- Node *s0* maintains a node membership list shown on the left side.
- Node *s0* notices that node s2's (member ID = 2) heartbeat counter has not increased for a long time.
- Node *s0* sends heartbeats that include *s2*'s info to a set of random nodes. Once other nodes confirm that *s2*'s heartbeat counter has not been updated for a long time, node *s2* is marked down, and this information is propagated to other nodes.

Handling temporary failures

After failures have been detected through the gossip protocol, the system needs to deploy certain mechanisms to ensure availability. In the strict quorum approach, read and write operations could be blocked as illustrated in the quorum consensus section.

A technique called "sloppy quorum" [4] is used to improve availability. Instead of enforcing the quorum requirement, the system chooses the first *W* healthy servers for writes and first *R* healthy servers for reads on the hash ring. Offline servers are ignored.

If a server is unavailable due to network or server failures, another server will process requests temporarily. When the down server is up, changes will be pushed back to achieve data consistency. This process is called hinted handoff. Since *s2* is unavailable in Figure 6-12, reads and writes will be handled by *s3* temporarily. When *s2* comes back online, *s3* will hand the data back to *s2*.

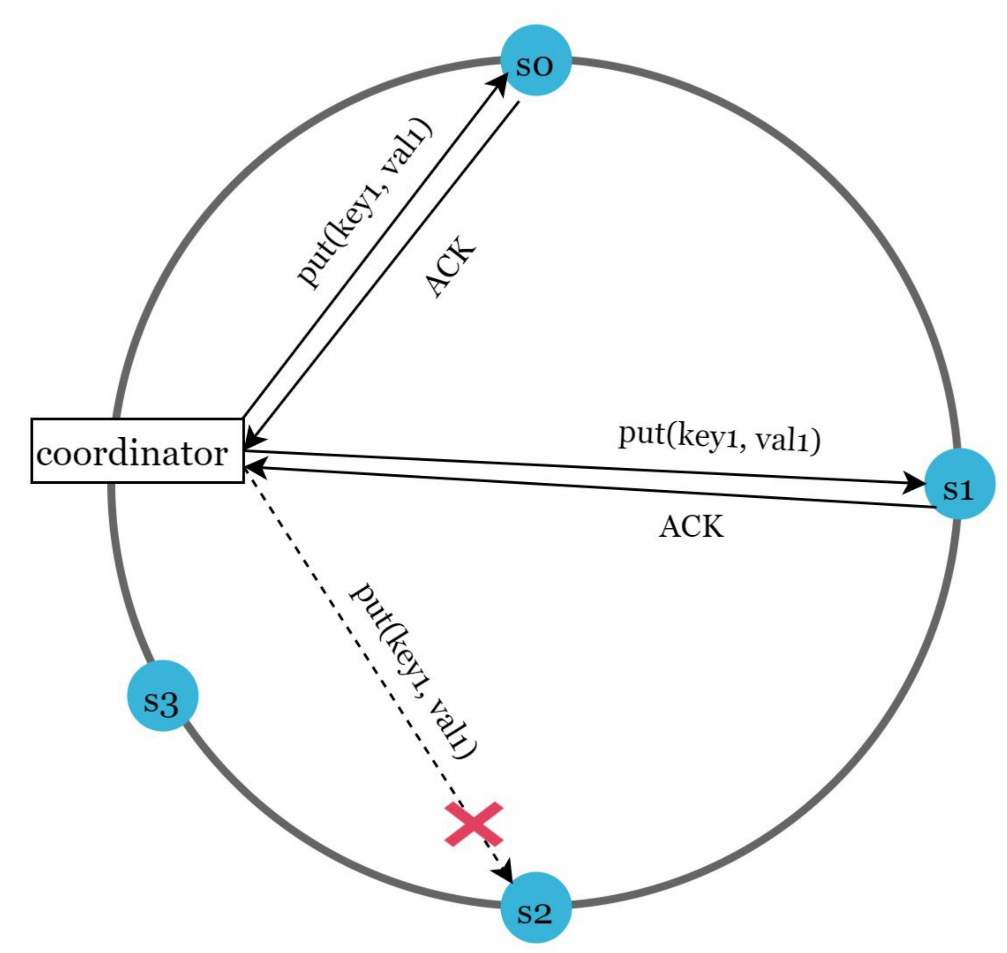


Figure 6-12

Handling permanent failures

Hinted handoff is used to handle temporary failures. What if a replica is permanently unavailable? To handle such a situation, we implement an anti-entropy protocol to keep replicas in sync. Anti-entropy involves comparing each piece of data on replicas and updating each replica to the newest version. A Merkle tree is used for inconsistency detection and minimizing the amount of data transferred.

Quoted from Wikipedia [7]: “A hash tree or Merkle tree is a tree in which every non-leaf node is labeled with the hash of the labels or values (in case of leaves) of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures”.

Assuming key space is from 1 to 12, the following steps show how to build a Merkle tree. Highlighted boxes indicate inconsistency.

Step 1: Divide key space into buckets (4 in our example) as shown in Figure 6-13. A bucket is used as the root level node to maintain a limited depth of the tree.

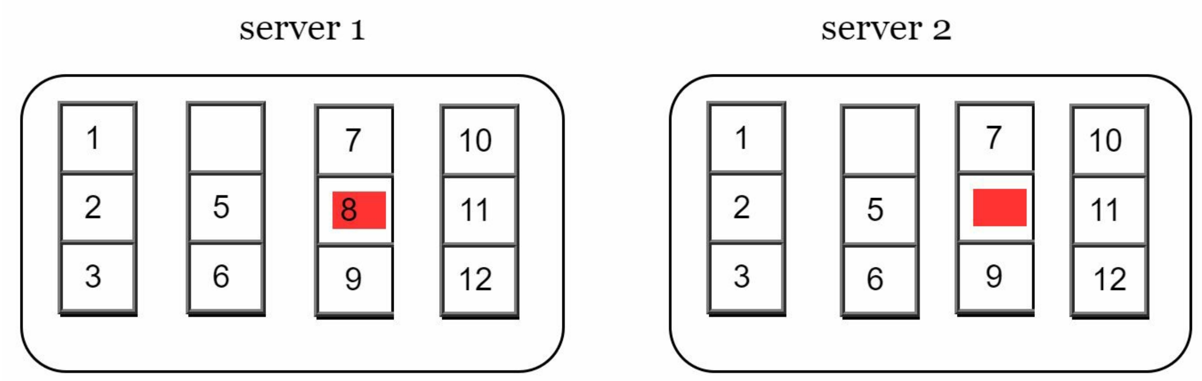


Figure 6-13

Step 2: Once the buckets are created, hash each key in a bucket using a uniform hashing method (Figure 6-14).

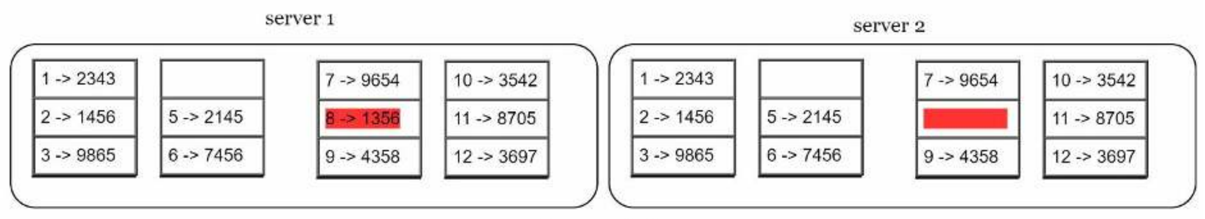


Figure 6-14

Step 3: Create a single hash node per bucket (Figure 6-15).

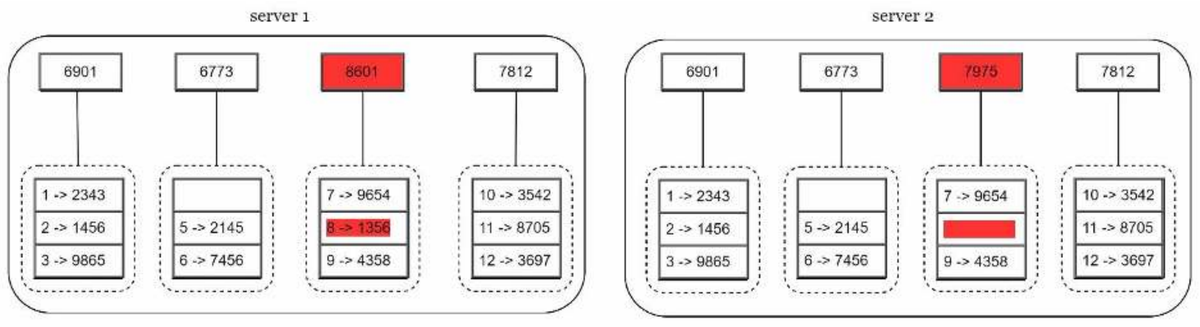


Figure 6-15

Step 4: Build the tree upwards till root by calculating hashes of children (Figure 6-16).

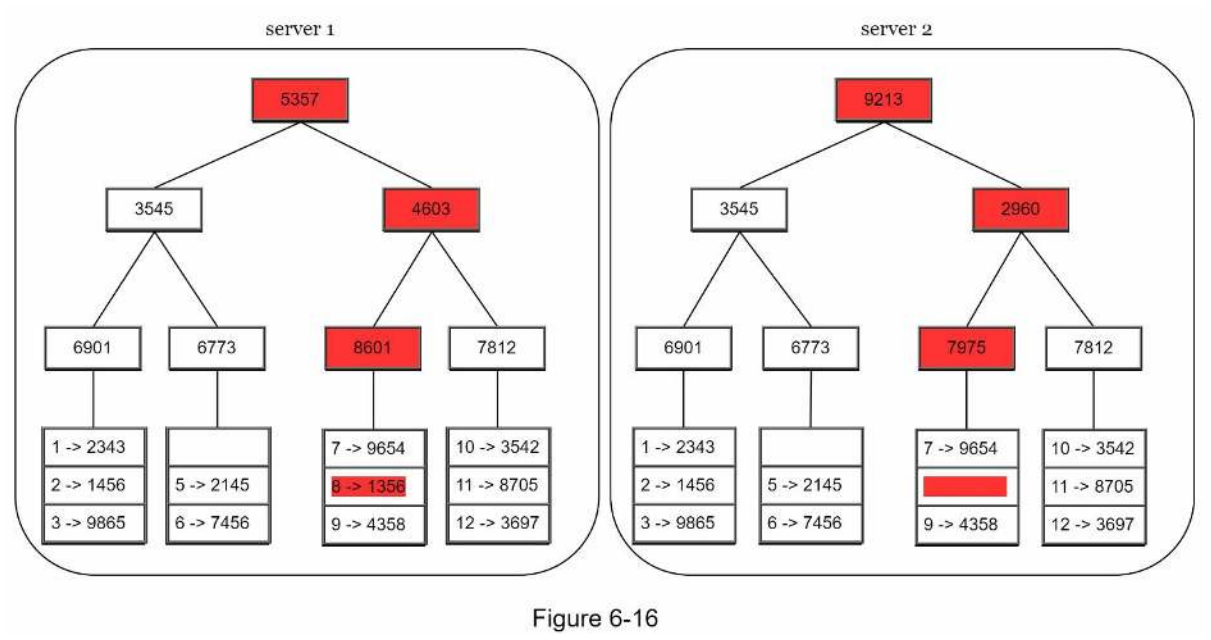


Figure 6-16

To compare two Merkle trees, start by comparing the root hashes. If root hashes match, both servers have the same data. If root hashes disagree, then the left child hashes are compared followed by right child hashes. You can traverse the tree to find which buckets are not synchronized and synchronize those buckets only.

Using Merkle trees, the amount of data needed to be synchronized is proportional to the differences between the two replicas, and not the amount of data they contain. In real-world systems, the bucket size is quite big. For instance, a possible configuration is one million buckets per one billion keys, so each bucket only contains 1000 keys.

Handling data center outage

Data center outage could happen due to power outage, network outage, natural disaster, etc. To build a system capable of handling data center outage, it is important to replicate data across multiple data centers. Even if a data center is completely offline, users can still access data through the other data centers.

System architecture diagram

Now that we have discussed different technical considerations in designing a key-value store, we can shift our focus on the architecture diagram, shown in Figure 6-17.

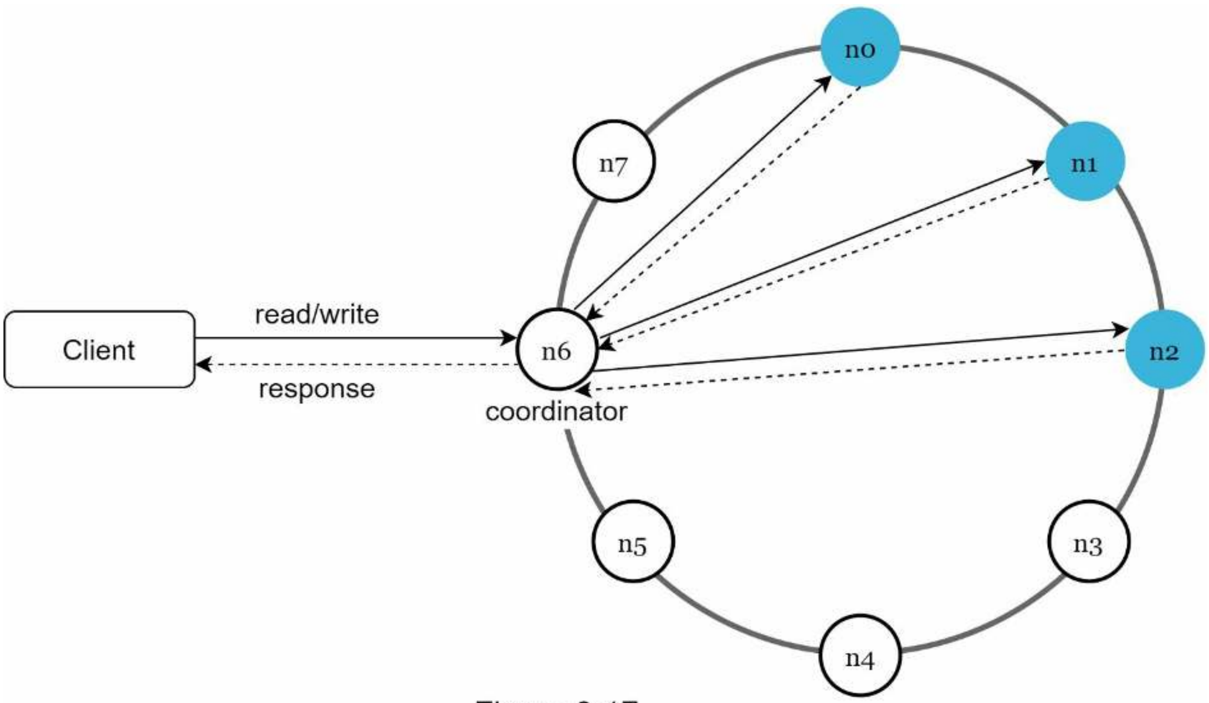


Figure 6-17

Main features of the architecture are listed as follows:

- Clients communicate with the key-value store through simple APIs: *get(key)* and *put(key, value)*.
- A coordinator is a node that acts as a proxy between the client and the key-value store.
- Nodes are distributed on a ring using consistent hashing.
- The system is completely decentralized so adding and moving nodes can be automatic.
- Data is replicated at multiple nodes.
- There is no single point of failure as every node has the same set of responsibilities.

As the design is decentralized, each node performs many tasks as presented in Figure 6-18.

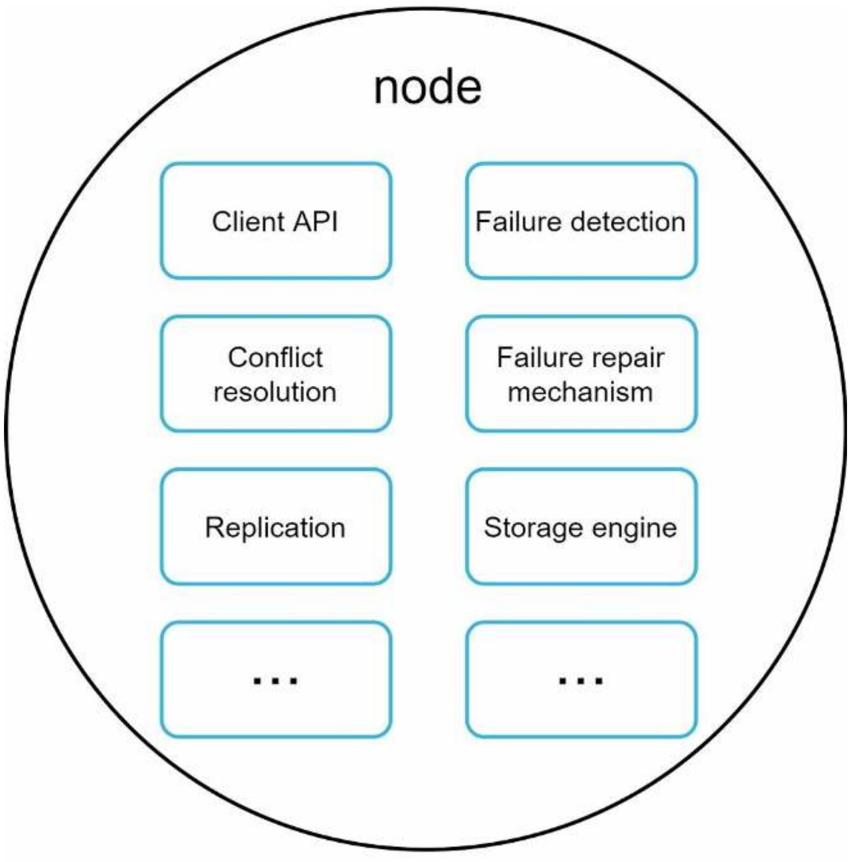


Figure 6-18

Write path

Figure 6-19 explains what happens after a write request is directed to a specific node. Please note the proposed designs for write/read paths are primary based on the architecture of Cassandra [8].

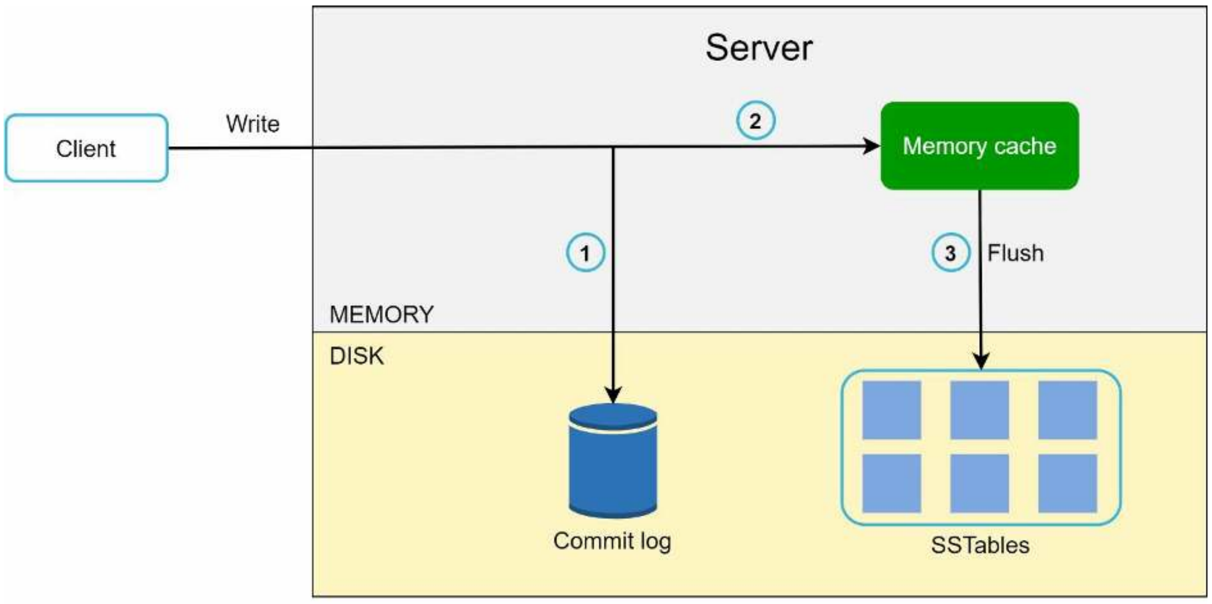


Figure 6-19

1. The write request is persisted on a commit log file.
2. Data is saved in the memory cache.

3. When the memory cache is full or reaches a predefined threshold, data is flushed to SSTable [9] on disk. Note: A sorted-string table (SSTable) is a sorted list of <key, value> pairs. For readers interested in learning more about SStable, refer to the reference material [9].

Read path

After a read request is directed to a specific node, it first checks if data is in the memory cache. If so, the data is returned to the client as shown in Figure 6-20.

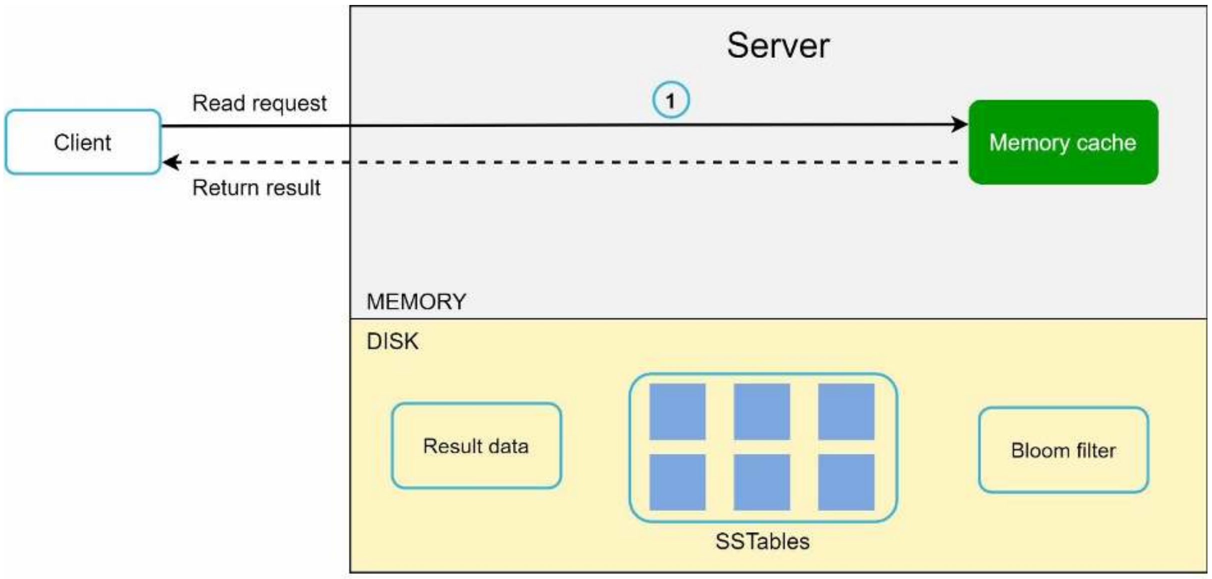


Figure 6-20

If the data is not in memory, it will be retrieved from the disk instead. We need an efficient way to find out which SSTable contains the key. Bloom filter [10] is commonly used to solve this problem.

The read path is shown in Figure 6-21 when data is not in memory.

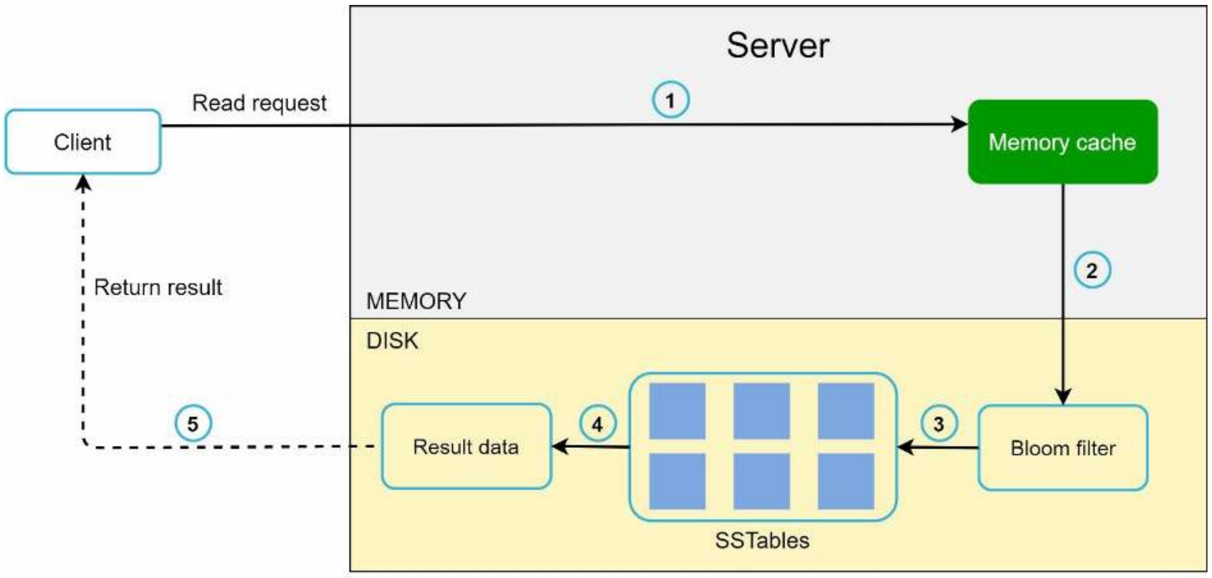


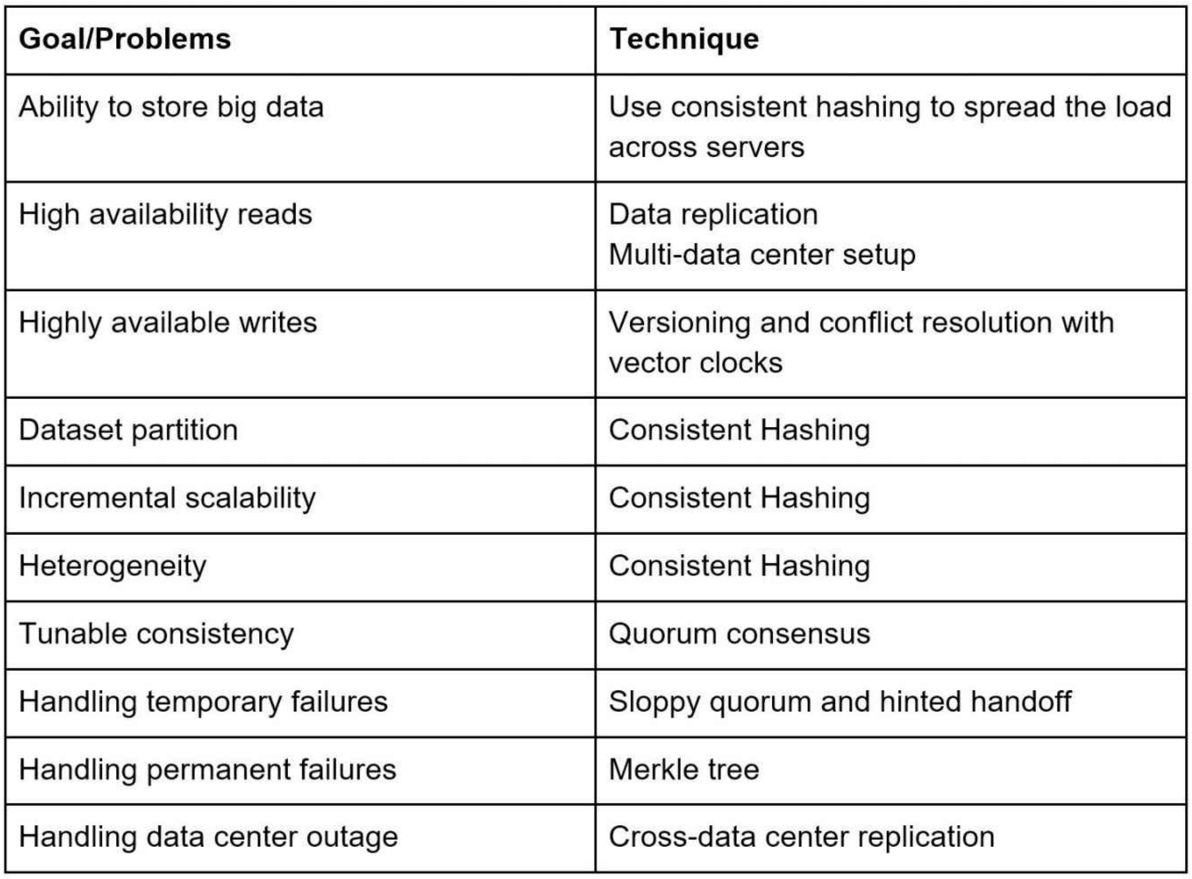
Figure 6-21

1. The system first checks if data is in memory. If not, go to step 2.
2. If data is not in memory, the system checks the bloom filter.

3. The bloom filter is used to figure out which SSTables might contain the key.
4. SSTables return the result of the data set.
5. The result of the data set is returned to the client.

Summary

This chapter covers many concepts and techniques. To refresh your memory, the following table summarizes features and corresponding techniques used for a distributed key-value store.



Goal/Problems	**Technique**
Ability to store big data	Use consistent hashing to spread the load across servers
High availability reads	Data replication Multi-data center setup
Highly available writes	Versioning and conflict resolution with vector clocks
Dataset partition	Consistent Hashing
Incremental scalability	Consistent Hashing
Heterogeneity	Consistent Hashing
Tunable consistency	Quorum consensus
Handling temporary failures	Sloppy quorum and hinted handoff
Handling permanent failures	Merkle tree
Handling data center outage	Cross-data center replication

Table 6-2

Reference materials

[1] Amazon DynamoDB: https://aws.amazon.com/dynamodb/

[2] memcached: https://memcached.org/

[3] Redis: https://redis.io/

[4] Dynamo: Amazon’s Highly Available Key-value Store: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[5] Cassandra: https://cassandra.apache.org/

[6] Bigtable: A Distributed Storage System for Structured Data: https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

[7] Merkle tree: https://en.wikipedia.org/wiki/Merkle_tree

[8] Cassandra architecture: https://cassandra.apache.org/doc/latest/architecture/

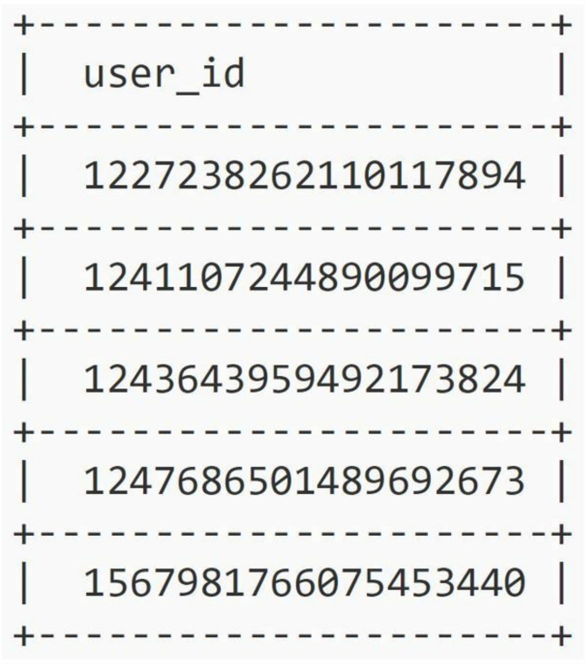
[9] SStable: https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/

[10] Bloom filter https://en.wikipedia.org/wiki/Bloom_filter

CHAPTER 7: DESIGN A UNIQUE ID GENERATOR IN DISTRIBUTED SYSTEMS

In this chapter, you are asked to design a unique ID generator in distributed systems. Your first thought might be to use a primary key with the *auto_increment* attribute in a traditional database. However, *auto_increment* does not work in a distributed environment because a single database server is not large enough and generating unique IDs across multiple databases with minimal delay is challenging.

Here are a few examples of unique IDs:



```
+---------------------+
|  user_id            |
+---------------------+
|  1227238262110117894 |
+---------------------+
|  1241107244890099715 |
+---------------------+
|  1243643959492173824 |
+---------------------+
|  1247686501489692673 |
+---------------------+
|  1567981766075453440 |
+---------------------+
```

Figure 7-1

Step 1 - Understand the problem and establish design scope

Asking clarification questions is the first step to tackle any system design interview question. Here is an example of candidate-interviewer interaction:

Candidate: What are the characteristics of unique IDs?
Interviewer: IDs must be unique and sortable.

Candidate: For each new record, does ID increment by 1?
Interviewer: The ID increments by time but not necessarily only increments by 1. IDs created in the evening are larger than those created in the morning on the same day.

Candidate: Do IDs only contain numerical values?
Interviewer: Yes, that is correct.

Candidate: What is the ID length requirement?
Interviewer: IDs should fit into 64-bit.

Candidate: What is the scale of the system?
Interviewer: The system should be able to generate 10,000 IDs per second.

Above are some of the sample questions that you can ask your interviewer. It is important to understand the requirements and clarify ambiguities. For this interview question, the requirements are listed as follows:

- IDs must be unique.
- IDs are numerical values only.
- IDs fit into 64-bit.
- IDs are ordered by date.
- Ability to generate over 10,000 unique IDs per second.

Step 2 - Propose high-level design and get buy-in

Multiple options can be used to generate unique IDs in distributed systems. The options we considered are:

- Multi-master replication
- Universally unique identifier (UUID)
- Ticket server
- Twitter snowflake approach

Let us look at each of them, how they work, and the pros/cons of each option.

Multi-master replication

As shown in Figure 7-2, the first approach is multi-master replication.

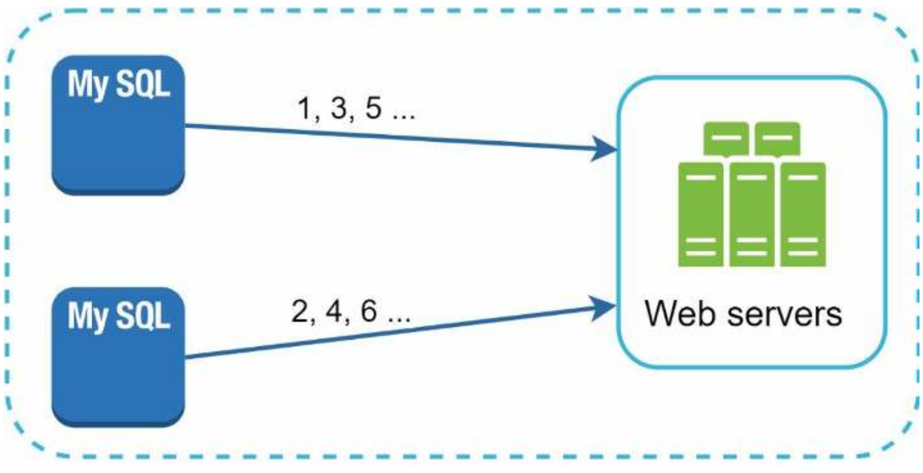


Figure 7-2

This approach uses the databases' *auto_increment* feature. Instead of increasing the next ID by 1, we increase it by *k*, where *k* is the number of database servers in use. As illustrated in Figure 7-2, next ID to be generated is equal to the previous ID in the same server plus 2. This solves some scalability issues because IDs can scale with the number of database servers. However, this strategy has some major drawbacks:

- Hard to scale with multiple data centers
- IDs do not go up with time across multiple servers.
- It does not scale well when a server is added or removed.

UUID

A UUID is another easy way to obtain unique IDs. UUID is a 128-bit number used to identify information in computer systems. UUID has a very low probability of getting collusion. Quoted from Wikipedia, "after generating 1 billion UUIDs every second for approximately 100 years would the probability of creating a single duplicate reach 50%" [1].

Here is an example of UUID: *09c93e62-50b4-468d-bf8a-c07e1040bfb2*. UUIDs can be generated independently without coordination between servers. Figure 7-3 presents the UUIDs design.

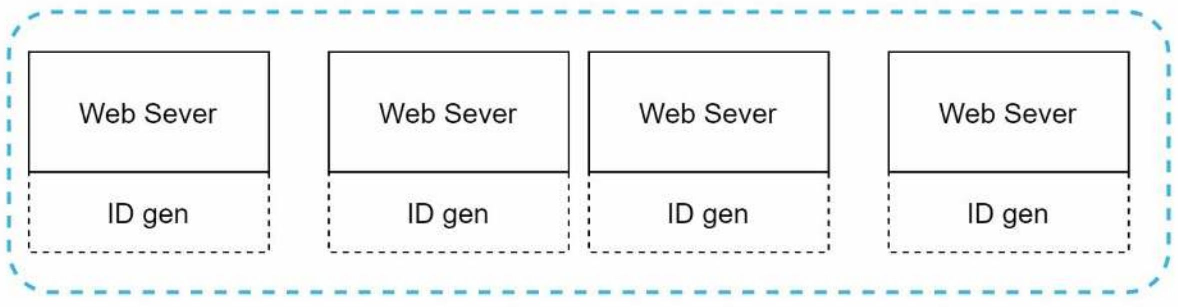


Figure 7-3

In this design, each web server contains an ID generator, and a web server is responsible for generating IDs independently.

Pros:

- Generating UUID is simple. No coordination between servers is needed so there will not be any synchronization issues.
- The system is easy to scale because each web server is responsible for generating IDs they consume. ID generator can easily scale with web servers.

Cons:

- IDs are 128 bits long, but our requirement is 64 bits.
- IDs do not go up with time.
- IDs could be non-numeric.

Ticket Server

Ticket servers are another interesting way to generate unique IDs. Flicker developed ticket servers to generate distributed primary keys [2]. It is worth mentioning how the system works.

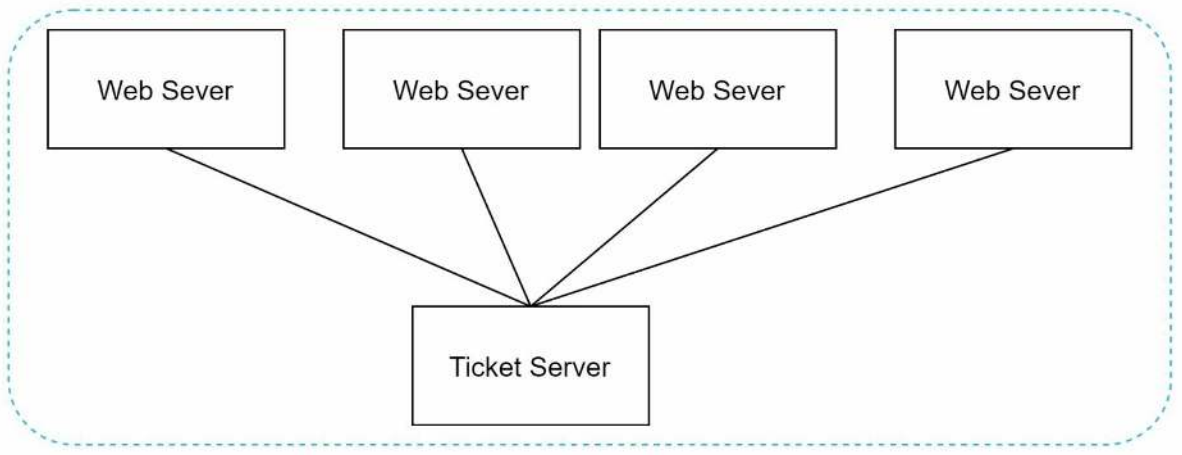


Figure 7-4

The idea is to use a centralized *auto_increment* feature in a single database server (Ticket Server). To learn more about this, refer to flicker’s engineering blog article [2].

Pros:

- Numeric IDs.
- It is easy to implement, and it works for small to medium-scale applications.

Cons:

• Single point of failure. Single ticket server means if the ticket server goes down, all systems that depend on it will face issues. To avoid a single point of failure, we can set up multiple ticket servers. However, this will introduce new challenges such as data synchronization.

Twitter snowflake approach

Approaches mentioned above give us some ideas about how different ID generation systems work. However, none of them meet our specific requirements; thus, we need another approach. Twitter's unique ID generation system called "snowflake" [3] is inspiring and can satisfy our requirements.

Divide and conquer is our friend. Instead of generating an ID directly, we divide an ID into different sections. Figure 7-5 shows the layout of a 64-bit ID.

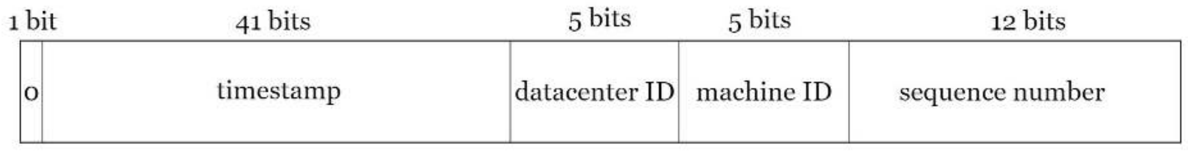


Figure 7-5

Each section is explained below.

• Sign bit: 1 bit. It will always be 0. This is reserved for future uses. It can potentially be used to distinguish between signed and unsigned numbers.

• Timestamp: 41 bits. Milliseconds since the epoch or custom epoch. We use Twitter snowflake default epoch 1288834974657, equivalent to Nov 04, 2010, 01:42:54 UTC.

• Datacenter ID: 5 bits, which gives us *2 ^ 5 = 32* datacenters.

• Machine ID: 5 bits, which gives us *2 ^ 5 = 32* machines per datacenter.

• Sequence number: 12 bits. For every ID generated on that machine/process, the sequence number is incremented by 1. The number is reset to 0 every millisecond.

Step 3 - Design deep dive

In the high-level design, we discussed various options to design a unique ID generator in distributed systems. We settle on an approach that is based on the Twitter snowflake ID generator. Let us dive deep into the design. To refresh our memory, the design diagram is relisted below.

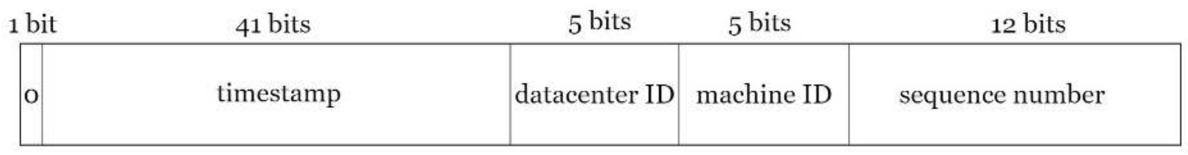


Figure 7-6

Datacenter IDs and machine IDs are chosen at the startup time, generally fixed once the system is up running. Any changes in datacenter IDs and machine IDs require careful review since an accidental change in those values can lead to ID conflicts. Timestamp and sequence numbers are generated when the ID generator is running.

Timestamp

The most important 41 bits make up the timestamp section. As timestamps grow with time, IDs are sortable by time. Figure 7-7 shows an example of how binary representation is converted to UTC. You can also convert UTC back to binary representation using a similar method.

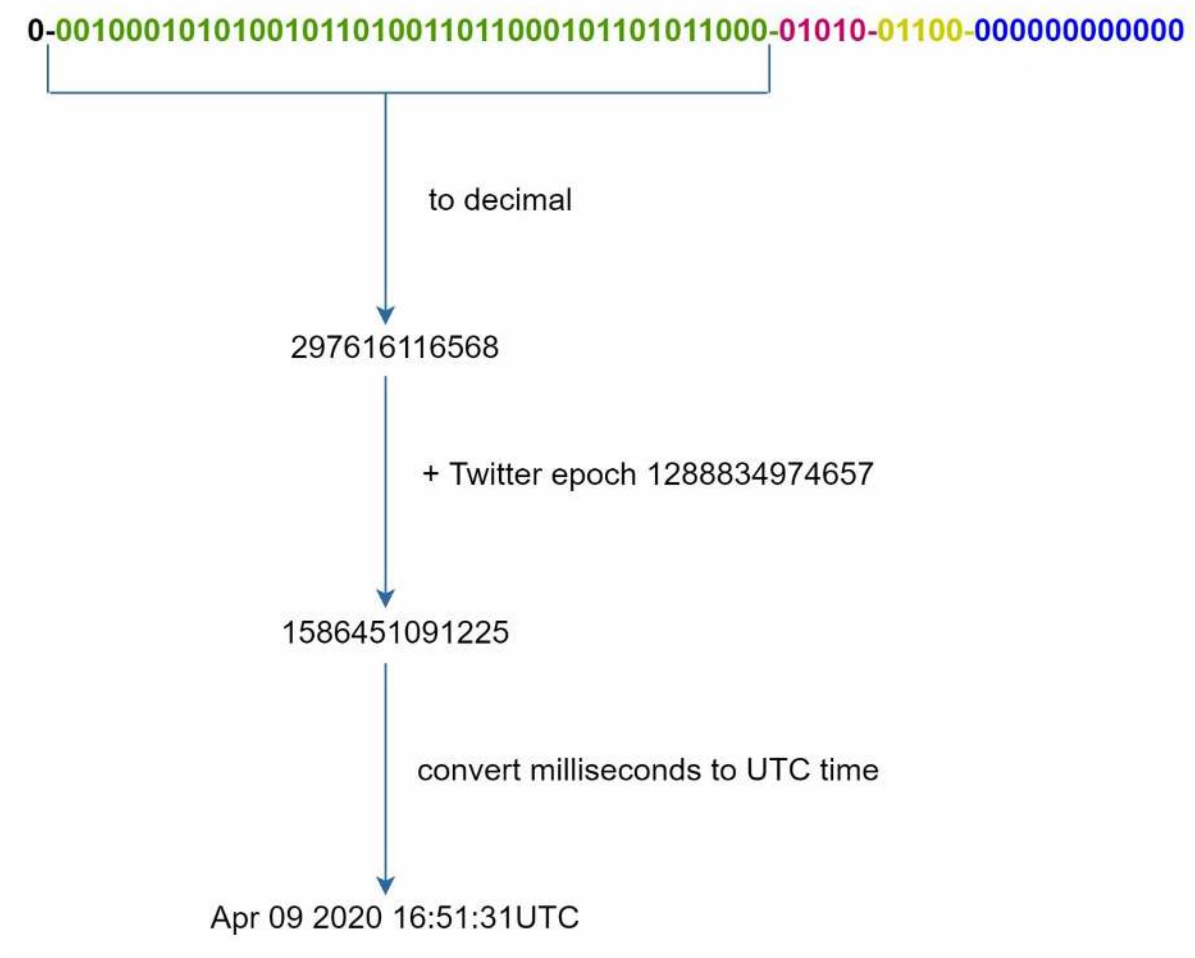


Figure 7-7

The maximum timestamp that can be represented in 41 bits is

2 ^ 41 - 1 = 2199023255551 milliseconds (ms), which gives us: ~ 69 years =

2199023255551 ms / 1000 seconds / 365 days / 24 hours/ 3600 seconds. This means the ID generator will work for 69 years and having a custom epoch time close to today's date delays the overflow time. After 69 years, we will need a new epoch time or adopt other techniques to migrate IDs.

Sequence number

Sequence number is 12 bits, which give us 2 ^ 12 = 4096 combinations. This field is 0 unless more than one ID is generated in a millisecond on the same server. In theory, a machine can support a maximum of 4096 new IDs per millisecond.

Step 4 - Wrap up

In this chapter, we discussed different approaches to design a unique ID generator: multi-master replication, UUID, ticket server, and Twitter snowflake-like unique ID generator. We settle on snowflake as it supports all our use cases and is scalable in a distributed environment.

If there is extra time at the end of the interview, here are a few additional talking points:

- Clock synchronization. In our design, we assume ID generation servers have the same clock. This assumption might not be true when a server is running on multiple cores. The same challenge exists in multi-machine scenarios. Solutions to clock synchronization are out of the scope of this book; however, it is important to understand the problem exists. Network Time Protocol is the most popular solution to this problem. For interested readers, refer to the reference material [4].
- Section length tuning. For example, fewer sequence numbers but more timestamp bits are effective for low concurrency and long-term applications.
- High availability. Since an ID generator is a mission-critical system, it must be highly available.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] Universally unique identifier: https://en.wikipedia.org/wiki/Universally_unique_identifier

[2] Ticket Servers: Distributed Unique Primary Keys on the Cheap: https://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/

[3] Announcing Snowflake: https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html

[4] Network time protocol: https://en.wikipedia.org/wiki/Network_Time_Protocol

CHAPTER 8: DESIGN A URL SHORTENER

In this chapter, we will tackle an interesting and classic system design interview question: designing a URL shortening service like tinyurl.

Step 1 - Understand the problem and establish design scope

System design interview questions are intentionally left open-ended. To design a well-crafted system, it is critical to ask clarification questions.

Candidate: Can you give an example of how a URL shortener work?
Interviewer: Assume URL https://www.systeminterview.com/q=chatsystem&c=loggedin&v=v3&l=long is the original URL. Your service creates an alias with shorter length: https://tinyurl.com/ y7keocwj. If you click the alias, it redirects you to the original URL.

Candidate: What is the traffic volume?
Interviewer: 100 million URLs are generated per day.

Candidate: How long is the shortened URL?
Interviewer: As short as possible.

Candidate: What characters are allowed in the shortened URL?
Interviewer: Shortened URL can be a combination of numbers (0-9) and characters (a-z, A-Z).

Candidate: Can shortened URLs be deleted or updated?
Interviewer: For simplicity, let us assume shortened URLs cannot be deleted or updated.

Here are the basic use cases:

1. URL shortening: given a long URL => return a much shorter URL
2. URL redirecting: given a shorter URL => redirect to the original URL
3. High availability, scalability, and fault tolerance considerations

Back of the envelope estimation

- Write operation: 100 million URLs are generated per day.
- Write operation per second: 100 million / 24 /3600 = 1160
- Read operation: Assuming ratio of read operation to write operation is 10:1, read operation per second: 1160 * 10 = 11,600
- Assuming the URL shortener service will run for 10 years, this means we must support 100 million * 365 * 10 = 365 billion records.
- Assume average URL length is 100.
- Storage requirement over 10 years: 365 billion * 100 bytes * 10 years = 365 TB

It is important for you to walk through the assumptions and calculations with your interviewer so that both of you are on the same page.