce workloads follow the **write once, read many data-access** pattern. MapReduce is a restricted computational model with predefined stages. The reducers in MapReduce write independent files to HDFS as output. HDFS focuses on fast read access for multiple clients at a time.

HDFS consistency model

HDFS follows a **strong consistency model**. As stated above, each data block written to HDFS is replicated to multiple nodes. To ensure strong consistency, a write is declared successful only when all replicas have been written

successfully. This way, all clients see the same (and consistent) view of the file. Since HDFS does not allow multiple concurrent writers to write to an HDFS file, implementing strong consistency becomes a relatively easy task.

[!\[\]\(d0a1791f26d167e866e44ebbf83efebe_img.jpg\) Back](#)

High-level Architecture

[Next !\[\]\(eafc244b53721dd1ec133f0772f70fc7_img.jpg\)](#)

Anatomy of a Read Operation

Anatomy of a Read Operation

We'll cover the following



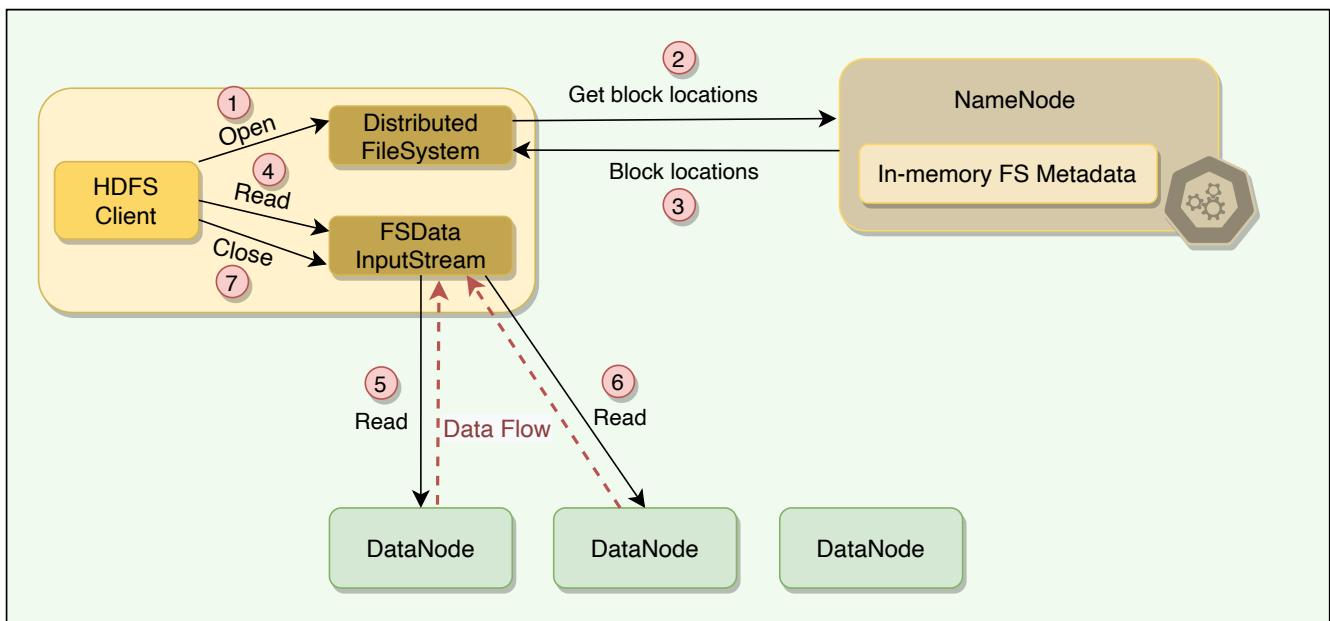
- HDFS read process
- Short circuit read

HDFS read process

HDFS read process can be outlined as follows:

1. When a file is opened for reading, HDFS client initiates a read request, by calling the `open()` method of the `Distributed FileSystem` object. The client specifies the file name, start offset, and the read range length.
2. The `Distributed FileSystem` object calculates what blocks need to be read based on the given offset and range length, and requests the locations of the blocks from the `NameNode`.
3. `NameNode` has metadata for all blocks' locations. It provides the client a list of blocks and the locations of each block replica. As the blocks are replicated, `NameNode` finds the closest replica to the client when providing a particular block's location. The closest locality of each block is determined as follows:
 - If a required block is within the same node as the client, it is preferred.
 - Then, the block in the same rack as the client is preferred.
 - Finally, an off-rack block is read.

4. After getting the block locations, the client calls the `read()` method of `FSDa InputStream`, which takes care of all the interactions with the `DataNodes`. In step 4 in the below diagram, once the client invokes the `read()` method, the input stream object establishes a connection with the closest `DataNode` with the first block of the file.
5. The data is read in the form of streams. As the data is streamed, it is passed to the requesting application. Hence, the block does not have to be transferred in its entirety before the client application starts processing it.
6. Once the `FSDa InputStream` receives all data of a block, it closes the connection and moves on to connect the `DataNode` for the next block. It repeats this process until it finishes reading all the required blocks of the file.
7. Once the client finishes reading all the required blocks, it calls the `close()` method of the input stream object.



The anatomy of a read operation

Short circuit read

As we saw above, the client reads the data directly from DataNode. The client uses TCP sockets for this. If the data and the client are on the same machine, HDFS can directly read the file bypassing the DataNode. This scheme is called short circuit read and is quite efficient as it reduces overhead and other processing resources.

[< Back](#)[Next >](#)

Deep Dive

Anatomy of a Write Operation

Anatomy of a Write Operation

This lesson explains how HDFS handles a write operation.

We'll cover the following



- HDFS write process

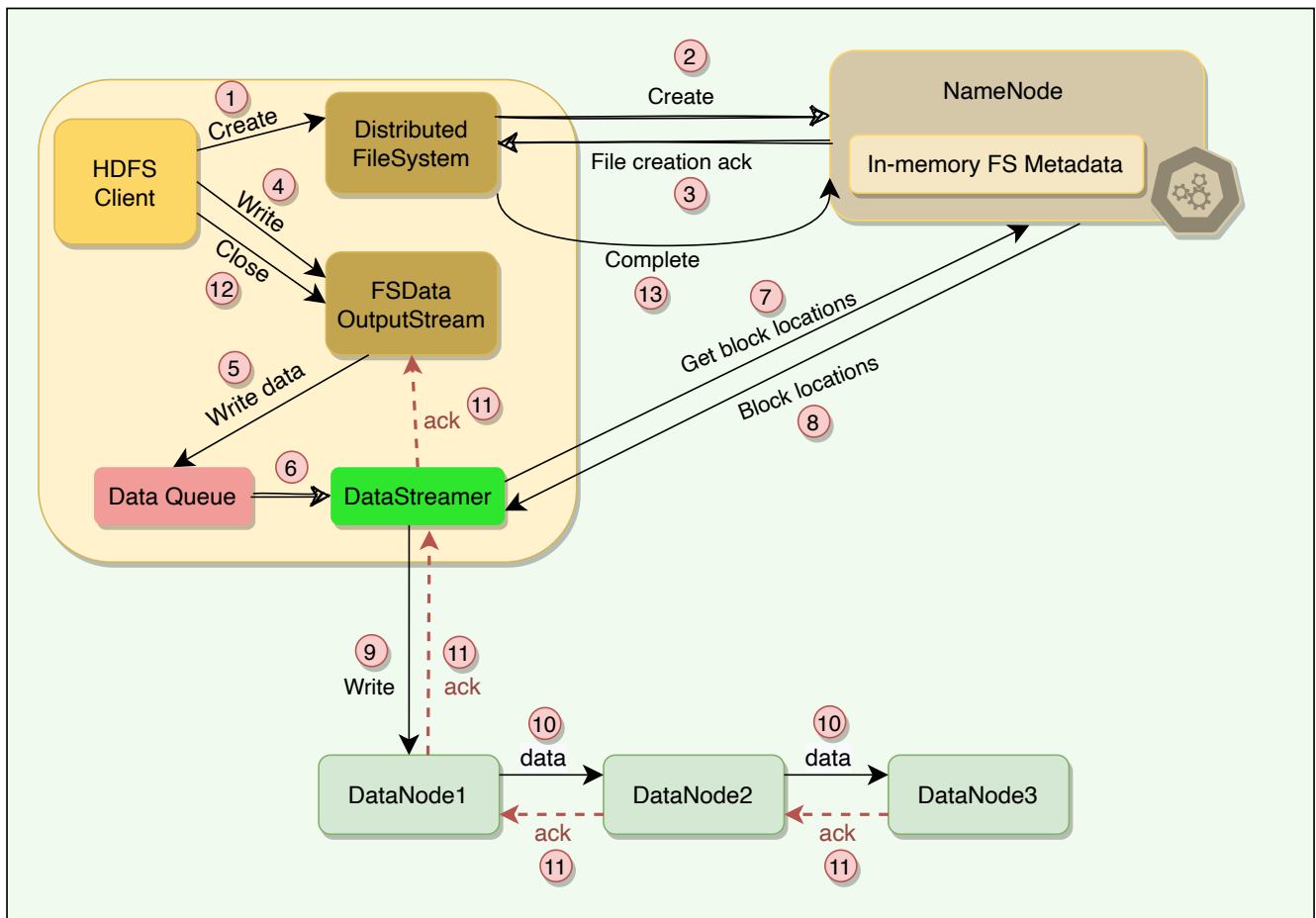
HDFS write process

HDFS write process can be outlined as follows:

1. HDFS client initiates a write request by calling the `create()` method of the `Distributed FileSystem` object.
2. The `Distributed FileSystem` object sends a file creation request to the `NameNode`.
3. The `NameNode` verifies that the file does not already exist and that the client has permission to create the file. If both these conditions are verified, the `NameNode` creates a new file record and sends an acknowledgment.
4. The client then proceeds to write the file using `FSDa` `OutputStream`
5. The `FSDa OutputStream` writes data to a local queue called 'Data Queue.' The data is kept in the queue until a complete block of data is accumulated.
6. Once the queue has a complete block, another component called `DataStream` is notified to manage data transfer to the `DataNode`.
7. `DataStream` first asks the `NameNode` to allocate a new block on `DataNodes`, thereby picking desirable `DataNodes` to be used for

replication.

8. The NameNode provides a list of blocks and the locations of each block replica.
9. Upon receiving the block locations from the NameNode, the `DataStream` starts transferring the blocks from the internal queue to the nearest `DataNode`.
10. Each block is written to the first `DataNode`, which then pipelines the block to other `DataNodes` in order to write replicas of the block. This way, the blocks are replicated during the file write itself. It is important to note that HDFS does not acknowledge a write to the client until all the replicas for that block have been written by the `DataNodes`.
11. Once the `DataStream` finishes writing all blocks, it waits for acknowledgments from all the `DataNodes`.
12. Once all acknowledgments are received, the client calls the `close()` method of the `OutputStream`.
13. Finally, the `Distributed FileSystem` contacts the NameNode to notify that the file write operation is complete. At this point, the NameNode commits the file creation operation, which makes the file available to be read. If the NameNode dies before this step, the file is lost.



← Back

Next →

Anatomy of a Read Operation

Data Integrity & Caching

Data Integrity & Caching

Let's explore how HDFS ensures data integrity and implements caching.

We'll cover the following



- Data integrity
 - Block scanner
- Caching

Data integrity

Data Integrity refers to ensuring the correctness of the data. When a client retrieves a block from a DataNode, the data may arrive corrupted. This corruption can occur because of faults in the storage device, network, or the software itself. HDFS client uses checksum to verify the file contents. When a client stores a file in HDFS, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents, it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another replica.

Block scanner

A block scanner process periodically runs on each DataNode to scan blocks stored on that DataNode and verify that the stored checksums match the block data. Additionally, when a client reads a complete block and checksum

verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica. Whenever a client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt and initiates the process to create a new good replica of the block.

Caching

Normally, blocks are read from the disk, but for frequently accessed files, blocks may be explicitly cached in the DataNode's memory, in an off-heap block cache. HDFS offers a Centralized Cache Management scheme to allow its users to specify paths to be cached. Clients can tell the NameNode which files to cache. NameNode communicates with the DataNodes that have the desired blocks on disk and instructs them to cache the blocks in off-heap caches.

Centralized cache management in HDFS has many significant advantages:

1. Explicitly specifying blocks for caching prevents the eviction of frequently accessed data from memory. This is particularly important as most of the HDFS workloads are bigger than the main memory of the DataNode.
2. Because the NameNode manages DataNode caches, applications can query the set of cached block locations when making MapReduce task placement decisions. Co-locating a task with a cached block replica improves read performance.
3. When a DataNode has cached a block, clients can use a new, more efficient, zero-copy read API. As the block is already in memory and its checksum verification has already been done by the DataNode, clients can incur essentially zero overhead when using this new API.
4. Centralized caching can improve overall cluster memory utilization. When relying on the OS buffer cache at each DataNode, repeated reads of a block will result in all 'n' replicas of the block being pulled into the

buffer cache. With centralized cache management, a user can explicitly specify only 'm' of the 'n' replicas, saving 'n-m' memory.

← Back

Anatomy of a Write Operation

Next →

Fault Tolerance

Fault Tolerance

Let's explore what techniques HDFS uses for fault tolerance.

We'll cover the following



- How does HDFS handle DataNode failures?
 - Replication
 - HeartBeat
- What happens when the NameNode fails?
 - FsImage and EditLog
 - Metadata backup

How does HDFS handle DataNode failures?

Replication

When a DataNode dies, all of its data becomes unavailable. HDFS handles this data unavailability through replication. As stated earlier, every block written to HDFS is replicated to multiple (default three) DataNodes.

Therefore, if one DataNode becomes inaccessible, its data can be read from other replicas.

HeartBeat

The NameNode keeps track of DataNodes through a heartbeat mechanism. Each DataNode sends periodic heartbeat messages (every few seconds) to the NameNode. If a DataNode dies, the heartbeats will stop, and the NameNode will detect that the DataNode has died. The NameNode will then mark the DataNode as dead and will no longer forward any read/write request to that DataNode. Because of replication, the blocks stored on that DataNode have additional replicas on other DataNodes. The NameNode performs regular status checks on the file system to discover under-replicated blocks and performs a **cluster rebalance** process to replicate blocks that have less than the desired number of replicas.

What happens when the NameNode fails?

FsImage and EditLog

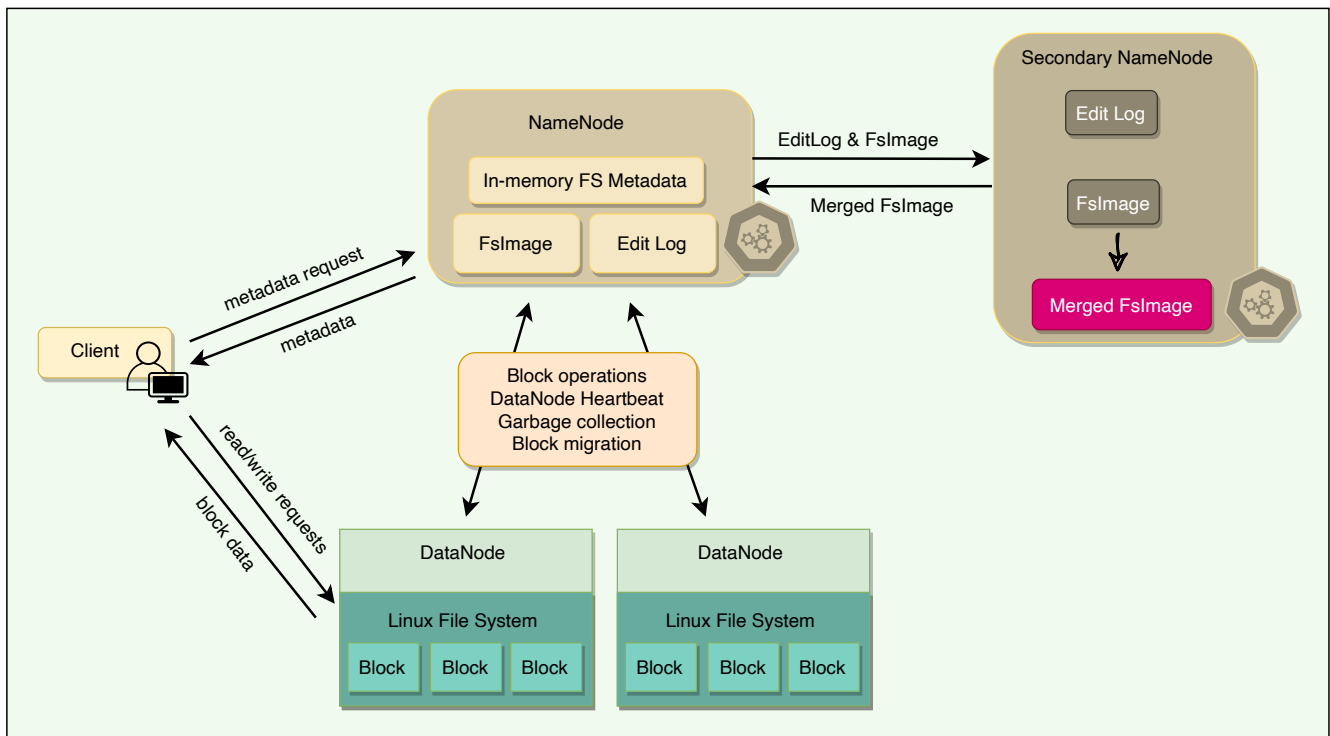
The NameNode is a **single point of failure (SPOF)**. A NameNode failure will bring the entire file system down. Internally, the NameNode maintains two on-disk data structures that store the file system's state: an **FsImage** file and an **EditLog**. FsImage is a checkpoint (or the image) of the file system metadata at some point in time, while the EditLog is a log of all of the file system metadata transactions since the image file was last created. All incoming changes to the file system metadata are written to the EditLog. At periodic intervals, the EditLog and FsImage files are merged to create a new image file snapshot, and the edit log is cleared out.

Metadata backup

On a NameNode failure, the metadata would be unavailable, and a disk failure on the NameNode would be catastrophic because the file metadata would be lost since there would be no way of knowing how to reconstruct

the files from the blocks on the DataNodes. For this reason, it is crucial to make the NameNode resilient to failure, and HDFS provides two mechanisms for this:

1. The first way is to back up and store multiple copies of FsImage and EditLog. The NameNode can be configured to maintain multiple copies of the files. Any update to either the FsImage or EditLog causes each copy of the FsImages and EditLogs to get updated synchronously and atomically. A common configuration is to maintain one copy of these files on a local disk and one on a remote Network File System (NFS) mount. This synchronous updating of multiple copies of the FsImage and EditLog may degrade the rate of namespace transactions per second that a NameNode can support. However, this degradation is acceptable because even though HDFS applications are very data-intensive, they are not metadata-intensive.
2. Another option provided by HDFS is to run a **Secondary NameNode**, which despite its name, is not a backup NameNode. Its main role is to help primary NameNode in taking the checkpoint of the filesystem. Secondary NameNode periodically merges the namespace image with the EditLog to prevent the EditLog from becoming too large. The secondary NameNode runs on a separate physical machine because it requires plenty of CPU and as much memory as the NameNode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the NameNode failure. However, the state of the secondary NameNode lags behind that of the primary, so in the event of total failure of the primary, data loss is almost inevitable. The usual course of action, in this case, is to copy the NameNode's metadata files that are on NFS to the secondary and run it as the new primary.



Role of primary and secondary NameNode

← Back

Next →

Data Integrity & Caching

HDFS High Availability (HA)

HDFS High Availability (HA)

Let's learn how HDFS achieves high availability.

We'll cover the following



- HDFS high availability architecture
 - QJM
 - Zookeeper
- Failover and fencing
 - Fencing

HDFS high availability architecture

Although NameNode's metadata is copied to multiple file systems to protect against data loss, it still does not provide high availability of the filesystem. If the NameNode fails, no clients will be able to read, write, or list files, because the NameNode is the sole repository of the metadata and the file-to-block mapping. In such an event, the whole Hadoop system would effectively be out of service until a new NameNode is brought online.

To recover from a failed NameNode scenario, an administrator will start a new primary NameNode with one of the filesystem metadata replicas and configure DataNodes and clients to use this new NameNode. The new NameNode is not able to serve requests until it has

1. loaded its namespace image into memory,
2. replayed its EditLog, and
3. received enough block reports from the DataNodes.

On large clusters with many files and blocks, it can take half an hour or more to perform a cold start of a NameNode. Furthermore, this long recovery time is a problem for routine maintenance. In fact, because an unexpected failure of the NameNode is rare, the case for planned downtime is actually more important in practice.

To solve this problem, Hadoop, in its 2.0 release, added support for HDFS High Availability (HA). In this implementation, there are two (or more) NameNodes in an active-standby configuration. At any point in time, exactly one of the NameNodes is in an active state, and the others are in a Standby state. The active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a follower of the active, maintaining enough state to provide a fast failover when required.

For the Standby nodes to keep their state synchronized with the active node, HDFS made a few architectural changes:

- The NameNodes must use highly available shared storage to share the EditLog (e.g., a Network File System (NFS) mount from a Network Attached Storage (NAS)).
- When a standby NameNode starts, it reads up to the end of the shared EditLog to synchronize its state with the active NameNode, and then continues to read new entries as the active NameNode writes them.
- DataNodes must send block reports to all the NameNodes because the block mappings are stored in a NameNode's memory, and not on disk.
- Clients must be configured to handle NameNode failover, using a mechanism that is transparent to users. Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to multiple NameNode addresses, and the client library tries each NameNode address until the operation succeeds.

There are two choices for the highly available shared storage: an NFS filer (as described above), or a **Quorum Journal Manager (QJM)**.

QJM

The sole purpose of the QJM is to provide a highly available EditLog. The QJM runs as a group of journal nodes, and each edit must be written to a quorum (or majority) of the journal nodes. Typically, there are three journal nodes, so that the system can tolerate the loss of one of them. This arrangement is similar to the way ZooKeeper (<https://zookeeper.apache.org/>) works, although it is important to realize that the QJM implementation does not use ZooKeeper.



Note: HDFS High Availability does use ZooKeeper for electing the active NameNode. More details on this later. QJM process runs on all NameNodes and communicates all EditLog changes to journal nodes using RPC.

Since the Standby NameNodes have the latest state of the metadata available in memory (both the latest EditLog and an up-to-date block mapping), any standby can take over very quickly (in a few seconds) if the active NameNode fails. However, the actual failover time will be longer in practice (around a minute or so) because the system needs to be conservative in deciding that the active NameNode has failed.

In the unlikely event of the Standbys being down when the active fails, the administrator can still do a cold start of a Standby. This is no worse than the non-HA case.

Zookeeper

ZooKeeper Failover Controller (ZKFC)
monitors the health of NN, OS, and HW



ZKFC also coordinates the NN election process

Block report to standby NNs

HDFS high availability architecture

275

A **Failover Controller** manages the transition from the active NameNode to the Standby. The default implementation of the failover controller uses **ZooKeeper** to ensure that only one NameNode is active. Failover Controller runs as a lightweight process on each NameNode and monitors the NameNode for failures (using Heartbeat), and triggers a failover when the active NameNode fails.

Graceful failover: For routine maintenance, an administrator can manually initiate a failover. This is known as a graceful failover, since the failover controller arranges an orderly transition from the active NameNode to the Standby.

Ungraceful failover: In the case of an ungraceful failover, however, it is impossible to be sure that the failed NameNode has stopped running. For example, a slow network or a network partition can trigger a failover transition, even though the previously active NameNode is still running and thinks it is still the active NameNode.

The HA implementation uses the mechanism of **Fencing** to prevent this “**split-brain**” scenario and ensure that the previously active NameNode is prevented from doing any damage and causing corruption.

Fencing

Fencing is the idea of putting a fence around a previously active NameNode so that it cannot access cluster resources and hence stop serving any read/write request. To apply fencing, the following two techniques are used:

- **Resource fencing:** Under this scheme, the previously active NameNode is blocked from accessing resources needed to perform essential tasks. For example, revoking its access to the shared storage directory (typically by using a vendor-specific NFS command), or disabling its network port via a remote management command.

- **Node fencing:** Under this scheme, the previously active NameNode is blocked from accessing all resources. A common way of doing this is to power off or reset the node. This is an effective method of keeping it from accessing anything at all. This technique is also called STONIT or “Shoot The Other Node In The Head.”

To learn more about automatic failover, take a look at Apache documentation (https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html#Automatic_Failover).

← Back

Fault Tolerance

Next →

HDFS Characteristics

HDFS Characteristics

This lesson will explore some important aspects of HDFS architecture.

We'll cover the following



- Security and permission
- HDFS federation
- Erasure coding
- HDFS in practice

Security and permission

HDFS provides a permissions model for files and directories which is similar to POSIX. Each file and directory is associated with an **owner** and a **group**. Each file or directory has separate permissions for the owner, other users who are members of a group, and all other users. There are three types of permission:

1. **Read permission (r)**: For files, *r* permission is required to read a file. For directories, *r* permission is required to list the contents of a directory.
2. **Write permission (w)**: For files, *w* permission is required to write or append to a file. For a directory, *w* permission is required to create or delete files or directories in it.
3. **Execute permission (x)**: For files, *x* permission is ignored as we cannot execute a file on HDFS. For a directory, *x* permission is required

to access a child of the directory.

HDFS also provides optional support for POSIX ACLs (Access Control Lists) to augment file permissions with finer-grained rules for specific named users or named groups.

HDFS federation

The NameNode keeps the metadata of the whole namespace in memory, which means that on very large clusters with many files, the memory becomes the limiting factor for scaling. A more serious problem is that a single NameNode, serving all metadata requests, can become a performance bottleneck. To help resolve these issues, HDFS Federation was introduced in the 2.x release, which allows a cluster to scale by adding NameNodes, each of which manages a portion of the filesystem namespace. For example, one NameNode might manage all the files rooted under `/user`, and a second NameNode might handle files under `/share`. Under federation:

- All NameNodes work independently. No coordination is required between NameNodes.
- DataNodes are used as the common storage by all the NameNodes.
- A NameNode failure does not affect the availability of the namespaces managed by other NameNodes.
- To access a federated HDFS cluster, clients use client-side mount tables to map file paths to NameNodes.

Multiple NameNodes running independently can end up generating the same **64-bit Block IDs** for their blocks. To avoid this problem, a namespace uses one or more **Block Pools**, where a unique ID identifies each block pool in a cluster. A block pool belongs to a single namespace and does not cross the namespace boundary. The extended block ID, which is a tuple of (Block Pool ID, Block ID), is used for block identification in HDFS Federation.

Erasure coding

By default, HDFS stores three copies of each block, resulting in a 200% overhead (to store two extra copies) in storage space and other resources (e.g., network bandwidth). Compared to this default replication scheme, Erasure Coding (EC) is probably the biggest change in HDFS in recent years. EC provides the same level of fault tolerance with much less storage space. In a typical EC setup, the storage overhead is no more than 50%. This fundamentally doubles the storage space capacity by bringing down the replication factor from 3x to 1.5x.

Under EC, data is broken down into fragments, expanded, encoded with redundant data pieces, and stored across different DataNodes. If, at some point, data is lost on a DataNode due to corruption, etc., then it can be reconstructed using the other fragments stored on other DataNodes. Although EC is more CPU intensive, it greatly reduces the storage needed for reliably storing a large data set.

For more details on how EC works, see blog (<https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/>) or wiki (https://en.wikipedia.org/wiki/Erasure_code).

HDFS in practice

Although HDFS was primarily designed to support Hadoop MapReduce jobs by providing a DFS for the Map and Reduce operations, HDFS has found many uses with big-data tools.

HDFS is used in several Apache projects that are built on top of the Hadoop framework, including Pig, Hive, HBase, and Giraph. HDFS support is also included in other projects, such as GraphLab.

The primary advantages of HDFS include the following:

- **High bandwidth for MapReduce workloads:** Large Hadoop clusters (thousands of machines) are known to continuously write up to one terabyte per second using HDFS.
- **High reliability:** Fault tolerance is a primary design goal in HDFS. HDFS replication provides high reliability and availability, particularly in large clusters, in which the probability of disk and server failures increases significantly.
- **Low costs per byte:** Compared to a dedicated, shared-disk solution such as a SAN, HDFS costs less per gigabyte because storage is collocated with compute servers. With SAN, we have to pay additional costs for managed infrastructure, such as the disk array enclosure and higher-grade enterprise disks, to manage hardware failures. HDFS is designed to run with commodity hardware, and redundancy is managed in software to tolerate failures.
- **Scalability:** HDFS allows DataNodes to be added to a running cluster and offers tools to manually rebalance the data blocks when cluster nodes are added, which can be done without shutting the file system down.

The primary disadvantages of HDFS include the following:

- **Small file inefficiencies:** HDFS is designed to be used with large block sizes (128MB and larger). It is meant to take large files (hundreds of megabytes, gigabytes, or terabytes) and chunk them into blocks, which can then be fed into MapReduce jobs for parallel processing. HDFS is inefficient when the actual file sizes are small (in the kilobyte range). Having a large number of small files places additional stress on the NameNode, which has to maintain metadata for all the files in the file system. Typically, HDFS users combine many small files into larger ones using techniques such as sequence files. A sequence file can be

understood as a container of binary key-value pairs, where the file name is the key, and the file contents are the value.

- **POSIX non-compliance:** HDFS was not designed to be a POSIX-compliant, mountable file system; applications will have to be either written from scratch or modified to use an HDFS client. Workarounds exist that enable HDFS to be mounted using a FUSE (https://en.wikipedia.org/wiki/Filesystem_in_Userspace) driver, but the file system semantics do not allow writes to files once they have been closed.
- **Write-once model:** The write-once model is a potential drawback for applications that require concurrent write accesses to the same file. However, the latest version of HDFS now supports file appends.

In short, HDFS is a good option as a storage backend for distributed applications that follow the MapReduce model or have been specifically written to use HDFS. HDFS can be used efficiently with a small number of large files rather than a large number of small files.

← Back

Next →

HDFS High Availability (HA)

Summary: HDFS

Summary: HDFS

Here is a quick summary of HDFS for you!

We'll cover the following



- Summary
- System design patterns
- References and further reading

Summary

- HDFS is a scalable distributed file system for large, distributed data-intensive applications.
- HDFS uses commodity hardware to reduce infrastructure costs.
- HDFS provides APIs for usual file operations like `create`, `delete`, `open`, `close`, `read`, and `write`.
- Random writes are not possible; writes are always made at the end of the file in an append-only fashion.
- HDFS does not support multiple concurrent writers.
- An HDFS cluster consists of a **single NameNode** and **multiple DataNodes** and is accessed by multiple clients.
- **Block:** Files are broken into fixed-size blocks (default 128MB), and blocks are replicated across a number of DataNodes to ensure fault-tolerance. The block size and the replication factor are configurable.
- DataNodes store blocks on local disk as Linux files.

- NameNode server is the coordinator of an HDFS cluster and is responsible for keeping track of all filesystem metadata.
- NameNode keeps all metadata in memory for faster operations. For fault-tolerance and in the event of NameNode crash, all metadata changes are written to the disk onto an **EditLog**. This EditLog can also be replicated on a remote filesystem (e.g., NFS) or a secondary NameNode.
- The NameNode does not keep a persistent record of which DataNodes have a replica of a given block. Instead, the NameNode asks each DataNode about what blocks it holds at NameNode startup and whenever a DataNode joins the cluster.
- **FsImage**: The NameNode state is periodically serialized to disk and then replicated, so that on recovery, a NameNode may load the checkpoint into memory, replay any subsequent operations from the edit log, and be available again very quickly.
- **HeartBeat**: The NameNode communicates with each DataNode through Heartbeat messages to pass instructions and collect its state.
- **Client**: User applications interact with HDFS through its client. HDFS Client interacts with NameNode for metadata, but all data transfers happen directly between the client and DataNodes.
- **Data Integrity**: Each DataNode uses checksumming to detect the corruption of stored data.
- **Garbage Collection**: Any deleted file is renamed to a hidden name to be garbage collected later.
- **Consistency**: HDFS is a strongly consistent file system. Each data block is replicated to multiple nodes, and a write is declared to be successful only after all the replicas have been written successfully.
- **Cache**: For frequently accessed files, the blocks may be explicitly cached in the DataNode's memory, in an off-heap block cache.
- **Erasure coding**: HDFS uses erasure coding to reduce replication overhead.

System design patterns

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

- **Write-Ahead Log:** For fault tolerance and in the event of NameNode crash, all metadata changes are written to the disk onto an EditLog which is a write-ahead log.
- **HeartBeat:** The HDFS NameNode periodically communicates with each DataNode in HeartBeat messages to give it instructions and collect its state.
- **Split-Brain:** ZooKeeper is used to ensure that only one NameNode is active at any time. Fencing is used to put a fence around a previously active NameNode so that it cannot access cluster resources and hence stop serving any read/write request.
- **Checksum:** Each DataNode uses checksumming to detect the corruption of stored data.

References and further reading

- HDFS paper
(<https://storageconference.us/2010/Papers/MSST/Shvachko.pdf>)
- HDFS High Availability (HA) architecture
(<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html>)
- Apache HDFS Architecture
(<https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>)

- Distributed File Systems: A Survey

(<http://ijcsit.com/docs/Volume%205/vol5issue03/ijcsit20140503234.pdf>)

← Back

HDFS Characteristics

Next →

Quiz: HDFS

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

BigTable: Introduction

Let's explore Bigtable and its use cases.

We'll cover the following



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

Goal

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

What is BigTable?

BigTable is a **distributed** and **massively scalable** wide-column store. It is designed to store huge sets of structured data. As its name suggests, BigTable provides storage for very big tables (often in the terabyte range).

In terms of the CAP theorem, BigTable is a CP system, i.e., **it has strictly consistent reads and writes**. BigTable can be used as an input source or output destination for MapReduce

(https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html) jobs.

Background

BigTable was developed at Google and has been in use since 2005 in dozens of Google services. Because of the large scale of its services, Google could not use commercial databases. Also, the cost of using an external solution would have been too high. That is why Google chose to build an in-house solution. BigTable is a highly available and high-performing database that powers multiple applications across Google — where each application has different needs in terms of the size of data to be stored and latency with which results are expected.

Though BigTable is not open-source itself, its paper was crucial in inspiring powerful open-source databases like Cassandra (<https://cassandra.apache.org/>) (which borrows BigTable's data model), HBase (<https://hbase.apache.org/>) (a distributed non-relational database) and Hypertable (<https://hypertable.org/>).

BigTable use cases

Google built BigTable to store large amounts of data and perform thousands of queries per second on that data. Examples of BigTable data are billions of URLs with many versions per page, petabytes of Google Earth data, and billions of users' search data.

BigTable is suitable to store large datasets that are greater than one TB where each row is less than 10MB. Since BigTable does not provide ACID (atomicity, consistency, isolation, durability) properties or transaction support, Online Transaction Processing (OLTP (https://en.wikipedia.org/wiki/Online_transaction_processing)) applications

with transaction processes should not use BigTable. For BigTable, data should be structured in the form of key-value pairs or rows-columns. Non-structured data like images or movies should not be stored in BigTable.

Here are the examples of data that Google stores in BigTable:

- URL and its related data, e.g., PageRank, page contents, crawl metadata (e.g., when a page was crawled, what was the response code, etc.), links, anchors (links pointing to a page). There are billions of URLs with many versions of a page.
- Per-user data, e.g., preference settings, recent queries/search results. Google has hundreds of millions of users.

BigTable can be used to store the following types of data:

1. Time series data: As the data is naturally ordered
2. Internet of Things (IoT) data: Constant streams of writes
3. Financial Data: Often represented as time-series data

← Back

Mock Interview: HDFS

Next →

BigTable Data Model

BigTable Data Model

This lesson explains how BigTable models its data.

We'll cover the following



- Rows
- Column families
- Columns
- Timestamps

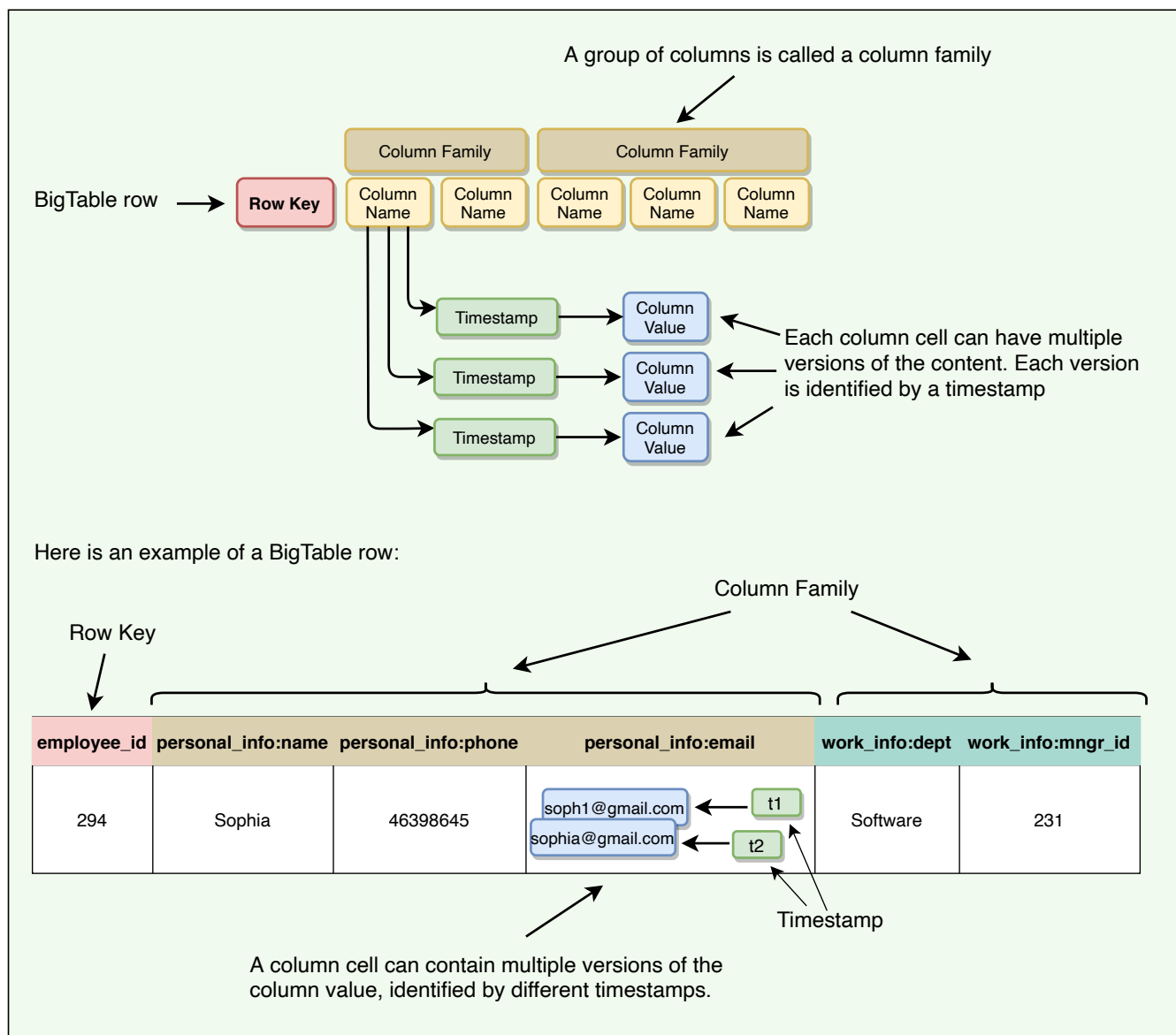
In simple terms, BigTable can be characterized as a sparse, distributed, persistent, multidimensional, sorted map. Let's dig deeper to understand each of these characteristics of BigTable.

Traditional DBs have a two-dimensional layout of the data, where each cell value is identified by the '**Row ID**' and '**Column Name**':

	employee_id	name	phone	email	department	manager_id
	294	Sophia	46398645	sophia@email.com	Software	231
	321	Xi Peng	null	xipen@email.com	Software	144
Row ID	321	Adam	87533210	null	HRD	57
	326	Rahul	23746398	rahul@email.com	HRD	57
	400	Peter	65289398	peter@email.com	Software	88

BigTable has a **four-dimensional data model**. The four dimensions are:

1. Row Key: Uniquely identifies a row
2. Column Family: Represents a group of columns
3. Column Name: Uniquely identifies a column
4. Timestamp: Each column cell can have different versions of a value, each identified by a timestamp



BigTable's four-dimensional data model

The data is indexed (or sorted) by row key, column key, and a timestamp. Therefore, to access a cell's contents, we need values for all of them. If no timestamp is specified, BigTable retrieves the most recent version.

```
( row_key : string, column_name : string, timestamp : int64 ) → cell  
contents (string)
```

Rows

Each row in the table has an associated row key that is an arbitrary string of up to 64 kilobytes in size (although most keys are significantly smaller):

- Each row is uniquely identified by the 'row key.'
- Each 'row key' is internally represented as a string.
- Every read or write of data under a single row is atomic. This also means that atomicity across rows is not guaranteed, e.g., when updating two rows, one might succeed, and the other might fail.
- Each table's data is only indexed by row key, column key, and timestamp. There are no secondary indices.

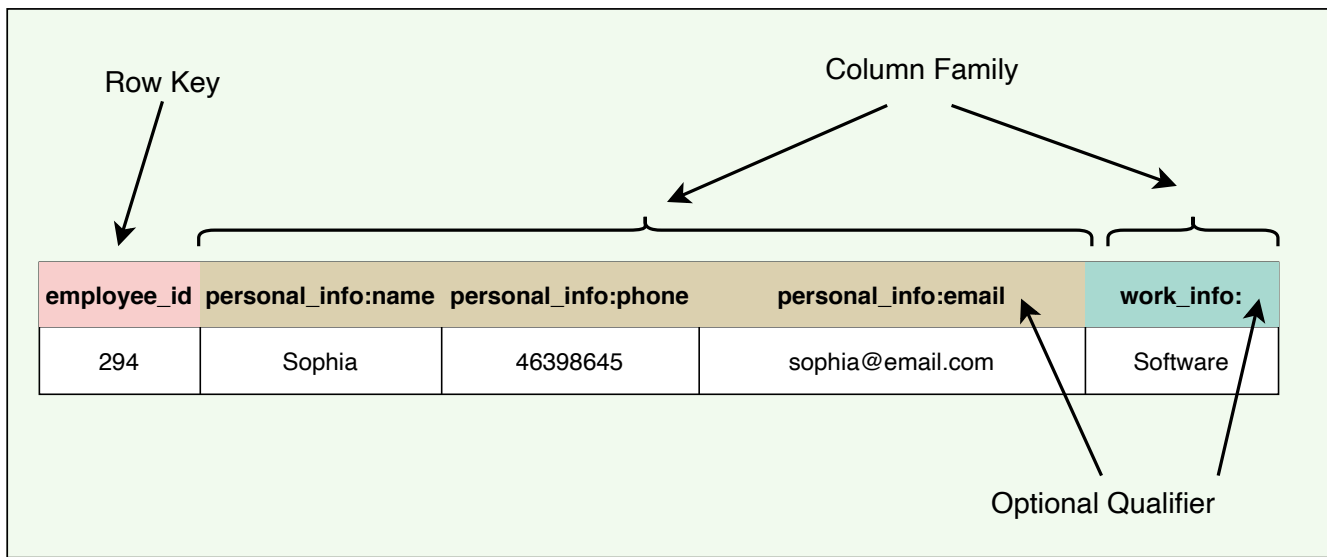
A **column** is a key-value pair where the key is represented as 'column key' and the value as 'column value.'

Column families

Column keys are grouped into sets called column families. All data stored in a column family is usually of the same type. The number of distinct column families in a table should be small (in the hundreds at maximum), and

families should rarely change during operation. Access control as well as both disk and memory accounting are performed at the column-family level.

The following figure shows a single row from a table. The row key is 294 , and there are two column families: `personal_info` and `work_info` , with three columns under the `personal_info` column family.



Column families

- Column family format: `family:optional qualifier`
- All rows have the same set of column families.
- BigTable can retrieve data from the same column family efficiently.
- Short Column family names are better as names are included in the data transfer.

Columns

- Columns are units within a column family.
- A BigTable may have an unbounded number of columns.
- New columns can be added on the fly.

- Short column names are better as names are passed in each data transfer, e.g., `ColumnFamily:ColumnName => Work:Dept`
- As mentioned above, BigTable is quite suitable for **sparse data**. This is because empty columns are not stored.

Timestamps

Each column cell can contain multiple versions of the content. For example, as we saw in the earlier example, we may have several timestamped versions of an employee's email. A 64-bit timestamp identifies each version that either represents real time or a custom value assigned by the client. While reading, if no timestamp is specified, BigTable returns the most recent version. If the client specifies a timestamp, the latest version that is earlier than the specified timestamp is returned.

BigTable supports two per-column-family settings to garbage-collect cell versions automatically. The client can specify that only the last 'n' versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the previous seven days).

[< Back](#)[Next >](#)

BigTable: Introduction

System APIs

System APIs

Let's explore BigTable APIs.

We'll cover the following



- Metadata operations
- Data operations

BigTable provides APIs for two types of operations:

- Metadata operations
- Data operations

Metadata operations

BigTable provides APIs for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Data operations

Clients can insert, modify, or delete values in BigTable. Clients can also lookup values from individual rows or iterate over a subset of the data in a table.

- BigTable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a

single row key.

- Bigtable does not support transactions across row keys, but provides a client interface for batch writing across row keys.
- BigTable allows cells to be used as integer counters.
- A set of wrappers allow a BigTable to be used both as an input source and as an output target for MapReduce (https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html) jobs.
- Clients can also write scripts in Sawzall (a language developed at Google) to instruct server-side data processing (transform, filter, aggregate) prior to the network fetch.

Here are APIs for write operations:

- **Set()** : write cells in a row
- **DeleteCells()** : delete cells in a row
- **DeleteRow()** : delete all cells in a row

A read or scan operation can read arbitrary cells in a BigTable:

- Each row read operation is atomic.
- Can ask for data from just one row, all rows, etc.
- Can restrict returned rows to a particular range.
- Can ask for all columns, just certain columns families, or specific columns.

← Back

Next →

Partitioning and High-level Architecture

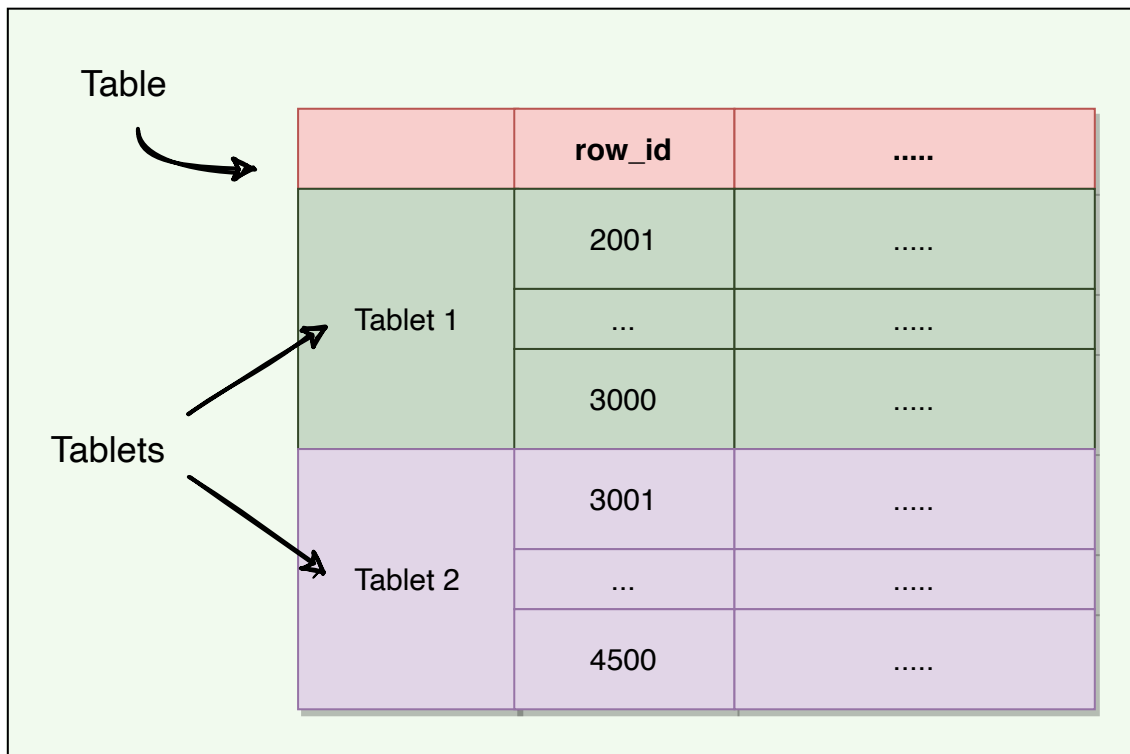
This lesson gives a brief overview of BigTable's architecture and its data partitioning scheme.

We'll cover the following ^

- Table partitioning
- High-level architecture

Table partitioning

A single instance of a BigTable implementation is known as a cluster. Each cluster can store a number of tables where each table is split into multiple **Tablets**, each around 100–200 MB in size.



- A Tablet holds a contiguous range of rows.
- The table is broken into Tablets at row boundaries.
- Initially, each table consists of only one Tablet. As the table grows, multiple Tablets are created. By default, a table is split at around 100 to 200 MB.
- Tablets are the unit of distribution and load balancing (more about this later).
- Since the table is sorted by row, reads of short ranges of rows are always efficient, that is to say, communicating with a small number of Tablets. This also means that selecting a row key with a high degree of locality is very important.
- Each Tablet is assigned to a **Tablet server** (discussed later), which manages all read/write requests of that Tablet.

High-level architecture

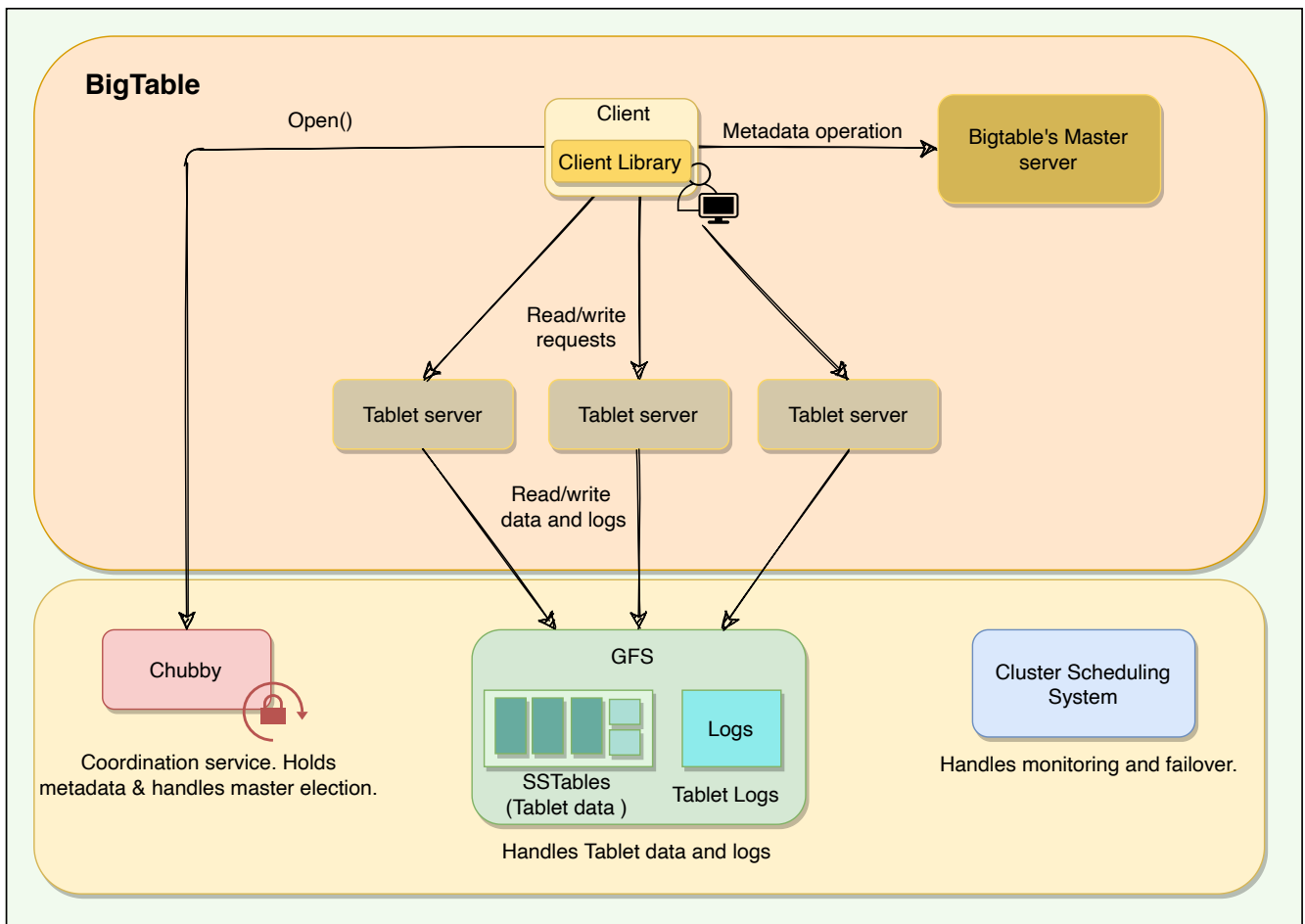
The architecture of a BigTable cluster consists of three major components:

1. **Client Library:** A library component that is linked into every client. The client talks to BigTable through this library.
2. **One master server:** Responsible for performing metadata operations and assigning Tablets to Tablet servers and managing them.
3. **Many Tablet servers:** Each Tablet server serves read and write of the data to the Tablets it is assigned.

BigTable is built on top of several other pieces from Google infrastructure:

1. **GFS:** BigTable uses the Google File System to store its data and log files.
2. **SSTable:** Google's SSTable (Sorted String Table) file format is used to store BigTable data. SSTable provides a persistent, ordered, and immutable map from keys to values (more on this later). SSTable is designed in such a way that any data access requires, at most, a single disk access.
3. **Chubby:** BigTable uses a highly available and persistent distributed lock service called Chubby to handle synchronization issues and store configuration information.
4. **Cluster Scheduling System:** Google has a cluster management system that schedules, monitors, and manages the Bigtable's cluster.

Let's understand these components one by one.



High-level architecture of BigTable

← Back

System APIs

Next →

SSTable

SSTable

Let's learn how Tablets are stored in SSTables.

We'll cover the following

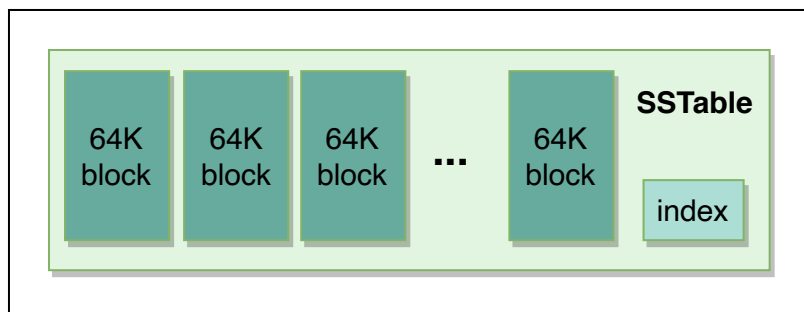


- How are Tablets stored in GFS?
- Table vs. Tablet vs. SSTable

How are Tablets stored in GFS?

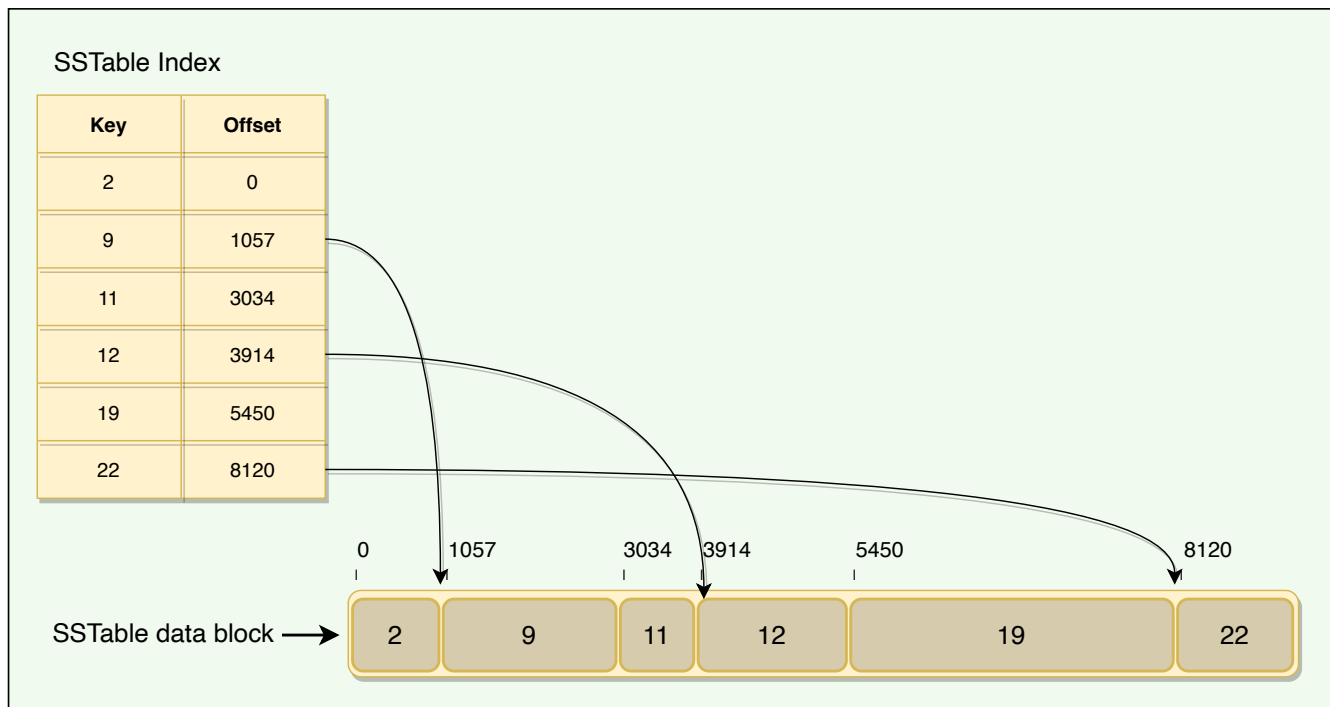
BigTable uses Google File System (GFS), a persistent distributed file storage system to store data as files. The file format used by BigTable to store its files is called SSTable:

- SSTables are persisted, ordered maps of keys to values, where both keys and values are arbitrary byte strings.
- Each Tablet is stored in GFS as a sequence of files called SSTables.
- An SSTable consists of a sequence of data blocks (typically 64KB in size).



SSTable contains multiple blocks

- A block index is used to locate blocks; the index is loaded into memory when the SSTable is opened.



Reading data from SSTable

- A lookup can be performed with a single disk seek. We first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from the disk.
- To read data from an SSTable, it can either be copied from disk to memory as a whole or just the index. The former approach avoids subsequent disk seeks for lookups, while the latter requires a single disk seek for each lookup.
- SSTables provide two operations:
 - Get the value associated with a given key
 - Iterate over a set of values in a given key range
- Each SSTable is immutable (read-only) once written to GFS. If new data is added, a new SSTable is created. Once an old SSTable is no longer needed, it is set out for garbage collection. SSTable immutability is at the

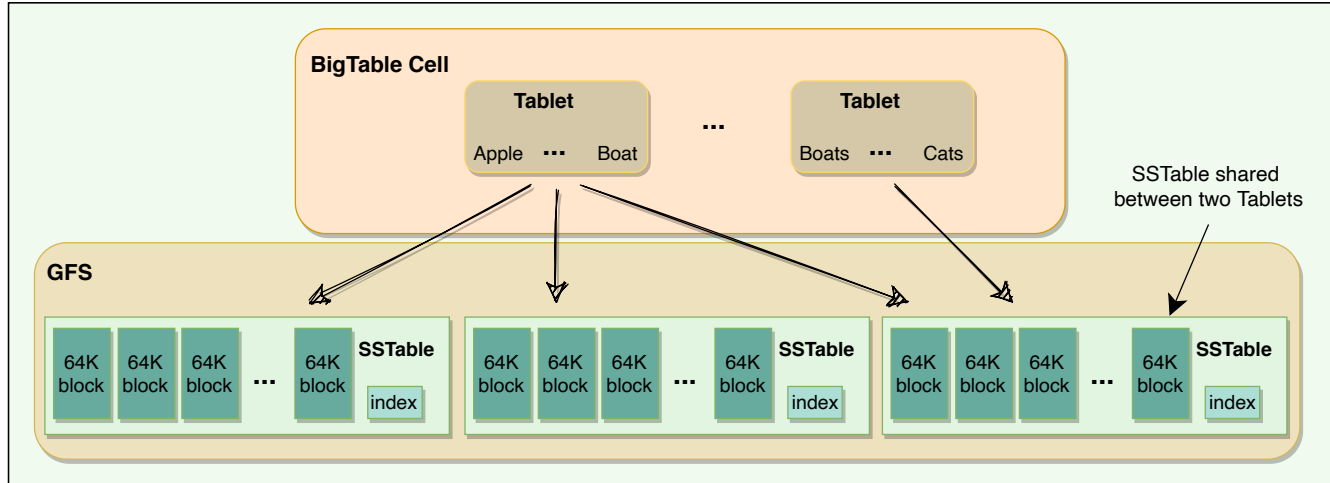
core of BigTable's data checkpointing and recovery routines. SSTable's immutability provides following advantages:

- No synchronization is needed during read operations.
- This also makes it easier to split Tablets.
- Garbage collector handles the permanent removal of deleted or stale data.

Table vs. Tablet vs. SSTable

Here is how we can define the relationship between Table, Tablet and SSTable:

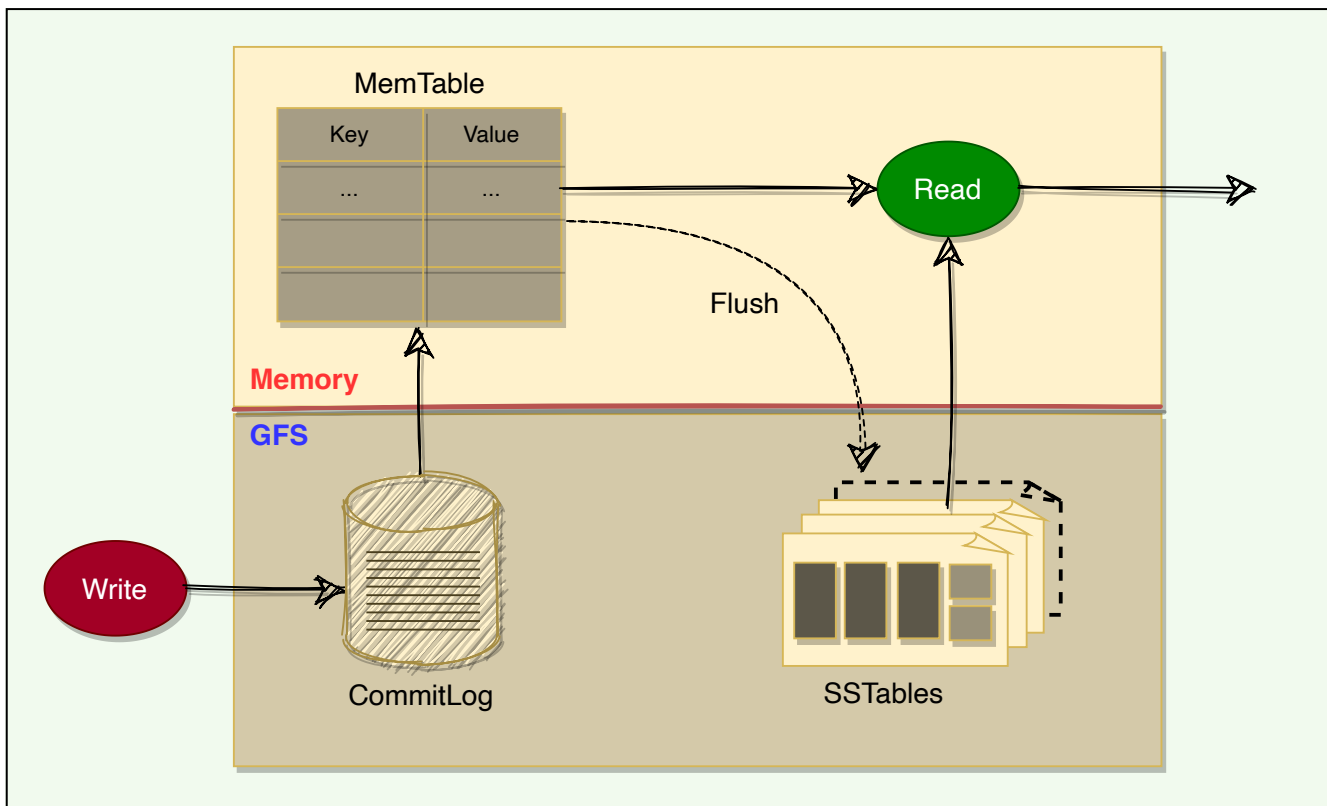
- Multiple Tablets make up a table.
- SSTables can be shared by multiple Tablets.
- Tablets do not overlap, SSTables can overlap.



Tablet vs SSTable

- To improve write performance, BigTable uses an in-memory, mutable sorted buffer called **MemTable** to store recent updates. As more writes are performed, MemTable size increases, and when it reaches a threshold, the MemTable is frozen, a new MemTable is created, and the frozen MemTable is converted to an SSTable and written to GFS.

- Each data update is also written to a commit-log which is also stored in GFS. This log contains redo records used for recovery if a Tablet server fails before committing a MemTable to SSTable.
- While reading, the data can be in MemTables or SSTables. Since both these tables are sorted, it is easy to find the most recent data.



Read and write workflow

[< Back](#)

[Next >](#)

Partitioning and High-level Architecture

GFS and Chubby

GFS and Chubby

Let's explore how BigTable interacts with GFS and Chubby.

We'll cover the following

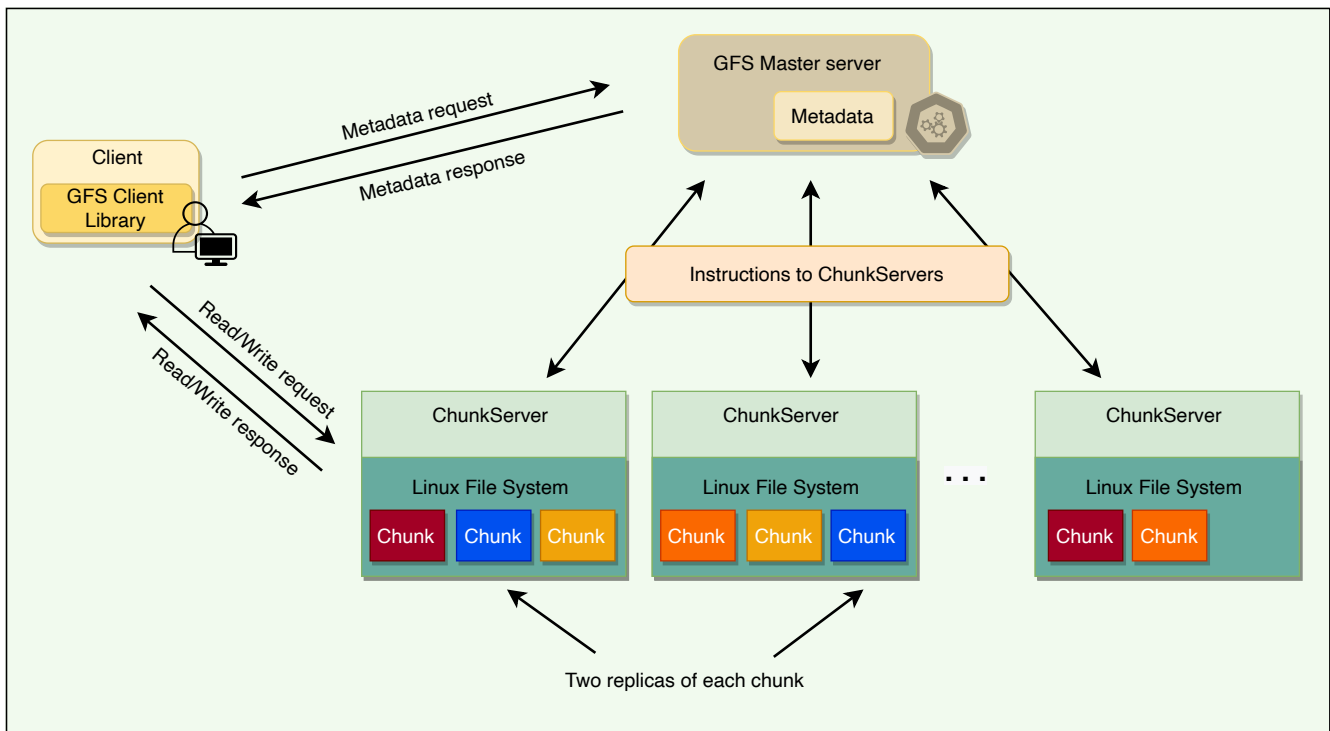


- GFS
- Chubby

GFS

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

- GFS files are broken into fixed-size blocks, called Chunks.
- Chunks are stored on data servers called ChunkServers.
- GFS master manages the metadata.
- SSTables are divided into fixed-size, blocks and these blocks are stored on ChunkServers.
- Each chunk in GFS is replicated across multiple ChunkServers for reliability.
- Clients interact with the GFS master for metadata, but all data transfers happen directly between the client and ChunkServers.



High-level architecture of GFS

For a detailed discussion, please see GFS

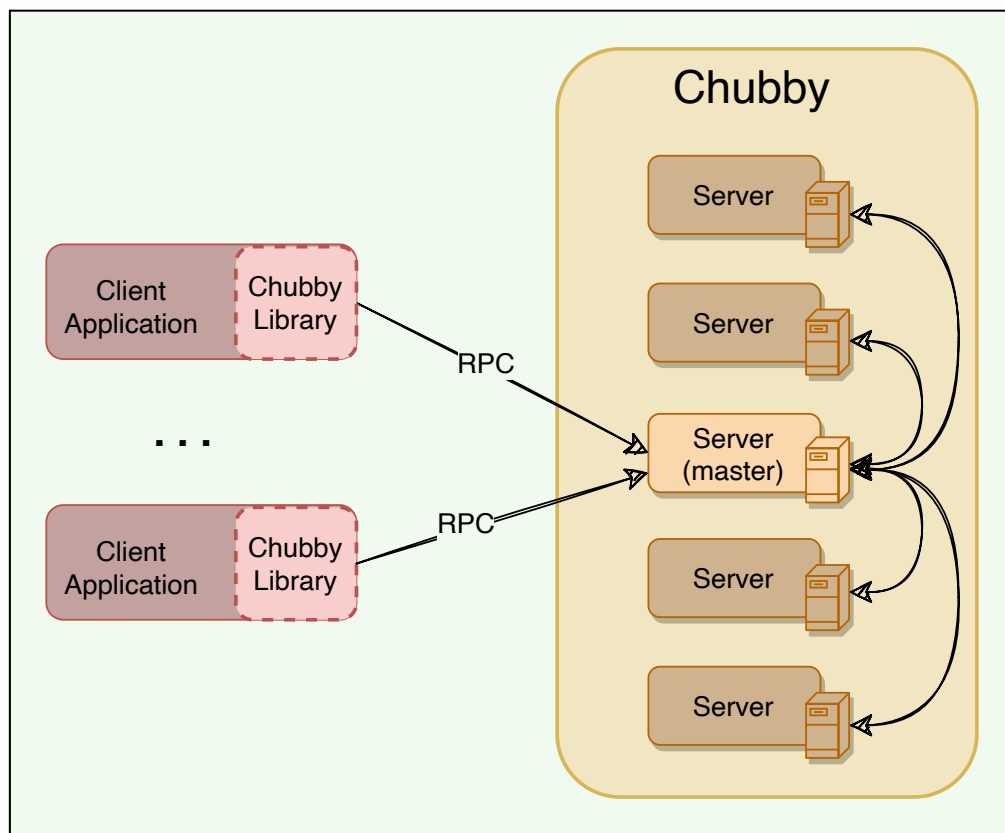
(<https://www.educative.io/collection/page/5668639101419520/5559029852536832/6383131278442496>).

Chubby

Chubby is a highly available and persistent distributed locking service that allows a multi-thousand node Bigtable cluster to stay coordinated.

- Chubby usually runs with five active replicas, one of which is elected as the master to serve requests. To remain alive, a majority of Chubby replicas must be running.
- BigTable depends on Chubby so much that if Chubby is unavailable for an extended period of time, BigTable will also become unavailable.
- Chubby uses the Paxos algorithm to keep its replicas consistent in the face of failure.

- Chubby provides a namespace consisting of files and directories. Each file or directory can be used as a lock.
- Read and write access to a Chubby file is atomic.
- Each Chubby client maintains a session with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.
- In BigTable, Chubby is used to:
 - Ensure there is only one active master. The master maintains a session lease with Chubby and periodically renews it to retain the status of the master.
 - Store the bootstrap location of BigTable data (discussed later)
 - Discover new Tablet servers as well as the failure of existing ones
 - Store BigTable schema information (the column family information for each table)
 - Store Access Control Lists (ACLs)



High-level architecture of Chubby

[< Back](#)

[Next >](#)

SSTable

Bigtable Components

Bigtable Components

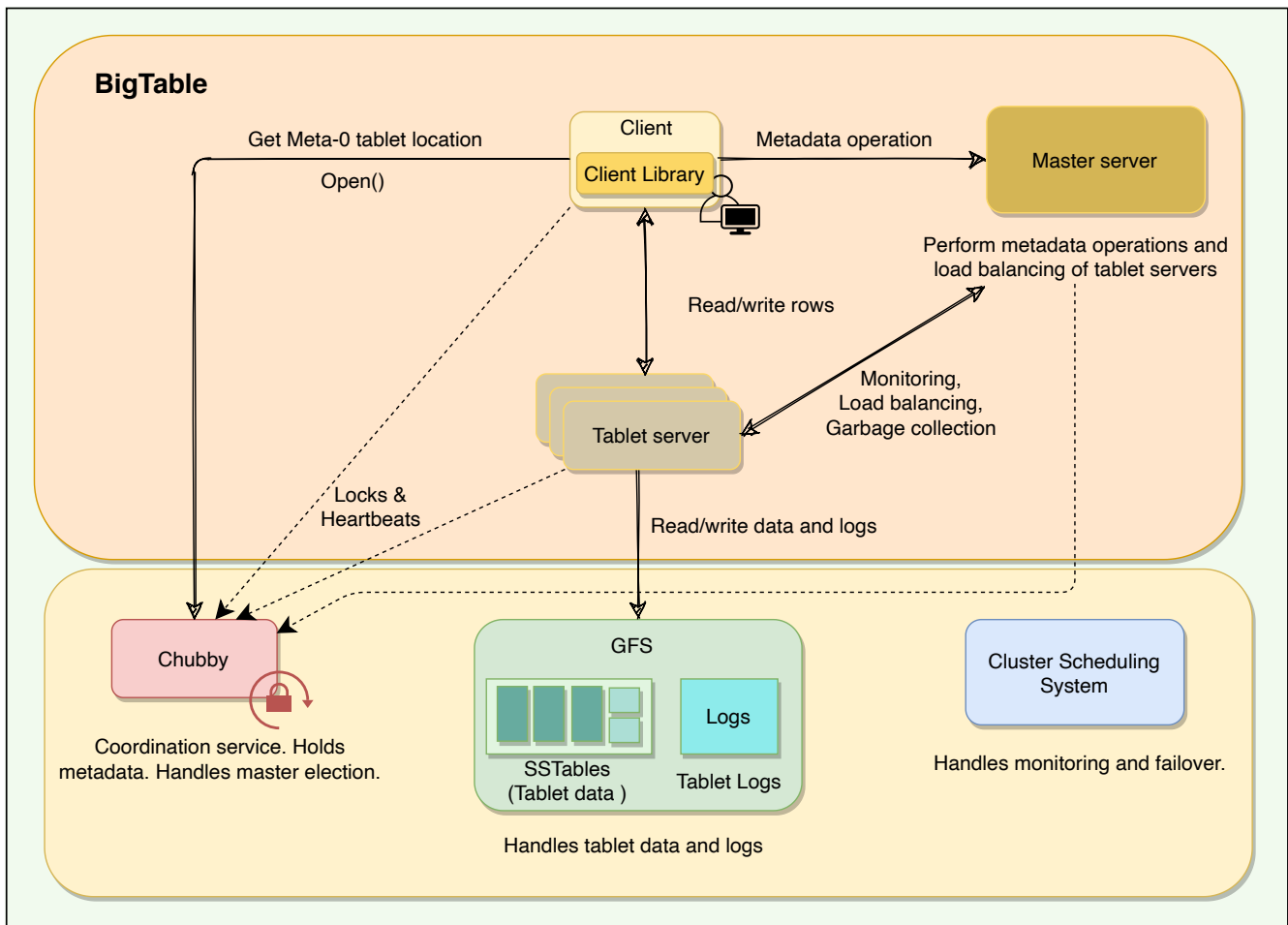
Let's explore the components that constitute BigTable.

We'll cover the following ^

- BigTable master server
- Tablet servers

As described previously, a BigTable cluster consists of three major components:

1. A library component that is linked into every client
2. One master server
3. Many Tablet servers



BigTable architecture

BigTable master server

There is only one master server in a BigTable cluster, and it is responsible for:

- Assigning Tablets to Tablet servers and ensuring effective load balancing
- Monitoring the status of Tablet servers and managing the joining or failure of Tablet server
- Garbage collection of the underlying files stored in GFS
- Handling metadata operations such as table and column family creations

Bigtable master is not involved in the core task of mapping tablets onto the underlying files in GFS (Tablet servers handle this). This means that Bigtable clients do not have to communicate with the master at all. This design decision significantly reduces the load on the master and the possibility of the master becoming a bottleneck.

Tablet servers

- Each Tablet server is assigned ownership of a number of Tablets (typically 10–1,000 Tablets per server) by the master.
- Each Tablet server serves read and write requests of the data of the Tablets it is assigned. The client communicates directly with the Tablet servers for reads/writes.
- Tablet servers can be added or removed dynamically from a cluster to accommodate changes in the workloads.
- Tablet creation, deletion, or merging is initiated by the master server, while Tablet partition is handled by Tablet servers who notify the master.

[< Back](#)

GFS and Chubby

[Next >](#)

Working with Tablets

Working with Tablets

We'll cover the following



- Locating Tablets
- Assigning Tablets
- Monitoring Tablet servers
- Load-balancing Tablet servers

Locating Tablets

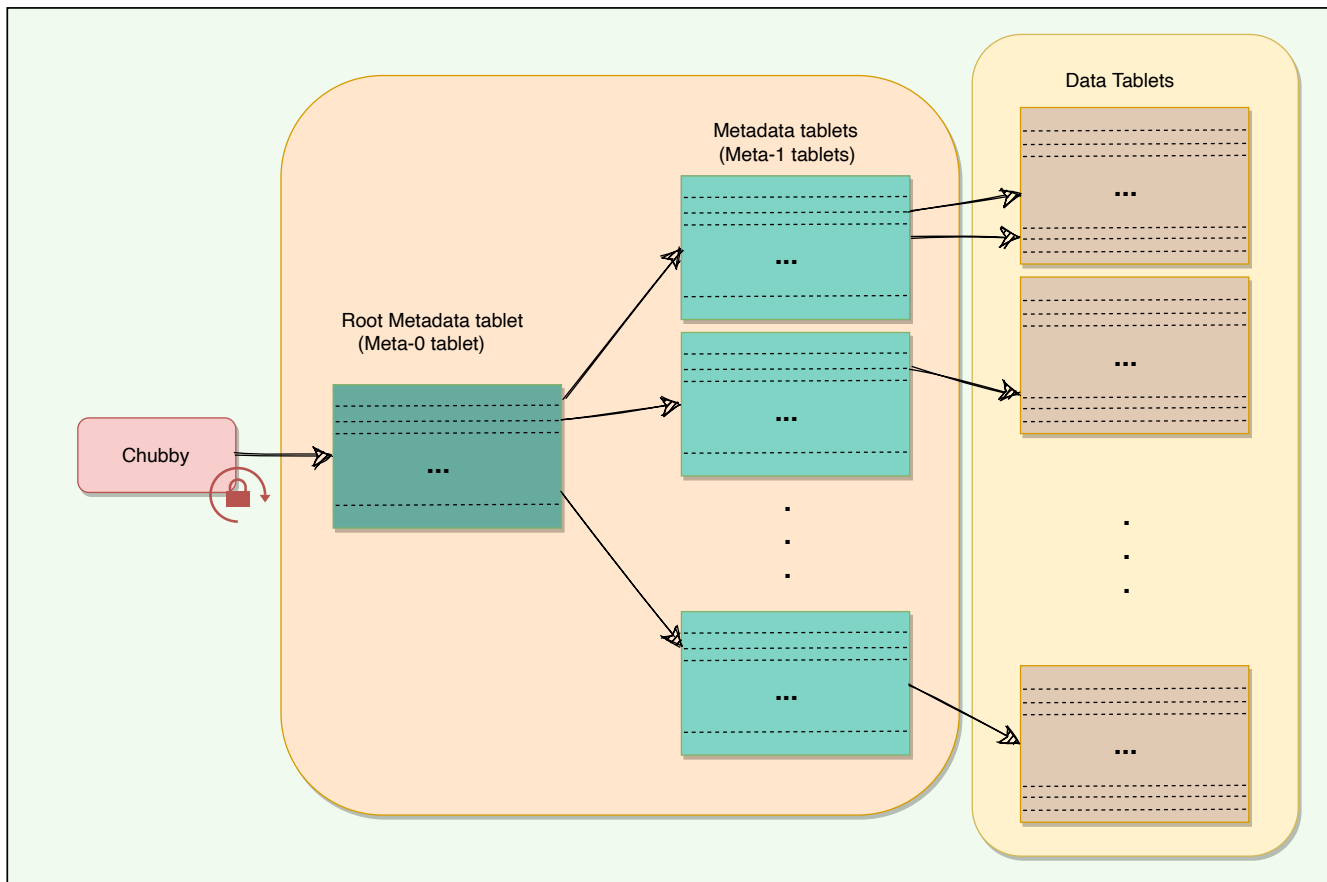
Since Tablets move around from server to server (due to load balancing, Tablet server failures, etc.), given a row, how do we find the correct Tablet server? To answer this, we need to find the Tablet whose row range covers the target row. BigTable maintains a 3-level hierarchy, analogous to that of a B+ tree, to store Tablet location information.

BigTable creates a special table, called **Metadata** table, to store Tablet locations. This Metadata table contains one row per Tablet that tells us which Tablet server is serving this Tablet. Each row in the METADATA table stores a Tablet's location under a row key that is an encoding of the Tablet's table identifier and its end row.

```
METADATA:   Key: table id + end row
            Data: tablet server location
```

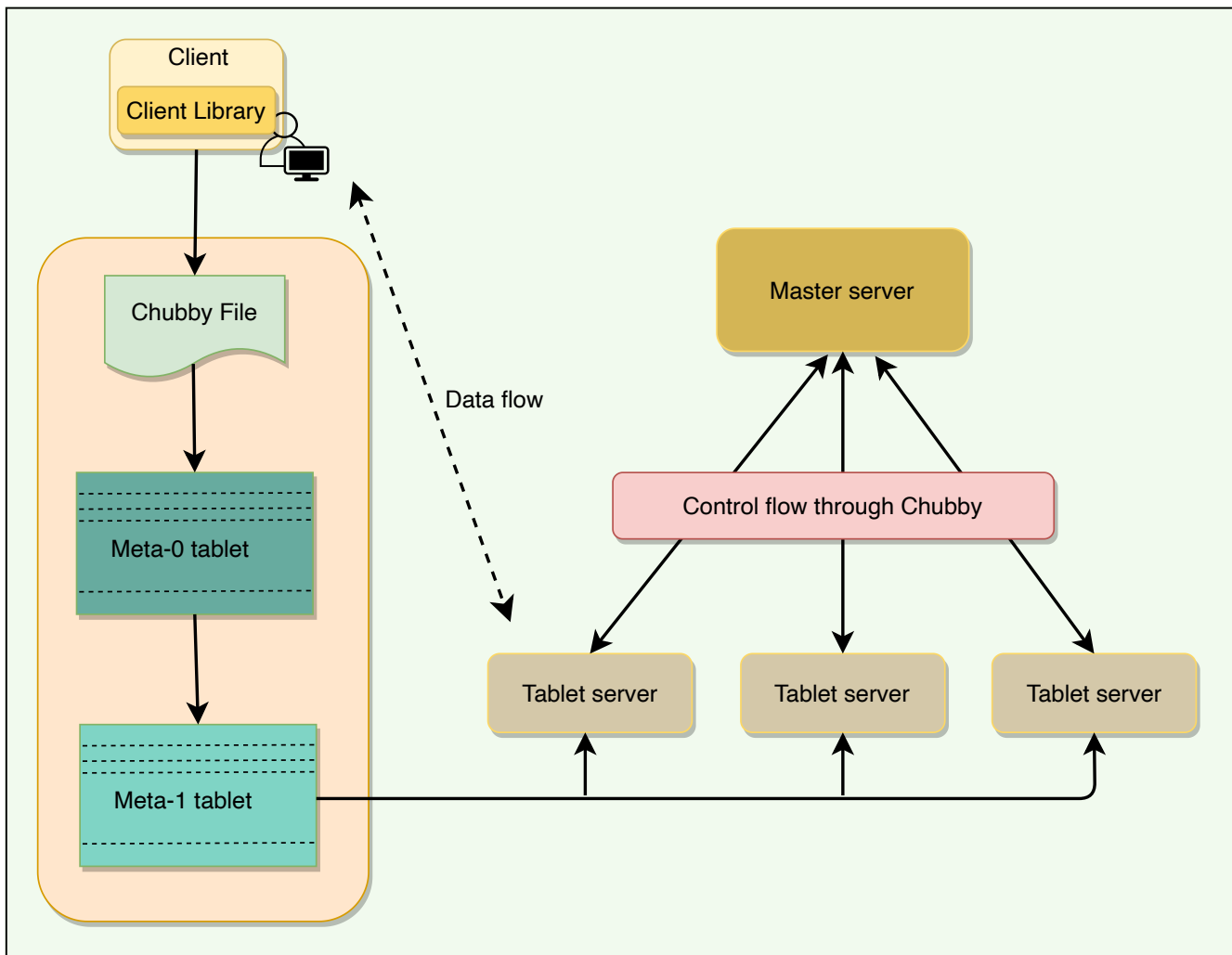
BigTable stores the information about the Metadata table in two parts:

1. **Meta-1 Tablet** has one row for each data Tablet (or non-meta Tablet). Since Meta-1 Tablet can be big, it is split into multiple metadata Tablets and distributed to multiple Tablet servers.
2. **Meta-0 Tablet** has one row for each Meta-1 Tablet. Meta-0 table never gets split. BigTable stores the location of the Meta-0 Tablet in a Chubby file.



Metadata tablets

A BigTable client seeking the location of a Tablet starts the search by looking up a particular file in Chubby that is known to hold the location of the Meta-0 Tablet. This Meta-0 Tablet contains information about other metadata Tablets, which in turn contain the location of the actual data Tablets. With this scheme, the depth of the tree is limited to three. For efficiency, the client library caches Tablet locations and also prefetch metadata associated with other Tablets whenever it reads the METADATA table.



Control and data flow in BigTable

Assigning Tablets

A Tablet is assigned to only one Tablet server at any time. The master keeps track of the set of live Tablet servers and the mapping of Tablets to Tablet servers. The master also keeps track of any unassigned Tablets and assigns them to Tablet servers with sufficient room.

When a Tablet server starts, it creates and acquires an exclusive lock on a uniquely named file in Chubby's "servers" directory. This mechanism is used to tell the master that the Tablet server is alive. When the master is restarted by the Cluster Management System, the following things happen:

1. The master grabs a unique master lock in Chubby to prevent multiple master instantiations.
2. The master scans the Chubby's "servers" directory to find the live Tablet servers.
3. The master communicates with every live Tablet server to discover what Tablets are assigned to each server.
4. The master scans the METADATA table to learn the full set of Tablets. Whenever this scan encounters a Tablet that is not already assigned, the master adds the Tablet to the set of unassigned Tablets. Similarly, the master builds a set of unassigned Tablet servers, which are eligible for Tablet assignment. The master uses this information to assign the unassigned Tablets to appropriate Tablet servers.

Monitoring Tablet servers

As stated above, BigTable maintains a 'Servers' directory in Chubby, which contains one file for each live Tablet server. Whenever a new Tablet server comes online, it creates a new file in this directory to signal its availability and obtains an exclusive lock on this file. As long as a Tablet server retains the lock on its Chubby file, it is considered alive.

BigTable's master keeps monitoring the 'Servers' directory, and whenever it sees a new file in this directory, it knows that a new Tablet server has become available and is ready to be assigned Tablets. In addition to that, the master regularly checks the status of the lock. If the lock is lost, the master assumes that there is a problem either with the Tablet server or the Chubby. In such a case, the master tries to acquire the lock, and if it succeeds, it concludes that Chubby is working fine, and the Tablet server is having problems. The master, in this case, deletes the file and reassigns the tablets of the failing Tablet server. The deletion of the file works as a signal for the failing Tablet server to terminate itself and stop serving the Tablets.

Whenever a Table server loses its lock on the file it has created in the “servers” directory, it stops serving its Tablets. It tries to acquire the lock again, and if it succeeds, it considers it a temporary network problem and starts serving the Tablets again. If the file gets deleted, then the Tablet server terminates itself to start afresh.

Load-balancing Tablet servers

As described above, the master is responsible for assigning Tablets to Tablet servers. The master keeps track of all available Tablet servers and maintains the list of Tablets that the cluster is supposed to serve. In addition to that, the master periodically asks Tablet servers about their current load. All this information gives the master a global view of the cluster and helps assign and load-balance Tablets.

[< Back](#)[Next >](#)

Bigtable Components

The Life of BigTable's Read & Write O...

The Life of BigTable's Read & Write Operations

Let's explore how BigTable handles its read and write operations.

We'll cover the following

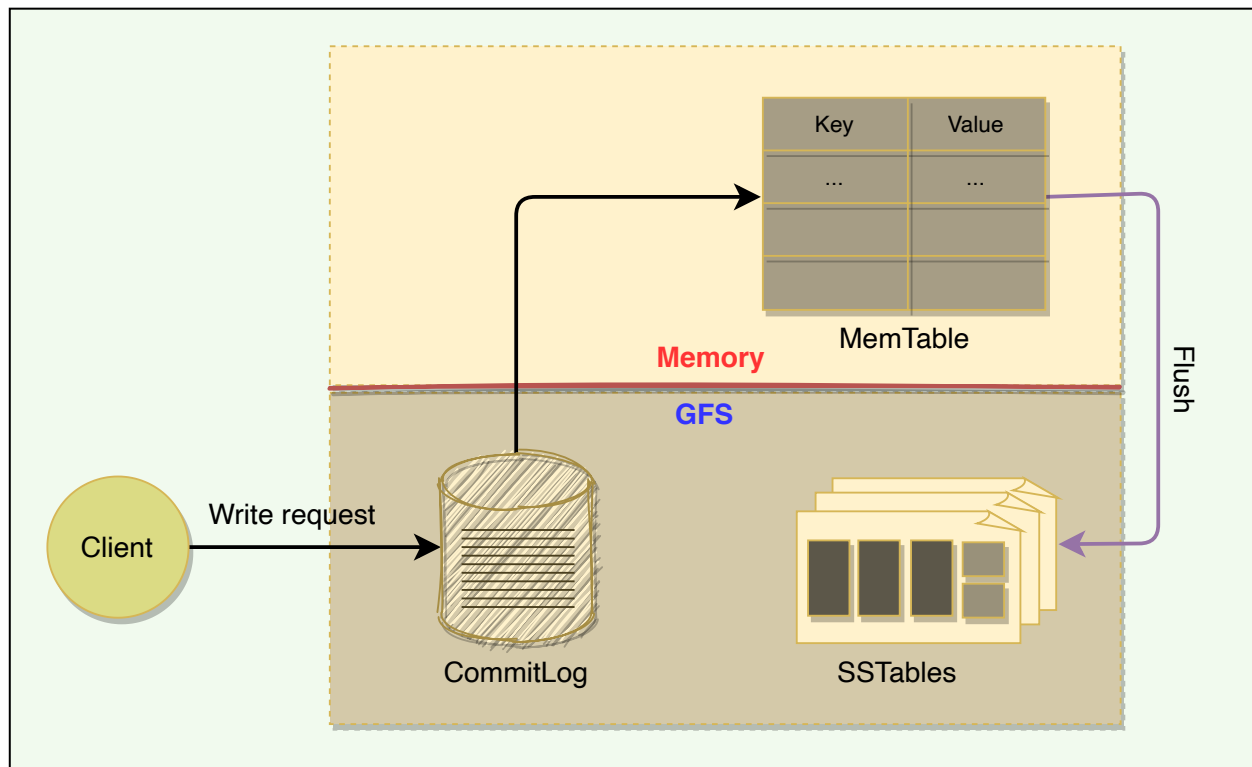


- Write request
- Read request

Write request

Upon receiving a write request, a Tablet server performs the following set of steps:

1. Checks that the request is well-formed.
2. Checks that the sender is authorized to perform the mutation. This authorization is performed based on the Access Control Lists (ACLs) that are stored in a chubby file.
3. If the above two conditions are met, the mutation is written to the commit-log in GFS that stores redo records.
4. Once the mutation is committed to the commit-log, its contents are stored in memory in a sorted buffer called MemTable.
5. After inserting the data into the MemTable, acknowledgment is sent to the client that the data has been successfully written.
6. Periodically, MemTables are flushed to SSTables, and SSTables are merged during compaction (discussed later).

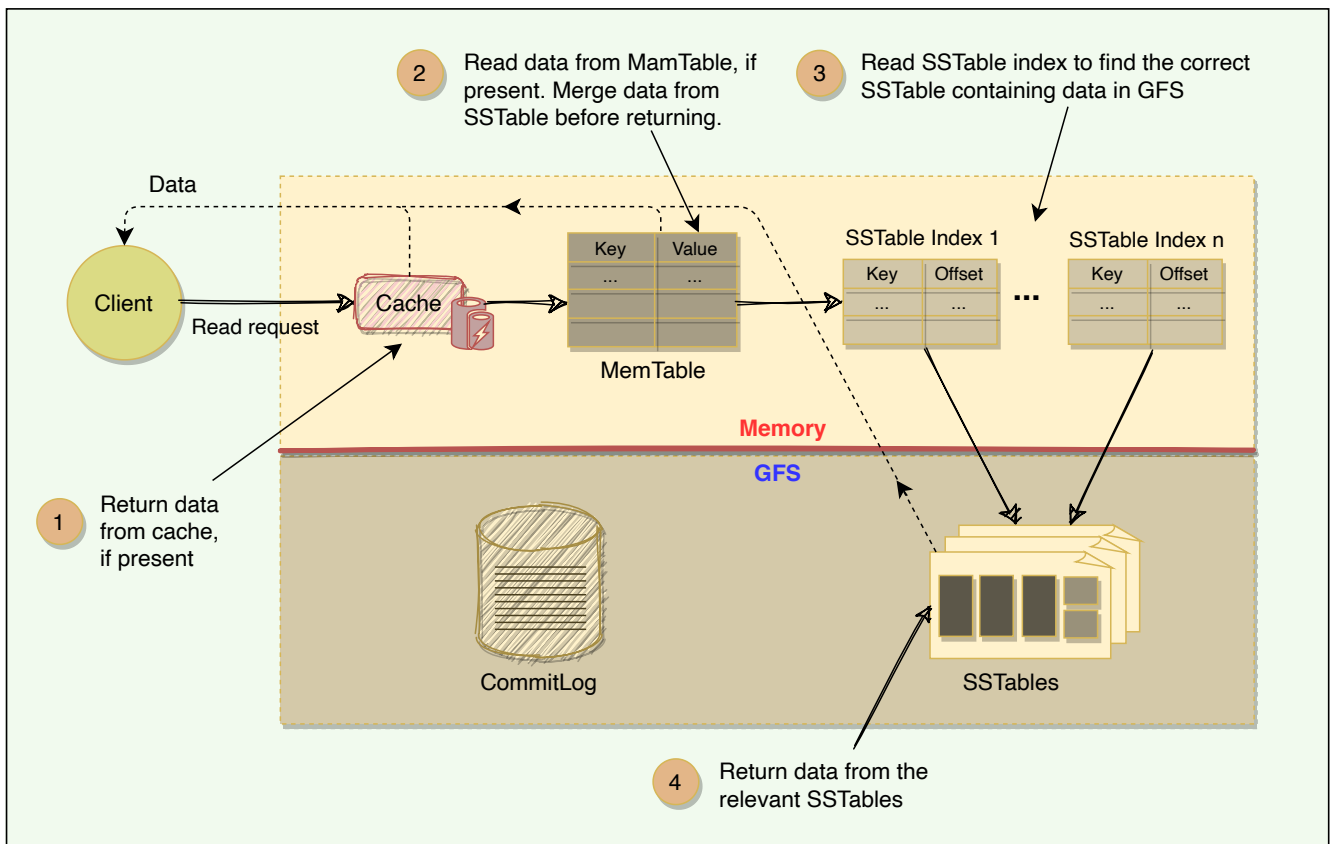


The anatomy of a write request

Read request

Upon receiving a read request, a Tablet server performs the following set of steps:

1. Checks that the request is well-formed and the sender is authorized
2. Returns the rows if they are available in the Cache (discussed later)
3. Reads MemTable first to find the required rows
4. Reads SSTable indexes that are loaded in memory to find SSTables that will have the required data, then reads the rows from those SSTables
5. Merge rows read from MemTable and SSTables to find the required version of the data. Since the SSTables and the MemTable are sorted, the merged view can be formed efficiently.



The anatomy of a read request

← Back

Next →

Working with Tablets

Fault Tolerance and Compaction

Fault Tolerance and Compaction

Let's learn how BigTable handles fault tolerance and data compaction.

We'll cover the following



- Fault tolerance and replication
 - Fault tolerance in Chubby and GFS
 - Fault tolerance for Tablet server
 - Fault tolerance for the Master
- Compaction

Fault tolerance and replication

Fault tolerance in Chubby and GFS

As discussed earlier

(<https://www.educative.io/collection/page/5668639101419520/5559029852536832/6338075595112448>), BigTable uses two independent systems Chubby and GFS. Both of these systems adopt a replication strategy for fault tolerance and higher availability. For example, a Chubby cell usually consists of five servers, where one server becomes the master and the remaining four work as replicas. In case the master fails, one of the replicas is elected to become the leader; thus, minimizing Chubby's downtime. Similarly, GFS stores multiple copies of data on different ChunkServers.

Fault tolerance for Tablet server

BigTable's master is responsible for monitoring the Tablet servers. The master does this by periodically checking the status of the Chubby lock against each Tablet server. When the master finds out that a Tablet server has gone dead, it reassigns the tablets of the failing Tablet server.

Fault tolerance for the Master

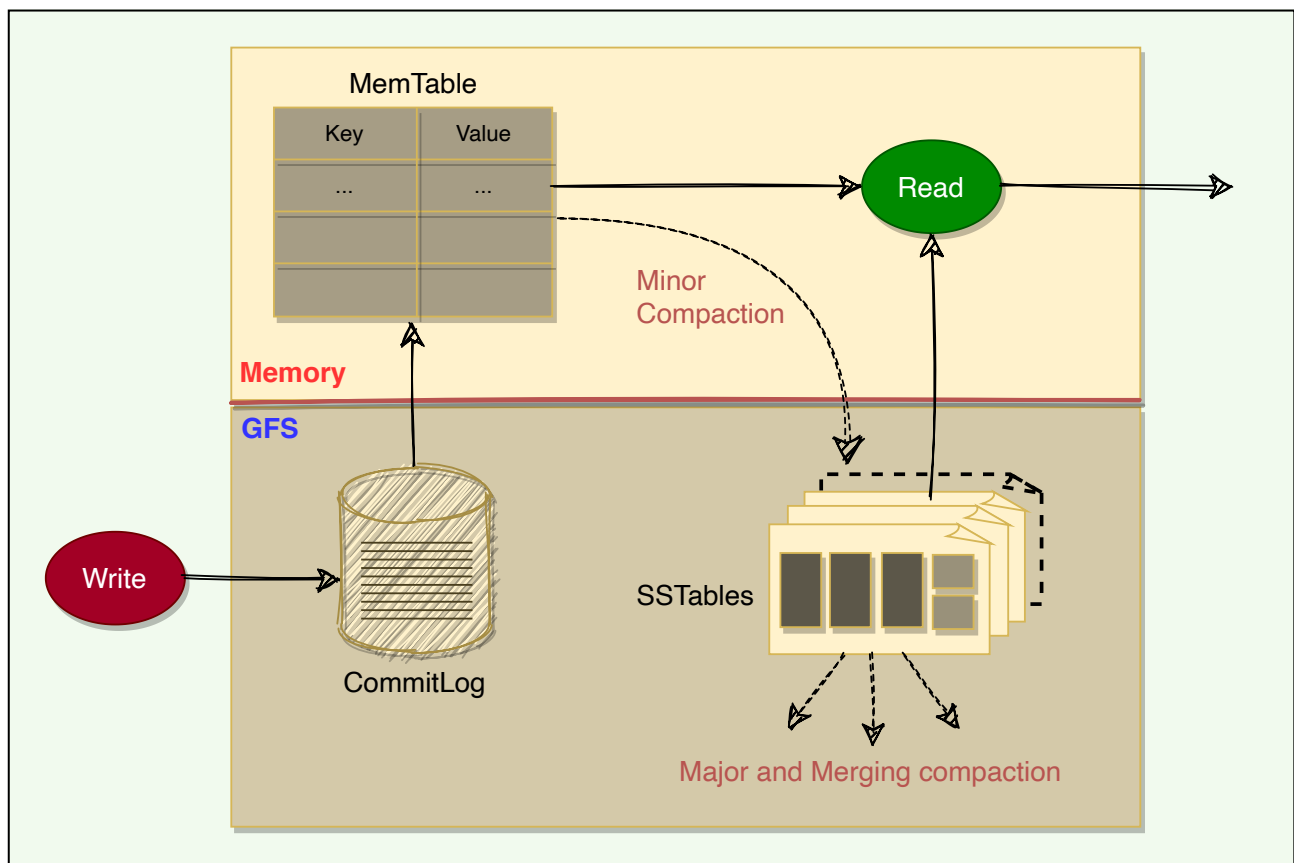
The master acquires a lock in a Chubby file and maintains a lease. If, at any time, the master's lease expires, it kills itself. When Google's Cluster Management System finds out that there is no active master, it starts one up. The new master has to acquire the lock on the Chubby file before acting as the master.

Compaction

Mutilations in BigTable take up extra space till compaction happens. BigTable manages compaction behind the scenes. Here is the list of compactions:

1. **Minor Compaction:** As write operations are performed, the MemTable grows in size. When the MemTable reaches a certain threshold, it is frozen, and a new MemTable is created. The frozen MemTable is converted to an SSTable and written to GFS. This process is called minor compaction. Each minor compaction creates a new SSTable and has the following two benefits:
 - It reduces the memory usage of the Tablet server, as it flushes the MemTable to GFS. Once a MemTable is written to GFS, corresponding entries in the commit-log are also removed.
 - It reduces the amount of data that has to be read from the commit log during recovery if this server dies.

2. **Merging Compaction** — Minor compaction keeps increasing the count of SSTables. This means that read operations might need to merge updates from an arbitrary number of SSTables. To reduce the number of SSTables, a merging compaction is performed which reads the contents of a few SSTables and the MemTable and writes out a new SSTable. The input SSTables and MemTable can be discarded as soon as the compaction has finished.
3. **Major Compaction** — In Major compaction, all the SSTables are written into a single SSTable. SSTables created as a result of major compaction do not contain any deletion information or deleted data, whereas SSTables created from non-major compactions may contain deleted entries. Major compaction allows BigTable to reclaim resources used by deleted data and ensures that deleted data disappears from the system quickly, which is important for services storing sensitive data.



Major, minor, and merging compaction in BigTable

← Back

Next →

The Life of BigTable's Read & Write O...

BigTable Refinements

BigTable Refinements

This lesson will explore different refinements that BigTable implemented.

We'll cover the following



- Locality groups
- Compression
- Caching
- Bloom filters
- Unified commit Log
- Speeding up Tablet recovery

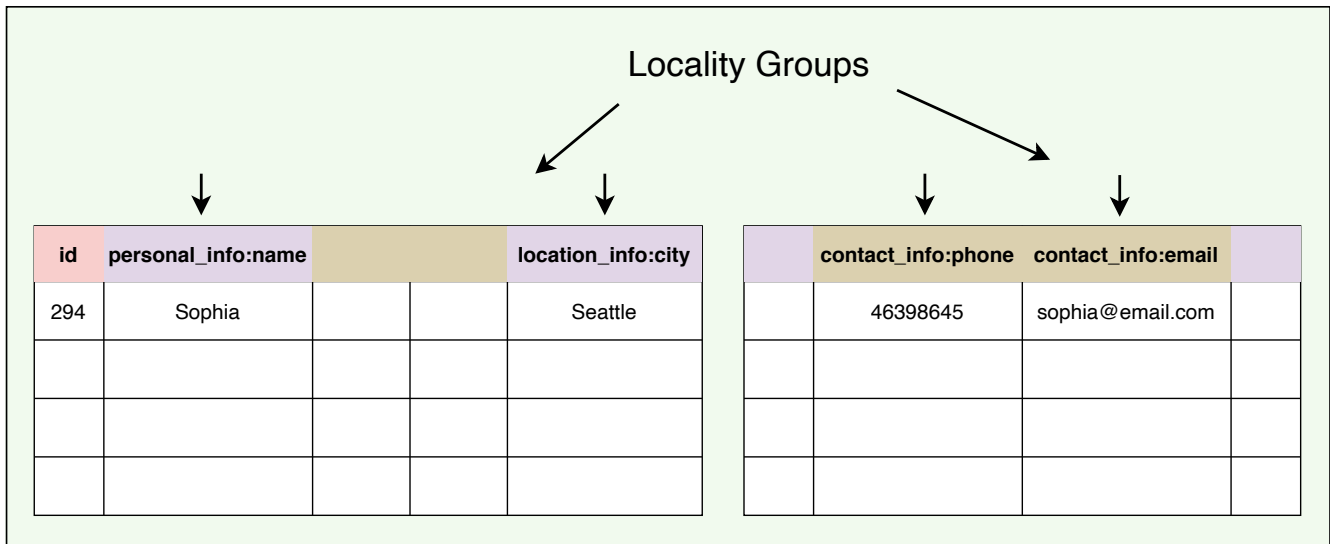
BigTable implemented certain refinements to achieve high performance, availability, and reliability. Here are their details:

Locality groups

Clients can club together multiple column families into a locality group. BigTable generates separate SSTables for each locality group. This has two benefits:

- Grouping columns that are frequently accessed together in a locality group enhances the read performance.
- Clients can explicitly declare any locality group to be in memory for faster access. This way, smaller locality groups that are frequently accessed can be kept in memory.

- Scans over one locality group are $O(\text{bytes_in_locality_group})$ and not $O(\text{bytes_in_table})$.



Grouping together columns to form locality groups

Compression

Clients can choose to compress the SSTable for a locality group to save space. BigTable allows its clients to choose compression techniques based on their application requirements. The compression ratio gets even better when multiple versions of the same data are stored. Compression is applied to each SSTable block separately.

Caching

To improve read performance, Tablet servers employ two levels of caching:

- Scan Cache** — It caches (key, value) pairs returned by the SSTable and is useful for applications that read the same data multiple times.

- **Block Cache** — It caches SSTable blocks read from GFS and is useful for the applications that tend to read the data which is close to the data they recently read (e.g., sequential or random reads of different columns in the same locality group within a frequently accessed row).

Bloom filters

Any read operation has to read from all SSTables that make up a Tablet. If these SSTables are not in memory, the read operation may end up doing many disk accesses. To reduce the number of disk accesses BigTable uses Bloom Filters.

Bloom Filters are created for SSTables (particularly for the locality groups). They help to reduce the number of disk accesses by predicting if an SSTable may contain data corresponding to a particular (row, column) pair. Bloom filters take a small amount of memory but can improve the read performance drastically.

Unified commit Log

Instead of maintaining separate commit log files for each Tablet, BigTable maintains one log file for a Tablet server. This gives better write performance. Since each write has to go to the commit log, writing to a large number of log files would be slow as it could cause a large number of disk seeks.

One disadvantage of having a single log file is that it complicates the Tablet recovery process. When a Tablet server dies, the Tablets that it served will be moved to other Tablet servers. To recover the state for a Tablet, the new Tablet server needs to reapply the mutations for that Tablet from the commit log written by the original Tablet server. However, the mutations for these

Tablets were co-mingled in the same physical log file. One approach would be for each new Tablet server to read this full commit log file and apply just the entries needed for the Tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single Tablet from a failed Tablet server, then the log file would be read 100 times. BigTable avoids duplicating log reads by first sorting the commit log entries in order of the keys `<table, row name, log sequence number>`. In the sorted output, all mutations for a particular Tablet are contiguous and can therefore be read efficiently

To further improve the performance, each Tablet server maintains two log writing threads — each writing to its own and separate log file. Only one of the threads is active at a time. If one of the threads is performing poorly (say, due to network congestion), the writing switches to the other thread. Log entries have sequence numbers to allow the recovery process.

Speeding up Tablet recovery

As we saw above, one of the complicated and time-consuming tasks while loading Tablets is to ensure that the Tablet server loads all entries from the commit log. When the master moves a Tablet from one Tablet server to another, the source Tablet server performs compactions to ensure that the destination Tablet server does not have to read the commit log. This is done in three steps:

- In the first step, the source server performs a minor compaction. This compaction reduces the amount of data in the commit log.
- After this, the source Tablet server stops serving the Tablet.
- Finally, the source server performs another (usually very fast) minor compaction to apply any new log entries that have arrived while the first minor compaction was being performed. After this second minor

compaction is complete, the Tablet can be loaded on another Tablet server without requiring any recovery of log entries.

[< Back](#)

[Next >](#)

Fault Tolerance and Compaction

BigTable Characteristics

BigTable Characteristics

This lesson will explore some miscellaneous characteristics of BigTable.

We'll cover the following



- BigTable performance
- Dynamo vs. BigTable
- Datastores developed on the principles of BigTable

BigTable performance

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

- **Distributed multi-level map:** BigTable can run on a large number of machines.
- **Scalable** means that BigTable can be easily scaled horizontally by adding more nodes to the cluster without any performance impact. No manual intervention or rebalancing is required. BigTable achieves linear scalability and proven fault tolerance on commodity hardware.
- **Fault-tolerant and reliable:** Since data is replicated to multiple nodes, fault tolerance is pretty high.
- **Durable:** BigTable stores data permanently.
- **Centralized:** BigTable adopts a single-master approach to maintain data consistency and a centralized view of the state of the system.
- **Separation between control and data:** BigTable maintains a strict separation between control and data flow. Clients talk to the Master for

all metadata operations, whereas all data access happens directly between the Clients and the Tablet servers.

Dynamo vs. BigTable

Here is the comparison between Dynamo and BigTable:

	Dynamo	BigTable
Architecture	Decentralized Every node has same set of responsibilities	Centralized Master handles metadata, tablet servers handle read/write
Data Model	Key-value	Multidimensional sorted map.
Security	x	Access rights at column family level
Partitioning	Consistent Hashing Each node is assigned to a random position on the ring.	Tablets Each table is broken into a contiguous range of rows called tablets.
Replication	Sloppy Quorum Each data item is replicated to 'N' number of nodes.	GFS Chunk replication Data is stored in GFS. Files in GFS are broken into chunks, and these chunks are replicated to different servers.
CAP	AP	CP
Operations	By key	By key range

Storage	Plug-in	SSTables in GFS
Memberships and failure detection	Gossip based protocol	Handshakes initiated by the master

Datastores developed on the principles of BigTable

Google's BigTable has inspired many NoSQL systems. Here is a list of a few famous ones:

HBase: HBase is an open-source, distributed non-relational database modeled after BigTable. It is built on top of the Hadoop Distributed File System (HDFS).

Hypertable: Similar to HBase, Hypertable is an open-source implementation of BigTable and is written in C++. Unlike BigTable, which uses only one storage layer (i.e., GFS), Hypertable is capable of running on top of any file system (e.g., HDFS (https://en.wikipedia.org/wiki/Apache_Hadoop#HDFS), GlusterFS (<https://en.wikipedia.org/wiki/Gluster#GlusterFS>), or the CloudStore (<https://en.wikipedia.org/wiki/CloudStore>)). To achieve this, the system has abstracted the interface to the file system by sending all data requests through a Distributed File System broker process.

Cassandra: Cassandra is a distributed, decentralized, and highly available NoSQL database. Its architecture is based on Dynamo and BigTable. Cassandra can be described as a BigTable-like datastore running on a Dynamo-like infrastructure. Cassandra is also a wide-column store and utilizes the storage model of BigTable, i.e., SSTables and MemTables.

[← Back](#)

[Next →](#)

BigTable Refinements

Summary: BigTable

Summary: BigTable

Here is a quick summary of BigTable for you!

We'll cover the following



- Summary
- References & further reading

Summary

- BigTable is Google's **distributed storage system** designed to manage large amounts of structured data with high availability, low latency, scalability, and fault-tolerance goals.
- BigTable is a **sparse, distributed**, persistent, **multidimensional sorted map**.
- The map is indexed by a unique key made up of a row key, a column key, and a timestamp (a 64-bit integer, “real time” in millisecond).
- Each row key is an arbitrary string of up to 64 kilobytes in size, although most keys are significantly smaller.
- Unlike a traditional relational database table, BigTable is a **wide-column datastore** with an unbounded number of columns.
- Columns are grouped into **column families**. Each column family stores similar types of data under a ‘family:qualifier’ column key.
- The row key and the column (family:qualifier) key uniquely identify a data cell. Within each cell, the data contents are further indexed by timestamps providing multiple versions of the data in time.

- Every read or write of data under a single row is atomic. Atomicity across rows is not guaranteed.
- BigTable provides APIs for metadata operations like creating and deleting tables and column families. BigTable clients can use data operation APIs for writing or deleting values, lookup values from individual rows, or iterate over a subset of the data in a table.
- A table is split into smaller ranges of rows called **Tablets**. A Tablet holds a contiguous range of rows.
- Tablets are the unit of distribution and load balancing.
- BigTable architecture consists of **one master server** and **multiple Tablet servers**.
- Master is responsible for assigning Tablets to Tablet servers, as well as monitoring and balancing Tablet servers' load.
- Each Tablet server serves read and write requests of the data to the Tablets it is assigned.
- BigTable clients communicate directly with the Tablet servers to read/write data.
- Each Tablet server stores the data in immutable **SSTable** files which are stored in **Google File System (GFS)**.
- New committed updates are first stored in a memory-based **MemTable**.
- BigTable performs all read operations against a combined view of SSTables and MemTable.
- Periodically, the MemTable is flushed into an SSTable, allowing for efficient memory utilization.
- To enhance read performance, BigTable makes use of caching and **Bloom filters**.
- BigTable relies heavily on distributed locking service **Chubby** for master server selection and monitoring.

References & further reading

- BigTable (<https://research.google/pubs/pub27898/>)
- SSTable (<https://medium.com/databasss/on-disk-io-part-3-lsm-trees-8b2da218496f>)
- Dynamo
(https://www.allthingsdistributed.com/2007/10/amazons_dynamo.html)
- Cassandra (<https://cassandra.apache.org/>)
- HBase (<https://hbase.apache.org/>)

← Back

BigTable Characteristics

Next →

Quiz: BigTable

System Design Patterns

Introduction: System Design Patterns

This lesson gives a brief overview of the system design patterns that we will be discussing in the following lessons.

In the following chapters, we will discuss a set of system design patterns. These patterns refer to common design problems related to distributed systems and their solutions. Knowing these patterns is very important as they can be applied to all types of distributed systems and are very handy, especially in a system design interview.

Here is the list of patterns we will be discussing:

1. Bloom Filters
2. Consistent Hashing
3. Quorum
4. Leader and Follower
5. Write-ahead Log
6. Segmented Log
7. High-Water mark
8. Lease
9. Heartbeat
10. Gossip Protocol
11. Phi Accrual Failure Detection
12. Split-brain
13. Fencing
14. Checksum
15. Vector Clocks
16. CAP Theorem

17. PACELEC Theorem

18. Hinted Handoff

19. Read Repair

20. Merkle Trees

Let's get going.

← Back

Mock Interview: BigTable

Next →

1. Bloom Filters

1. Bloom Filters

Let's learn about Bloom filters and how to use them.

We'll cover the following



- Background
- Definition
- Solution
- Example: BigTable

Background

If we have a large set of structured data (identified by record IDs) stored in a set of data files, what is the most efficient way to know which file might contain our required data? We don't want to read each file, as that would be slow, and we have to read a lot of data from the disk. One solution can be to build an index on each data file and store it in a separate index file. This index can map each record ID to its offset in the data file. Each index file will be sorted on the record ID. Now, if we want to search an ID in this index, the best we can do is a Binary Search. Can we do better than that?

Definition

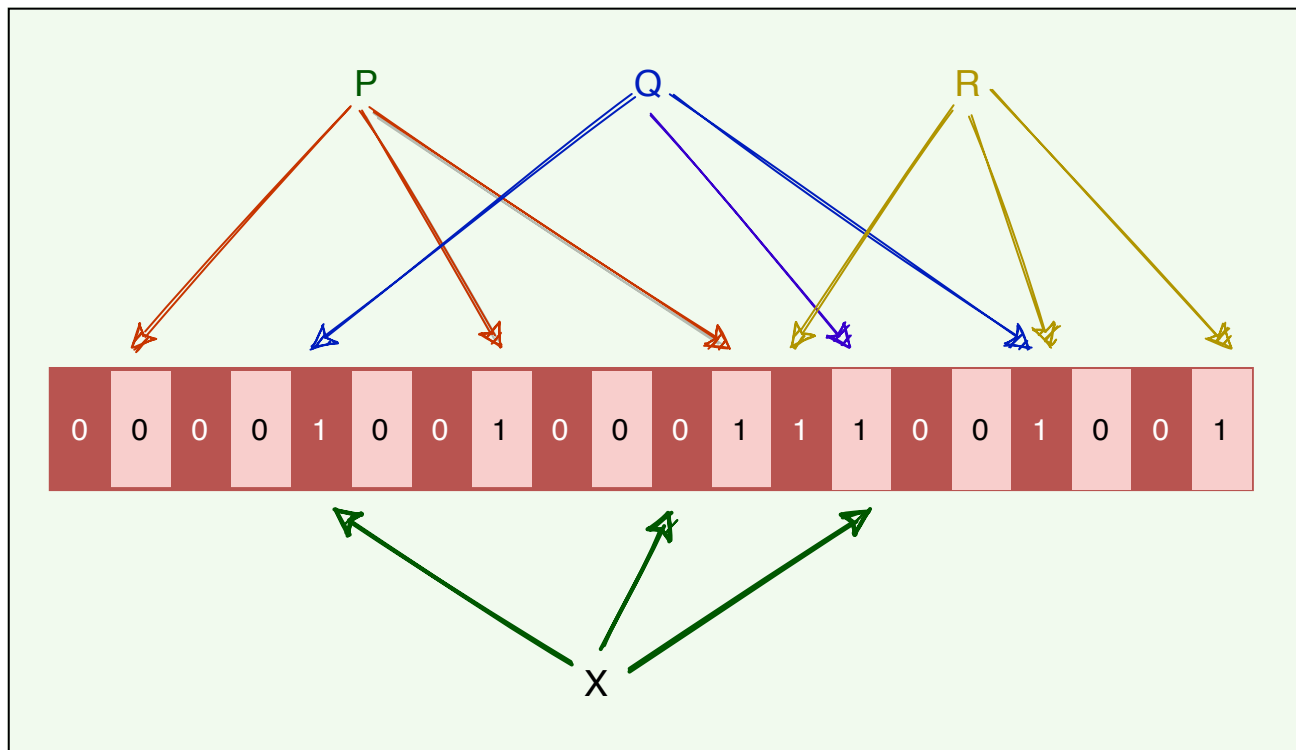
Use Bloom filters to quickly find if an element might be present in a set.

Solution

The Bloom filter data structure tells whether an element **may be in a set, or definitely is not**. The only possible errors are false positives, i.e., a search for a nonexistent element might give an incorrect answer. With more elements in the filter, the error rate increases. An empty Bloom filter is a bit-array of m bits, all set to 0. There are also k different hash functions, each of which maps a set element to one of the m bit positions.

- To add an element, feed it to the hash functions to get k bit positions, and set the bits at these positions to 1.
- To test if an element is in the set, feed it to the hash functions to get k bit positions.
 - If any of the bits at these positions is 0, the element is **definitely not** in the set.
 - If all are 1, then the element **may be** in the set.

Here is a Bloom filter with three elements P , Q , and R . It consists of 20 bits and uses three hash functions. The colored arrows point to the bits that the elements of the set are mapped to.



A Bloom filter consisting of 20 bits.

- The element x definitely is not in the set, since it hashes to a bit position containing 0.
- For a fixed error rate, adding a new element and testing for membership are both constant time operations, and a filter with room for 'n' elements requires $O(n)$ space.

Example: BigTable

In **BigTable** (and Cassandra), any read operation has to read from all SSTables that make up a Tablet. If these SSTables are not in memory, the read operation may end up doing many disk accesses. To reduce the number of disk accesses, BigTable uses Bloom filters.

Bloom filters are created for SSTables (particularly for the locality groups). They help reduce the number of disk accesses by predicting if an SSTable may contain data corresponding to a particular row or column pair. For

certain applications, a small amount of Tablet server memory used for storing Bloom filters drastically reduces the number of disk-seeks, thereby improving read performance.

[< Back](#)[Next >](#)

Introduction: System Design Patterns

2. Consistent Hashing

2. Consistent Hashing

Let's learn about Consistent Hashing and its usage.

We'll cover the following



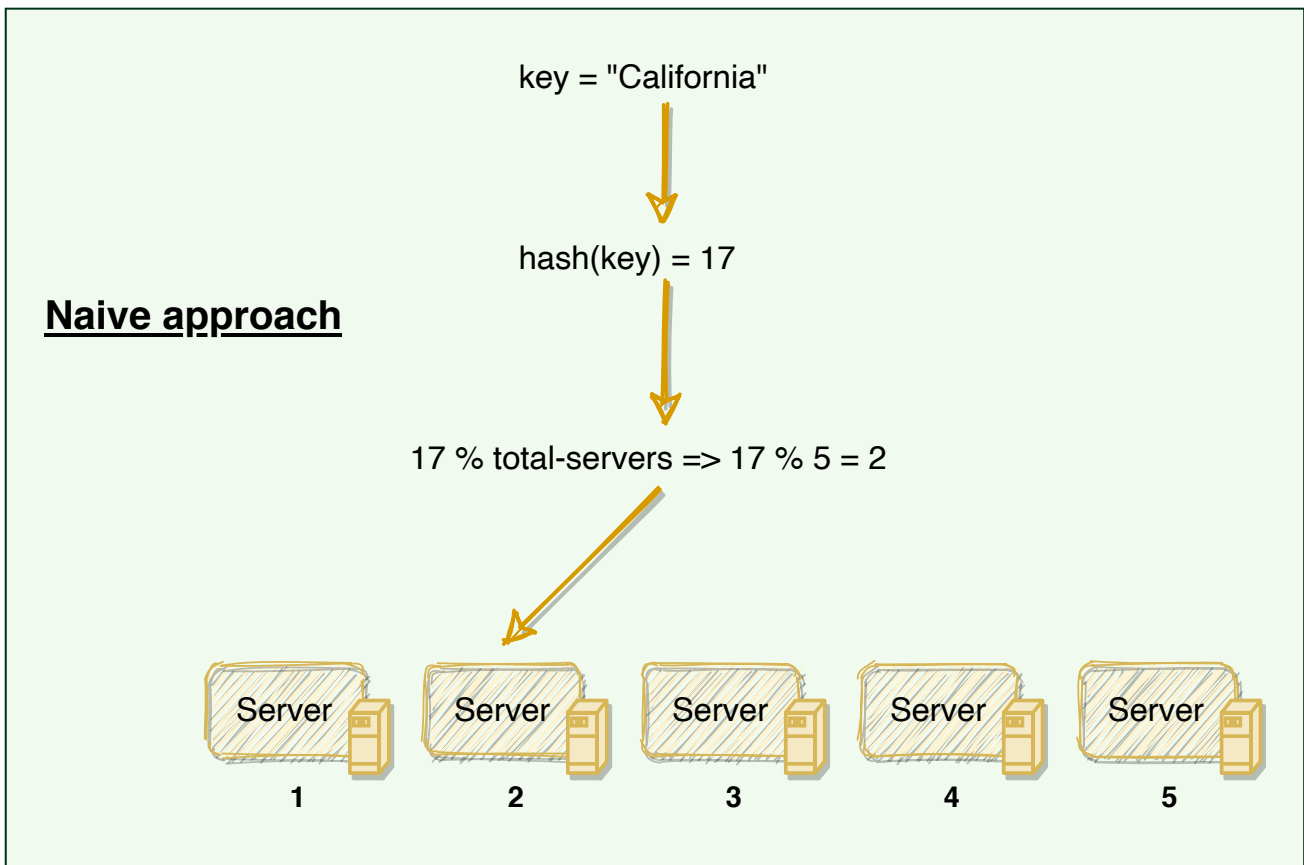
- Background
- Definition
- Solution
 - Virtual nodes
 - Advantages of Vnodes
- Examples

Background

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? Additionally, how can we minimize data movement when nodes join or leave?

A naive approach will 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:



Simple hashing

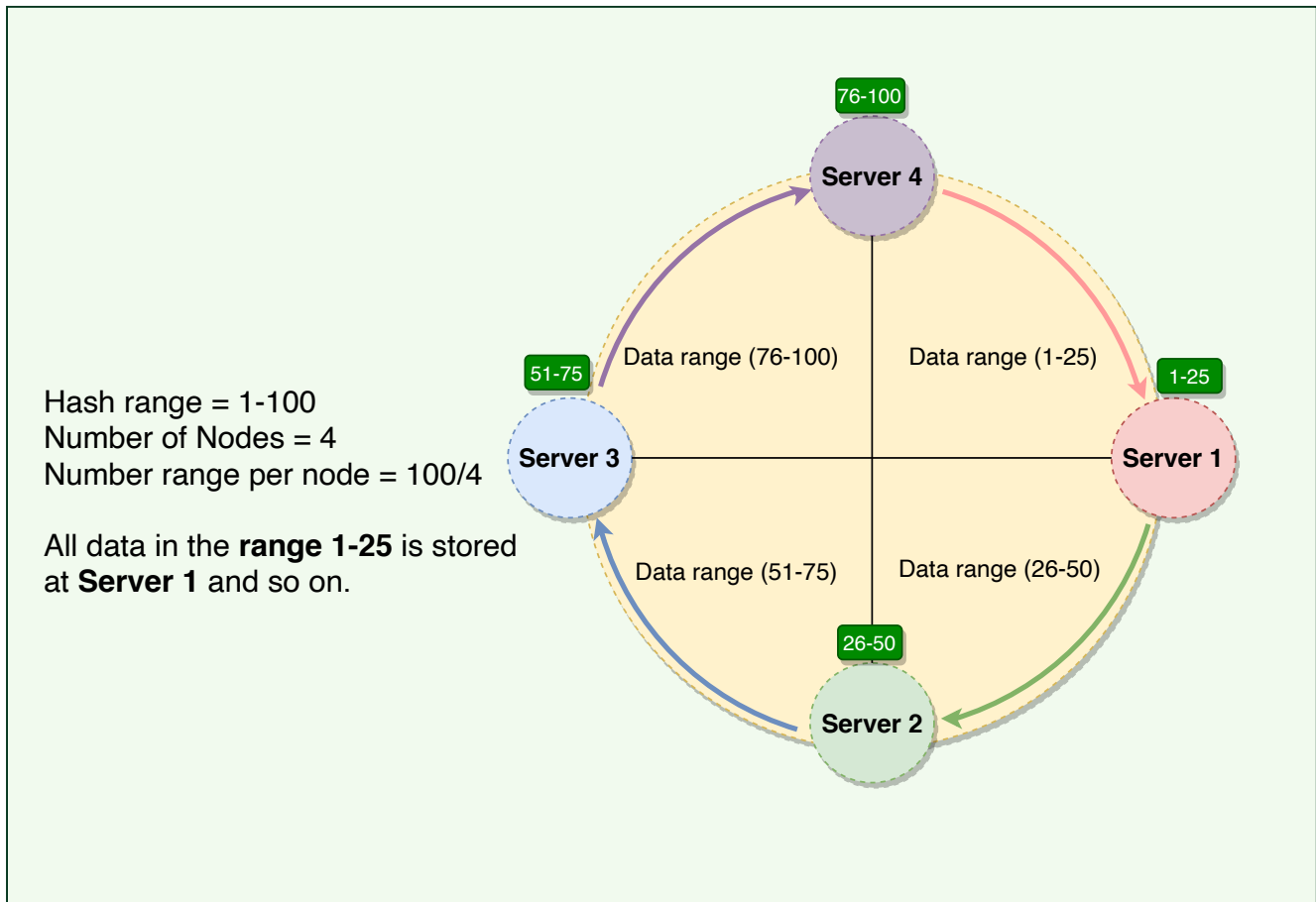
The scheme described in the above diagram solves the problem of finding a server for storing/retrieving the data. But when we add or remove a server, we have to remap all the keys and move our data based on the new server count, which will be a complete mess!

Definition

Use the Consistent Hashing algorithm to distribute data across nodes. Consistent Hashing maps data to physical nodes and ensures that **only a small set of keys move when servers are added or removed**.

Solution

Consistent Hashing technique stores the data managed by a distributed system in a ring. Each node in the ring is assigned a range of data. Here is an example of the consistent hash ring:



Consistent Hashing ring

With consistent hashing, the ring is divided into smaller, predefined ranges. Each node is assigned one of these ranges. 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:

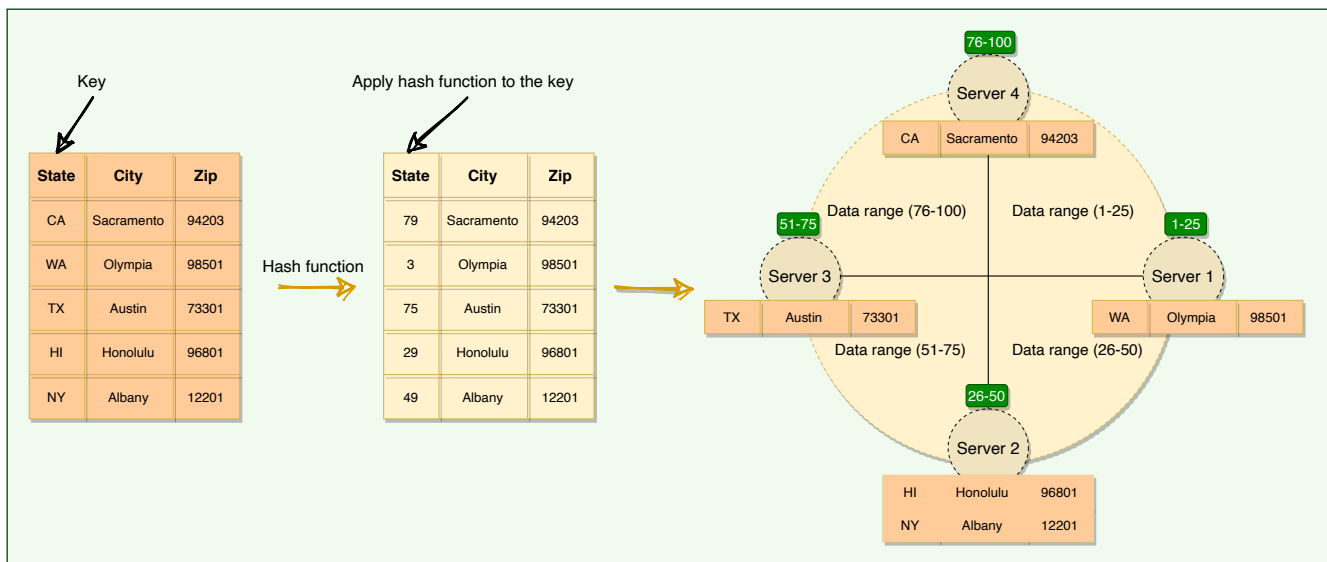
Range start: Token value

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 the system needs to read or write data, 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 is supposed to store data for a fixed range. Thus, the hash generated from the key tells us the node where the data will be stored.



Distributing data on the Consistent Hashing ring

The Consistent Hashing scheme above works great when a node is added or removed from the ring, as in these cases, since only the next node is affected. 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. This problem can be solved with the help of Virtual nodes.

Virtual nodes

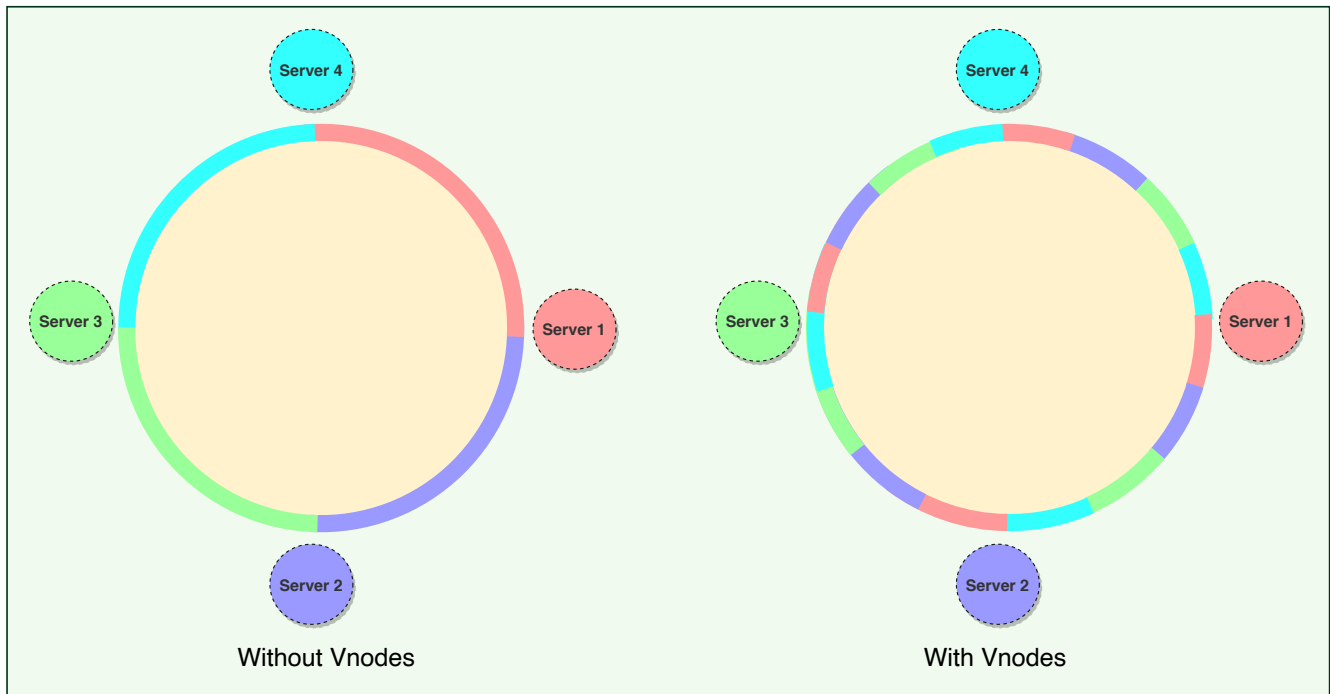
Adding and removing nodes in any distributed system is quite common. Existing nodes can die and may need to be decommissioned. Similarly, new nodes may be added to an existing cluster to meet growing demands. To efficiently handle these scenarios, Consistent Hashing makes use of virtual nodes (or Vnodes).

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 might be replicated (for fault-tolerance) on a fixed number of other nodes, 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.

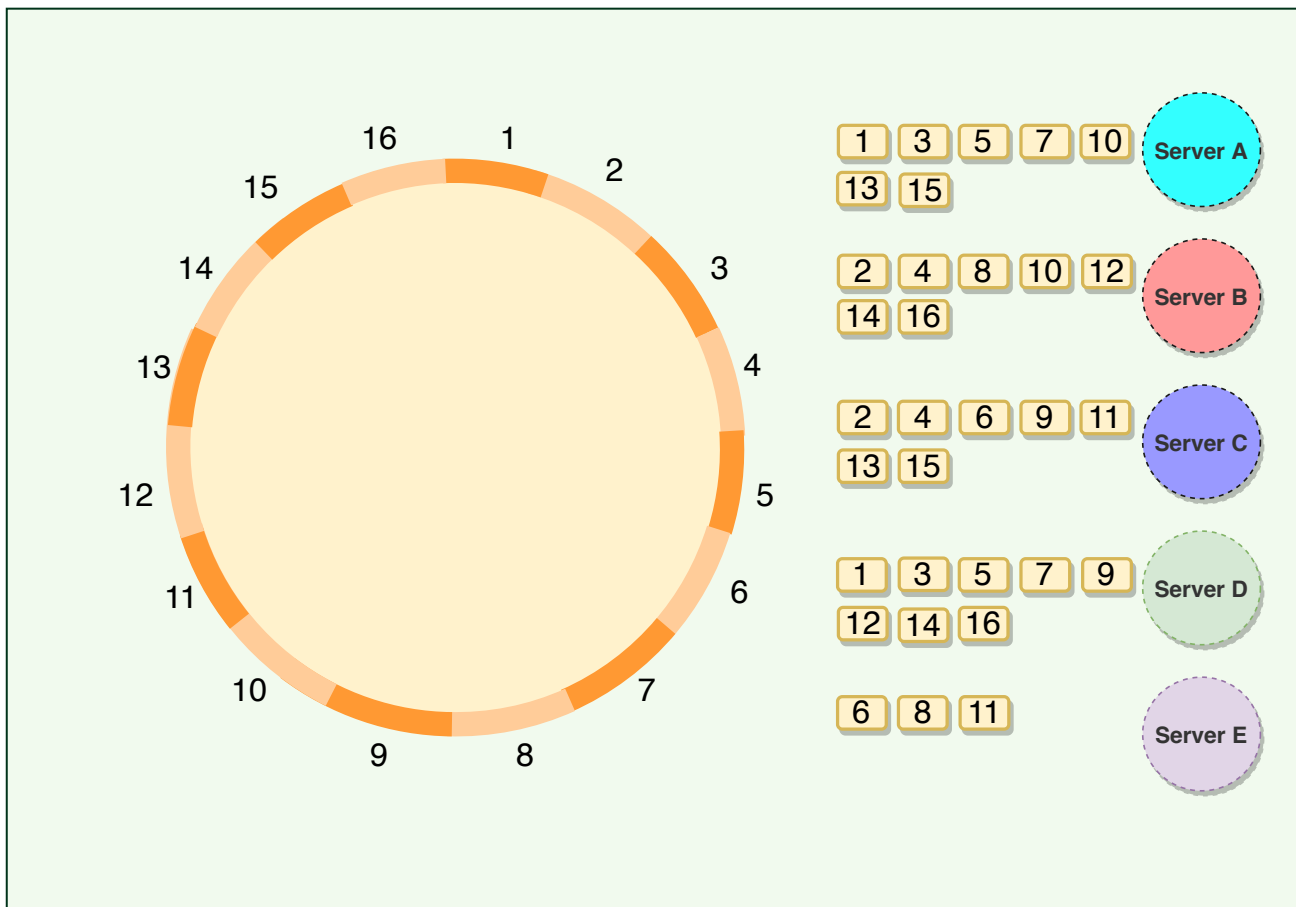
To handle these issues, Consistent Hashing introduces a new scheme of distributing the tokens to physical nodes. Instead of assigning a single token to a node, the hash range is divided into multiple smaller ranges, and each

physical node is assigned several of these smaller ranges. Each of these subranges is considered 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. Additionally, 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 gives the following advantages:

1. As 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 sub-ranges to a powerful server and a lower number of sub-ranges to a less powerful server.

3. In contrast to one big range, since Vnodes help assign smaller ranges to each physical node, this decreases the probability of hotspots.

Examples

Dynamo and **Cassandra** use Consistent Hashing to distribute their data across nodes.

← Back

1. Bloom Filters

Next →

3. Quorum

3. Quorum

Let's learn about Quorum and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In Distributed Systems, data is replicated across multiple servers for fault tolerance and high availability. Once a system decides to maintain multiple copies of data, another problem arises: how to make sure that all replicas are consistent, i.e., if they all have the latest copy of the data and that all clients see the same view of the data?

Definition

In a distributed environment, a quorum is the minimum number of servers on which a distributed operation needs to be performed successfully before declaring the operation's overall success.

Solution

Suppose a database is replicated on five machines. In that case, quorum refers to the minimum number of machines that perform the same action (commit or abort) for a given transaction in order to decide the final operation for that transaction. So, in a set of 5 machines, three machines form the majority quorum, and if they agree, we will commit that operation. Quorum enforces the consistency requirement needed for distributed operations.

In systems with multiple replicas, there is a possibility that the user reads inconsistent data. For example, when there are three replicas, R_1 , R_2 , and R_3 in a cluster, and a user writes value v_1 to replica R_1 . Then another user reads from replica R_2 or R_3 which are still behind R_1 and thus will not have the value v_1 , so the second user will not get the consistent state of data.

What value should we choose for a quorum? More than half of the number of nodes in the cluster: $(N/2 + 1)$ where N is the total number of nodes in the cluster, for example:

- In a 5-node cluster, three nodes must be online to have a majority.
- In a 4-node cluster, three nodes must be online to have a majority.
- With 5-node, the system can afford two node failures, whereas, with 4-node, it can afford only one node failure. Because of this logic, it is recommended to always have an odd number of total nodes in the cluster.

Quorum is achieved when nodes follow the below protocol: $R + W > N$, where:

N = nodes in the quorum group

W = minimum write nodes

R = minimum read nodes

If a distributed system follows $R + W > N$ rule, then every read will see at least one copy of the latest value written. For example, a common configuration could be (N=3, W=2, R=2) to ensure strong consistency. Here are a couple of other examples:

- (N=3, W=1, R=3): fast write, slow read, not very durable
- (N=3, W=3, R=1): slow write, fast read, durable

The following two things should be kept in mind before deciding read/write quorum:

- $R=1$ and $W=N \Rightarrow$ full replication (write-all, read-one): undesirable when servers can be unavailable because writes are not guaranteed to complete.
- Best performance (throughput/availability) when $1 < r < w < n$, because reads are more frequent than writes in most applications

Examples

- For leader election, **Chubby** uses Paxos, which use quorum to ensure strong consistency.
- As stated above, quorum is also used to ensure that at least one node receives the update in case of failures. For instance, in **Cassandra**, to ensure data consistency, each write request can be configured to be successful only if the data has been written to at least a quorum (or majority) of replica nodes.
- **Dynamo** replicates writes to a **sloppy quorum** of other nodes in the system, instead of a strict majority quorum like Paxos. 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 walking the consistent hashing ring.

4. Leader and Follower

Let's learn about leader and followers patterns and its usage in distributed systems.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

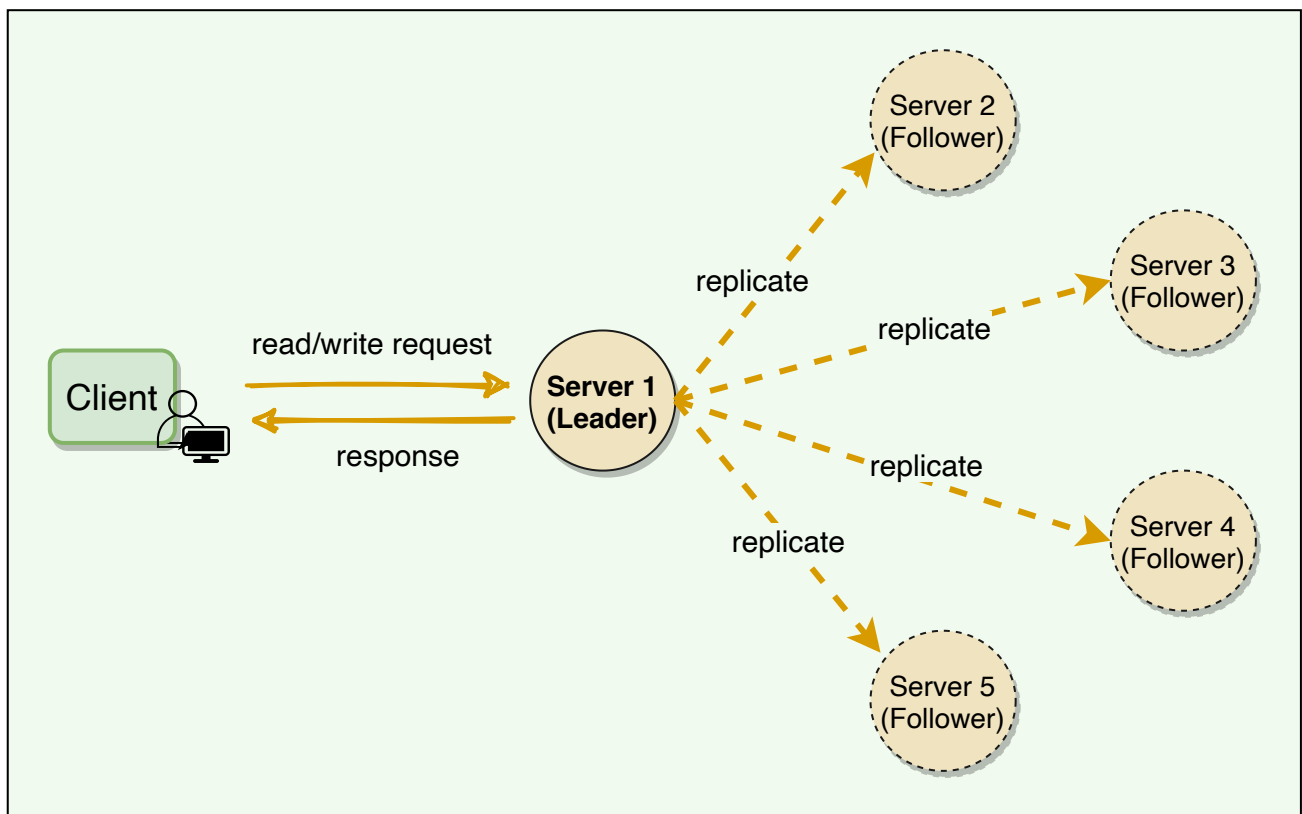
Distributed systems keep multiple copies of data for fault tolerance and higher availability. A system can use quorum to ensure data consistency between replicas, i.e., all reads and writes are not considered successful until a majority of nodes participate in the operation. However, using quorum can lead to another problem, that is, lower availability; at any time, the system needs to ensure that at least a majority of replicas are up and available, otherwise the operation will fail. Quorum is also not sufficient, as in certain failure scenarios, the client can still see inconsistent data.

Definition

Allow only a single server (called leader) to be responsible for data replication and to coordinate work.

Solution

At any time, one server is elected as the leader. This leader becomes responsible for data replication and can act as the central point for all coordination. The followers only accept writes from the leader and serve as a backup. In case the leader fails, one of the followers can become the leader. In some cases, the follower can serve read requests for load balancing.



Leader entertains requests from the client and is responsible for replicating and coordinating with followers

Examples

- In **Kafka**, each partition has a designated leader which is 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. This

provides redundancy of messages in a partition, so that a follower can take over the leadership if the leader goes down.

- Within the **Kafka** cluster, one broker is elected as the **Controller**. This Controller 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.
- To ensure strong consistency, **Paxos** (hence **Chubby**) performs leader election at startup. This leader is responsible for data replication and coordination.

← Back

3. Quorum

Next →

5. Write-ahead Log

5. Write-ahead Log

Let's learn about write-ahead logging and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

Machines can fail or restart anytime. If a program is in the middle of performing a data modification, what will happen when the machine it is running on loses power? When the machine restarts, the program might need to know the last thing it was doing. Based on its atomicity and durability needs, the program might need to decide to redo or undo or finish what it had started. How can the program know what it was doing before the system crash?

Definition

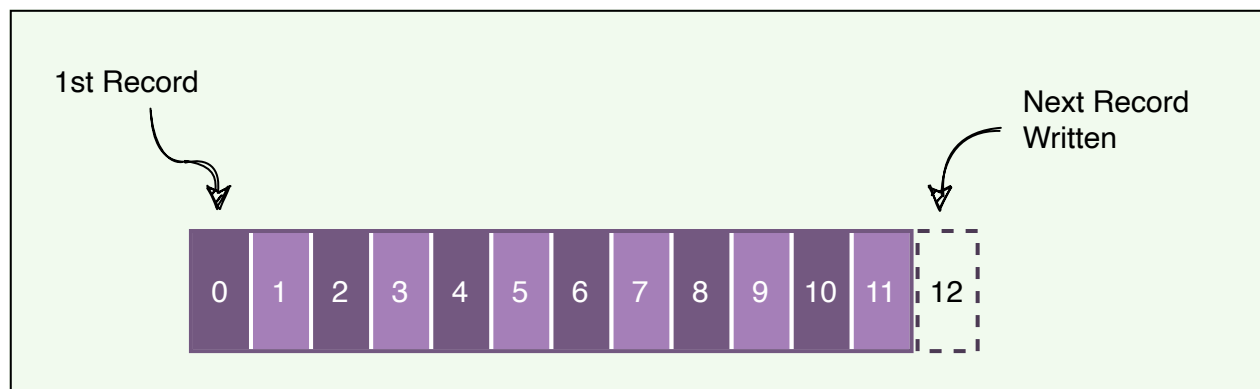
To guarantee durability and data integrity, each modification to the system is first written to an append-only log on the disk. This log is known as Write-Ahead Log (WAL) or transaction log or commit log. Writing to the WAL

guarantees that if the machine crashes, the system will be able to recover and reapply the operation if necessary.

Solution

The key idea behind the WAL is that all modifications before they are applied to the system are first written to a log file on the disk. Each log entry should contain enough information to redo or undo the modification. The log can be read on every restart to recover the previous state by replaying all the log entries. Using WAL results in a significantly reduced number of disk writes, because only the log file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction.

Each node, in a distributed environment, maintains its own log. WAL is always sequentially appended, which simplifies the handling of the log. Each log entry is given a unique identifier; this identifier helps in implementing certain other operations like **log segmentation**(discussed later) or **log purging**.



Write-ahead log

Examples

- **Cassandra:** To ensure durability, whenever a node receives a write request, it immediately writes the data to a commit log which is a WAL. Cassandra, before writing data to a MemTable, first writes it to the commit log. This provides durability in the case of an unexpected shutdown. On startup, any mutations in the commit log will be applied to MemTables.
- **Kafka** implements a distributed Commit Log to persistently store all messages it receives.
- **Chubby:** For fault tolerance and in the event of a leader crash, all database transactions are stored in a transaction log which is a WAL.

← Back

4. Leader and Follower

Next →

6. Segmented Log

6. Segmented Log

Let's learn about segmented log and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

A single log can become difficult to manage. As the file grows, it can also become a performance bottleneck, especially when it is read at the startup. Older logs need to be cleaned up periodically or, in some cases, merged. Doing these operations on a single large file is difficult to implement.

Definition

Break down the log into smaller segments for easier management.

Solution

A single log file is split into multiple parts, such that the log data is divided into equal-sized log segments. The system can roll the log based on a rolling policy - either a configurable period of time (e.g., every 4 hours) or a configurable maximum size (e.g., every 1GB).

Examples

- **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 in size and reaches its threshold in size, a new commit log is created. Hence, over time, several commit logs will exist, 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. A commit log segment can be **archived**, **deleted**, or **recycled** once all its data has been flushed to SSTables.
- **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.

← Back

Next →

5. Write-ahead Log

7. High-Water Mark

7. High-Water Mark

Let's learn about high-water mark and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

Distributed systems keep multiple copies of data for fault tolerance and higher availability. To achieve strong consistency, one of the options is to use a leader-follower setup, where the leader is responsible for entertaining all the writes, and the followers replicate data from the leader.

Each transaction on the leader is committed to a write-ahead log (WAL), so that the leader can recover from crashes or failures. A write request is considered successful as soon as it is committed to the WAL on the leader. The replication can happen asynchronously; either the leader can push the mutation to the followers, or the follower can pull it from the leader. In case the leader crashes and cannot recover, one of the followers will be elected as the new leader. Now, this new leader can be a bit behind the old leader, as there might be some transactions that have not been completely propagated before the old leader crashed. We do have these transactions in the WAL on the old leader, but those log entries cannot be recovered until the old leader

becomes alive again. So those transactions are considered lost. Under this scenario, the client can see some data inconsistencies, e.g., the last data that the client fetched from the old leader may not be available anymore. In such error scenarios, some followers can be missing entries in their logs, and some can have more entries than others. So, it becomes important for the leader and followers to know what part of the log is safe to be exposed to the clients.

Definition

Keep track of the last log entry on the leader, which has been successfully replicated to a quorum of followers. The index of this entry in the log is known as the High-Water Mark index. The leader exposes data only up to the high-water mark index.

Solution

For each data mutation, the leader first appends it to WAL and then sends it to all the followers. Upon receiving the request, the followers append it to their respective WAL and then send an acknowledgment to the leader. The leader keeps track of the indexes of the entries that have been successfully replicated on each follower. The high-water mark index is the highest index, which has been replicated on the quorum of the followers. The leader can propagate the high-water mark index to all followers as part of the regular Heartbeat message. The leader and followers ensure that the client can read data only up to the high-water mark index. This guarantees that even if the current leader fails and another leader is elected, the client will not see any data inconsistencies.

Examples

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

← Back

Next →

6. Segmented Log

8. Lease

8. Lease

Let's learn about lease and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In distributed systems, a lot of times clients need specified rights to certain resources. For example, a client might need exclusive rights to update the contents of a file. One way to fulfill this requirement is through distributed locking. A client first gets an exclusive (or write) lock associated with the file and then proceeds with updating the file. One problem with locking is that the lock is granted until the locking client explicitly releases it. If the client fails to release the lock due to any reason, e.g., process crash, deadlock, or a software bug, the resource will be locked indefinitely. This leads to resource unavailability until the system is reset. Is there an alternate solution?

Definition

Use time-bound leases to grant clients rights on resources.

Solution

A lease is like a lock, but it works even when the client goes away. The client asks for a lease for a limited period of time, after which the lease expires. If the client wants to extend the lease, it can renew the lease before it expires.

Examples

Chubby clients maintain a time-bound session lease with the leader. During this time interval, the leader guarantees to not terminate the session unilaterally.

[< Back](#)

7. High-Water Mark

[Next >](#)

9. Heartbeat

9. Heartbeat

Let's learn about heartbeat and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In a distributed environment, work/data is distributed among servers. To efficiently route requests in such a setup, servers need to know what other servers are part of the system. Furthermore, servers should know if other servers are alive and working. In a decentralized system, whenever a request arrives at a server, the server should have enough information to decide which server is responsible for entertaining that request. This makes the timely detection of server failure an important task, which also enables the system to take corrective actions and move the data/work to another healthy server and stop the environment from further deterioration.

Definition

Each server periodically sends a heartbeat message to a central monitoring server or other servers in the system to show that it is still alive and functioning.

Solution

Heartbeating is one of the mechanisms for detecting failures in a distributed system. If there is a central server, all servers periodically send a heartbeat message to it. If there is no central server, all servers randomly choose a set of servers and send them a heartbeat message every few seconds. This way, if no heartbeat message is received from a server for a while, the system can suspect that the server might have crashed. If there is no heartbeat within a configured timeout period, the system can conclude that the server is not alive anymore and stop sending requests to it and start working on its replacement.

Examples

- **GFS:** The leader periodically communicates with each ChunkServer in HeartBeat messages to give instructions and collect state.
- **HDFS:** The NameNode keeps track of DataNodes through a **heartbeat** mechanism. Each DataNode sends periodic heartbeat messages (every few seconds) to the NameNode. If a DataNode dies, then the heartbeats to the NameNode are stopped. The NameNode detects that a DataNode has died if the number of missed heartbeat messages reaches a certain threshold. The NameNode then marks the DataNode as dead and will no longer forward any I/O requests to that DataNode.

10. Gossip Protocol

Let's learn about gossip protocol and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In a large distributed environment where 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 a heartbeat with every other node. Then, when a node goes down, it will stop sending out heartbeats, and everyone else will find out immediately. But, this means $O(N^2)$ messages get sent every tick (N being the total number of nodes), which is a ridiculously high amount and will consume a lot of network bandwidth, and thus, not feasible in any sizable cluster. So, is there any other option for monitoring the state of the cluster?

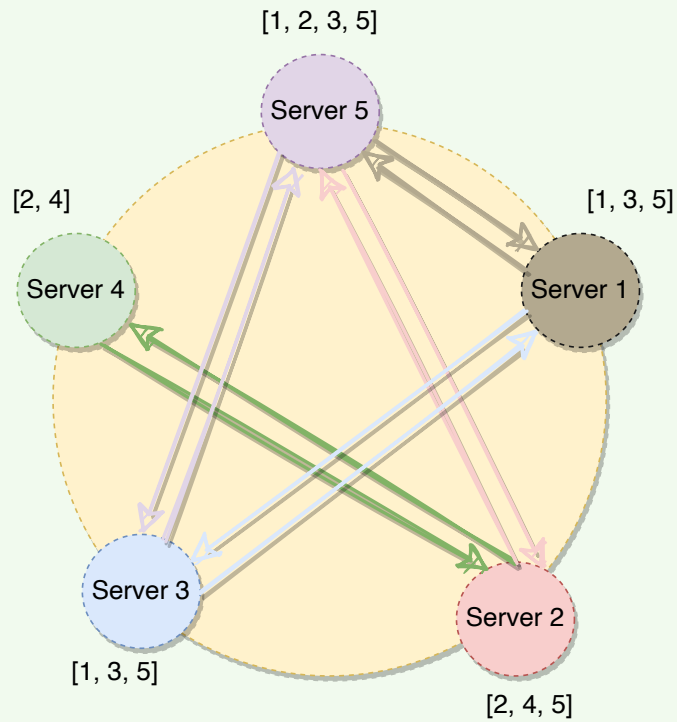
Definition

Each node keeps track of state information about other nodes in the cluster and gossip (i.e., share) this information to one other random node every second. This way, eventually, each node gets to know about the state of every other node in the cluster.

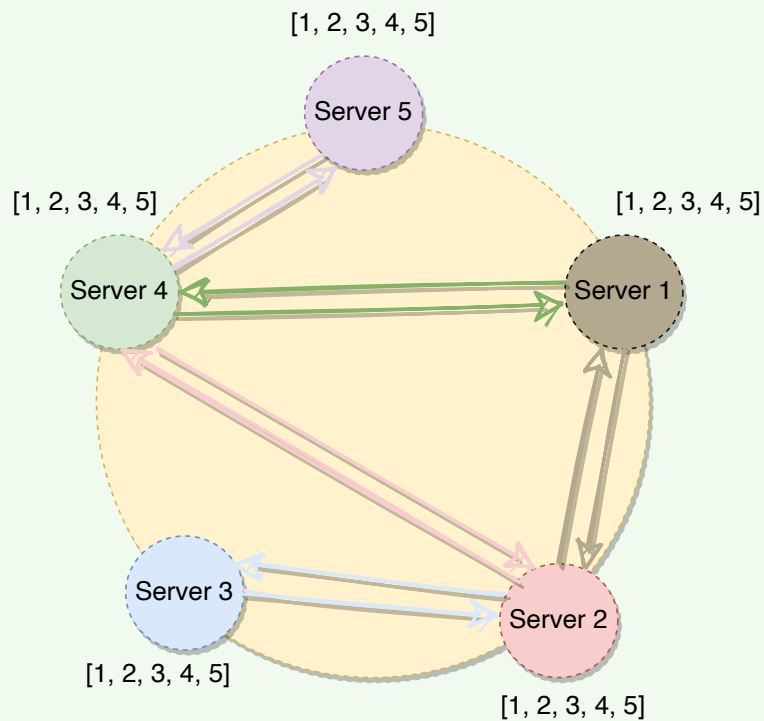
Solution

Gossip protocol is a peer-to-peer communication mechanism in which nodes periodically exchange state information about themselves and about other nodes they know about. Each node initiates a gossip round every second to exchange state information about themselves and other nodes with one other random node. This means that any state change 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



Examples

Dynamo & Cassandra use gossip protocol which allows 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, etc.

← Back

Next →

9. Heartbeat

11. Phi Accrual Failure Detection

11. Phi Accrual Failure Detection

Let's learn about Phi Accrual Failure Detection algorithm and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In distributed systems, accurately detecting failures is a hard problem to solve, as we cannot say with 100% surety if a system is genuinely down or is just very slow in responding due to heavy load, network congestion, etc. Conventional failure detection 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 detect failures quickly but with many false positives due to slow machines or faulty network. 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.

Definition

Use adaptive failure detection algorithm as described by Phi Accrual Failure Detector. Accrual means accumulation or the act of accumulating over time. This algorithm uses historical heartbeat information to make the threshold adaptive. Instead of telling if the server is alive or not, a generic Accrual Failure Detector outputs the suspicion level about a server. A higher suspicion level means there are higher chances that the server is down.

Solution

With 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 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.

Examples

Cassandra uses the Phi Accrual Failure Detector algorithm to determine the state of the nodes in the cluster.

[!\[\]\(a03a7eb2f4046e1d3c76772003e549ea_img.jpg\) Back](#)[Next !\[\]\(cbe2492b119e39e02a1dab2af4a4b296_img.jpg\)](#)

10. Gossip Protocol

12. Split Brain

12. Split Brain

Let's learn about split-brain and how to handle it.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In a distributed environment with a central (or leader) server, if the central server dies, the system must quickly find a substitute, otherwise, the system can quickly deteriorate.

One of the problems is that we cannot truly know if the leader has stopped for good or 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 leader. If the original leader had an intermittent failure, we now find ourselves with a so-called **zombie leader**. A zombie leader can be defined as a leader node that had been deemed dead by the system and has since come back online. Another node has taken its place, but the zombie leader might not know that yet. The system now has two active leaders that could be issuing conflicting commands. How can a system

detect such a scenario, so that all nodes in the system can ignore requests from the old leader and the old leader itself can detect that it is no longer the leader?

Definition

The common scenario in which a distributed system has two or more active leaders is called split-brain.

Split-brain is solved through the use of **Generation Clock**, which is simply a monotonically increasing number to indicate a server's generation.

Solution

Every time a new leader is elected, the generation number gets incremented. This means if the old leader had a generation number of '1', the new one will have '2'. This generation number is included in every request that is sent from the leader to other nodes. This way, nodes can now easily differentiate the real leader by simply trusting the leader with the highest number. The generation number should be persisted on disk, so that it remains available after a server reboot. One way is to store it with every entry in the Write-ahead Log.

Examples

Kafka: To handle Split-brain (where we could have multiple active controller brokers), Kafka uses '**Epoch number**,' which is simply a monotonically increasing number to indicate a server's generation.

HDFS: ZooKeeper is used to ensure that only one NameNode is active at any time. An epoch number is maintained as part of every transaction ID to reflect the NameNode generation.

Cassandra uses generation number to distinguish a node's state before and after a restart. Each node stores a generation number which is incremented every time a node restarts. This generation number is included in gossip messages exchanged between nodes and is used to distinguish the current state of a node from the 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 generation number in the gossip message. If the generation number in the gossip message is higher, it knows that the node was restarted.

[← Back](#)

11. Phi Accrual Failure Detection

[Next →](#)

13. Fencing

13. Fencing

Let's learn about fencing and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In a leader-follower setup, when a leader fails, it is impossible to be sure that the leader has stopped working. For example, a slow network or a network partition can trigger a new leader election, even though the previous leader is still running and thinks it is still the active leader. Now, in this situation, if the system elects a new leader, how do we make sure that the old leader is not running and possibly issuing conflicting commands?

Definition

Put a 'Fence' around the previous leader to prevent it from doing any damage or causing corruption.

Solution

Fencing is the idea of putting a fence around a previously active leader so that it cannot access cluster resources and hence stop serving any read/write request. The following two techniques are used:

- **Resource fencing:** Under this scheme, the system blocks the previously active leader from accessing resources needed to perform essential tasks. For example, revoking its access to the shared storage directory (typically by using a vendor-specific Network File System (NFS) command), or disabling its network port via a remote management command.
- **Node fencing:** Under this scheme, the system stops the previously active leader from accessing all resources. A common way of doing this is to power off or reset the node. This is a very effective method of keeping it from accessing anything at all. This technique is also called STONIT or “Shoot The Other Node In The Head.”

Examples

HDFS uses fencing to stop the previously active NameNode from accessing cluster resources, thereby stopping it from servicing requests.

← Back

12. Split Brain

Next →

14. Checksum

14. Checksum

Let's learn about checksum and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In a distributed system, while moving data between components, it is possible that the data fetched from a node may arrive corrupted. This corruption can occur because of faults in a storage device, network, software, etc. How can a distributed system ensure data integrity, so that the client receives an error instead of corrupt data?

Definition

Calculate a checksum and store it with data.

Solution

When a system is storing some data, it computes a checksum of the data, and stores the checksum with the data. When a client retrieves data, it verifies that the data it received from the server matches the checksum stored. If not, then the client can opt to retrieve that data from another replica.

Examples

HDFS and **Chubby** store the checksum of each file with the data.

← Back

13. Fencing

Next →

15. Vector Clocks

15. Vector Clocks

Let's learn about vector clocks and their usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

When a distributed system allows concurrent writes, it can result in multiple versions of an object. Different replicas of an object can end up with different versions of the data. Let's understand this with an example.

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 true. The problem is **clock skew** – 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.

So how can we reconcile and capture causality between different versions of the same object?

Definition

Use Vector clocks to keep track of value history and reconcile divergent histories at read time.

Solution

A vector clock is effectively a (node, counter) pair. One vector clock is associated with every version of every object. 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. Such conflicts are resolved at read-time, and if the system is not able to reconcile an object's state from its vector clocks, it sends it to the client application for reconciliation (since clients have more semantic information on the object and may be able to reconcile it). 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.

To see how Dynamo handles conflicting data, take a look at [Vector Clocks and Conflicting Data \(/02.Dynamo_How_Design_Key-value_Store/Vector_Clocks_and_Conflicting_Data - Grokking the Advanced System Design Interview.html\)](#)

Examples

To reconcile concurrent updates on an object Amazon's **Dynamo** uses Vector Clocks.

← Back

14. Checksum

Next →

16. CAP Theorem

16. CAP Theorem

Let's learn about the CAP theorem and its usage in distributed systems.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In distributed systems, different types of failures can occur, e.g., servers can crash or fail permanently, disks can go bad resulting in data losses, or network connection can be lost, making a part of the system inaccessible. How can a distributed system model itself to get the maximum benefits out of different resources available?

Definition

CAP theorem states that it is **impossible** for a distributed system to simultaneously provide all three of the following desirable properties:

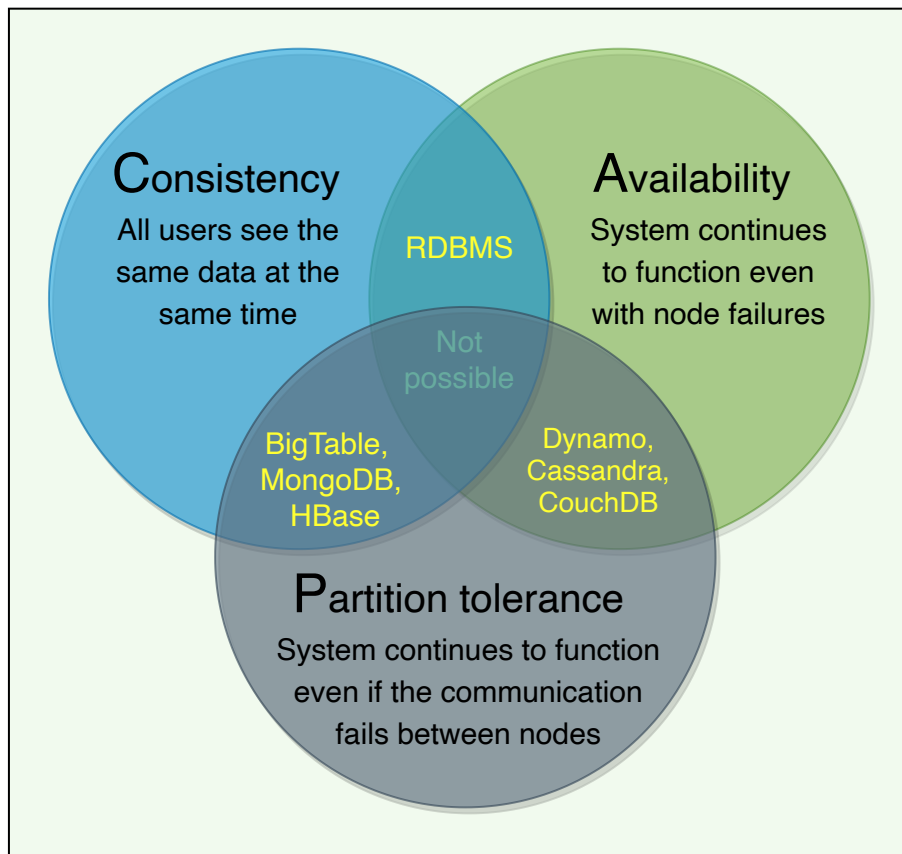
Consistency (C): All nodes see the same data at the same time. It is equivalent to having a single up-to-date copy of the data.

Availability (A): Every request received by a non-failing node in the system must result in a response. Even when severe network failures occur, every request must terminate.

Partition tolerance (P): A partition-tolerant system continues to operate despite partial system failure or arbitrary message loss. Such a system can sustain any network failure that does not result in a failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

Solution

According to the CAP theorem, any distributed system needs to pick two out of the three properties. The three options are CA, CP, and AP. However, CA is not really a coherent option, as a system that is not partition-tolerant will be forced to give up either Consistency or Availability in the case of a network partition. Therefore, the theorem can really be stated as: **In the presence of a network partition, a distributed system must choose either Consistency or Availability.**



Examples

Dynamo: In CAP theorem terms, Dynamo falls within the category of AP systems and is designed for **high availability** 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.

BigTable: In terms of the CAP theorem, BigTable is a CP system, i.e., it **has strictly consistent reads and writes**.

← Back

Next →

17. PACELC Theorem

Let's learn about PACELC theorem and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

We cannot avoid partition in a distributed system, therefore, according to the CAP theorem, a distributed system should choose between consistency or availability. ACID (Atomicity, Consistency, Isolation, Durability) databases chose consistency (refuse response if it cannot check with peers), while BASE (Basically Available, Soft-state, Eventually consistent) databases chose availability (respond with local data without ensuring it is the latest with its peers). One place where the CAP theorem is silent is what happens when there is no network partition? What choices does a distributed system have when there is no partition?

Definition

The PACELC theorem states that in a system that replicates data:

- if there is a partition ('P'), a distributed system can tradeoff between availability and consistency (i.e., 'A' and 'C');
- else ('E'), when the system is running normally in the absence of partitions, the system can tradeoff between latency ('L') and consistency ('C').

Solution

The first part of the theorem (PAC) is the same as the CAP theorem, and the ELC is the extension. The whole thesis is assuming we maintain high availability by replication. So, when there is a failure, CAP theorem prevails. But if not, we still have to consider the tradeoff between consistency and latency of a replicated system.

Examples

- **Dynamo and Cassandra** are PA/EL systems: They choose availability over consistency when a partition occurs; otherwise, they choose lower latency.
- **BigTable and HBase** are PC/EC systems: They will always choose consistency, giving up availability and lower latency.
- **MongoDB** is PA/EC: In case of a network partition, it chooses availability, but otherwise guarantees consistency.

← Back

Next →

16. CAP Theorem

18. Hinted Handoff

18. Hinted Handoff

Let's learn about hinted handoff and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

Depending upon the consistency level, a distributed system 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, the system can still write on the remaining two nodes to fulfill the consistency level, making the write successful. Now, when the node which was down comes online again, how should we write data to it?

Definition

For nodes that are down, the system keeps notes (or hints) of all the write requests they have missed. Once the failing nodes recover, the write requests are forwarded to them based on the hints stored.

Solution

When a node is down or is not responding to a write request, the node which is coordinating the operation, 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 coordinating node discovers that a node for which it holds hints has recovered, it forwards the write requests for each hint to the target.

Examples

- **Cassandra** nodes use Hinted Handoff to remember the write operation for failing nodes.
- **Dynamo** ensures that the system is “always-writeable” by using Hinted Handoff (and Sloppy Quorum).

[← Back](#)

17. PACELC Theorem

[Next →](#)

19. Read Repair

19. Read Repair

Let's learn about read repair and its usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

In Distributed Systems, where data is replicated across multiple nodes, some nodes can end up having stale data. Imagine a scenario where a node failed to receive a write or update request because it was down or there was a network partition. How do we ensure that the node gets the latest version of the data when it is healthy again?

Definition

Repair stale data during the read operation, since at that point, we can read data from multiple nodes to perform a comparison and find nodes that have stale data. This mechanism is called Read Repair. Once the node with old data is known, the read repair operation pushes the newer version of data to nodes with the older version.

Solution

Based on the quorum, the system reads data from multiple nodes. For example, for Quorum=2, the system reads data from one node and digest of the data from the second 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 system reads the data from all the replicas to find the latest data. The system returns the latest data to the client and initiates a **Read Repair** request. The read repair operation pushes the latest version of data to nodes with the older version.

When the read consistency level is less than 'All,' some systems perform a read repair probabilistically, for example, 10% of the requests. In this case, the system immediately sends a response to the client when the consistency level is met and performs the read repair asynchronously in the background.

Examples

Cassandra and **Dynamo** use 'Read Repair' to push the latest version of the data to nodes with the older versions.

[< Back](#)

18. Hinted Handoff

[Next >](#)

20. Merkle Trees

20. Merkle Trees

Let's learn about Merkle trees and their usage.

We'll cover the following



- Background
- Definition
- Solution
- Examples

Background

As we saw in the previous lesson

(<https://www.educative.io/collection/page/5668639101419520/5559029852536832/5978407785988096>), Read Repair removes conflicts while serving read requests. But, if a replica falls significantly behind others, it might take a very long time to resolve conflicts. 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 and figure out exactly which parts are different. In a distributed environment, how can we quickly compare two copies of a range of data residing on two different replicas and figure out exactly which parts are different?

Definition

A replica can contain a lot of data. Naively splitting up the entire range to calculate checksums for comparison, is not very feasible; there is simply too much data to be transferred. Instead, we can use Merkle trees to compare replicas of a range.

Solution

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 exactly which parts of the range are different, but the amount of data exchanged is minimized. The principal advantage of a Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Hence, Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads.

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

Examples

For anti-entropy and to resolve conflicts in the background, **Dynamo** uses Merkle trees.

← Back

Next →

19. Read Repair

Quiz I