Design Key-Value Store

What is a Key-Value Store?

A key-value database is a non-relational database that stores data using a simple key-value system. It stores data as a collection of key-value pairs, with the key serving as a unique identifier for the value. Both key and value can be any type of object, from simple to complex compound objects.

Key-value databases are highly partitionable and can scale horizontally to a larger extent than other databases. For example, Amazon DynamoDB will allocate additional partitions to a table if an existing partition becomes full and more storage space is needed.

Requirements of the System

Coming up with features that the system should offer is the initial stage of any system design interview. As an interviewee, we should try to think of as many features as possible that our system should support.

- Data size: Size of values is too large to be held in memory, and we should leverage the external storage for them. However, we can still keep the data keys in memory.
- Let’s assume a few 100 TB.
- Should support updates.
- Let’s assume that there is an upper cap of 1GB to the size of the value.

Functional Requirements

- Set a value for a key (or update the value if it already exists).
- Retrieve the value that the key specifies.
- Remove the value from the key (unset the key).

Non Functional Requirements

- **High scalability:** System should be able to handle a sudden increase in load by adding more instances.
- **Consistency:** System should provide a consistent and correct response to every request.
- **Durability:** System should not lose data in the event of network partition failures.
- **Availability:** System should not have a single point of failure. According to the CAP theorem, it should be a CP system, which means consistency takes precedence over availability.

Capacity Estimation

This is usually the second part of a design interview: Coming up with the estimated numbers of how scalable our system should be. Important parameters to remember for this section are the number of queries per second and the data which the system will be required to handle. Let's assume:

- Total Storage: 100 TB.
- Total queries per second (QPS): 100K.

How would we design a simple key-value storage system in a single machine?

A hash table is a type of data structure that allows for fast read and write access to key-value pairs. It is simple to use and is supported by most programming languages. However, the disadvantage of using a hash table is that the entire data set must be stored in memory, which may not be practical for large data sets.

There are two common solutions for dealing with large data sets in a hash table:

- Compressing the data: This involves reducing the amount of space needed to store the data by using techniques such as saving references rather than actual data, using smaller data types (e.g.

float32 instead of float64), and using specialized data representations like bit arrays or vectors.
- Storing data on a disk: When it is not possible to fit the entire data set in memory, it is possible to store a portion of the data on a disk and keep the frequently accessed data in memory. This can be thought of as a caching system to optimize performance.

Distributed key-value storage

Designing a distributed key-value storage system is a complex task, especially when it comes to scaling over multiple machines. This is an important issue to consider when dealing with large amounts of data, as it will likely require a distributed system to support it. Let's look at how a distributed key-value storage system could be designed.

To store large amounts of data, we need to divide it among multiple machines. This is known as data partitioning. One way to do this is to use a coordinator machine to direct clients to the machine that has necessary resources. However, the challenge is to effectively divide the data among the machines and come up with a good data partitioning approach.

Database Sharding

To store 100TB of data, a single system may not be able to handle such a large amount. If we want to use a single machine, it would have to handle all requests, which would cause performance issues. So what is the solution? To avoid this, we can divide data into smaller pieces and distribute them across multiple machines (shards). This way, we can better manage the load and improve the performance.

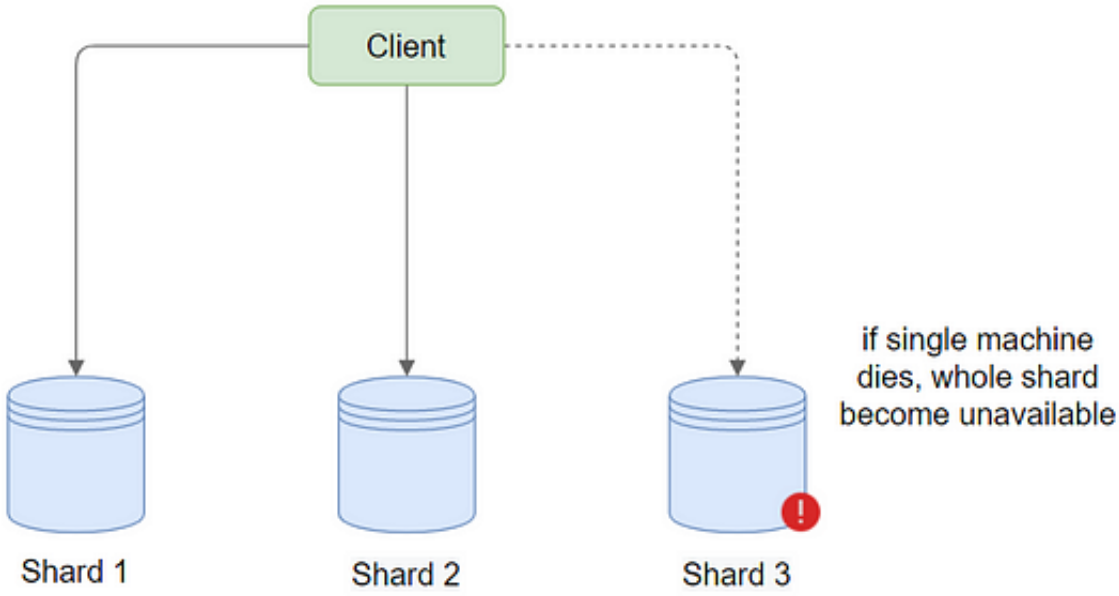
Let's take an example of messaging application where each user has their own inbox. In this situation, we can store data in a denormalized manner by sharding based on each user as a row. Here all data for a particular user will be stored in the same place, which can reduce latency when retrieving information. However, it is important to consider the sharding criteria carefully to avoid potential consistency issues, as the same data

will be stored in multiple places.

On the other side, if we choose to store the data in a normalized form, we will need to perform join across tables and rows to retrieve information. This can be undesirable because it may result in high latency and limited indexing support. So storing data in a denormalized manner will be more efficient due to the reduced latency, but it is important to take care of consistency issues.

Ensuring consistent data access is critical, and having only one copy of the data may not be enough. To achieve acceptable latency, it's important to distribute the data across multiple shards so that each shard has a reasonable load. Both reads and writes should function similarly to a single database, with an additional layer that resolves the relationship between rows, shards, and machine IPs. This layer helps identify which shard a particular row belongs to and which machine to query or write to.

The main concern with this approach is what happens if the machine hosting a particular shard fails? While maintenance downtime is acceptable, if a machine fails and the hard disk becomes corrupted, we risk losing data. This scenario could be disastrous, resulting in the loss of all communications if the shard is an important one. To mitigate this risk, it's necessary to keep multiple copies of the data we're writing. This ensures data redundancy and high availability, reducing the risk of data loss in the event of a machine failure.



System availability

One important measure to consider when evaluating a distributed system is system availability. What happens if one of our workstations breaks for some reason (hardware or software issues, for example), and how does this affect our key-value storage system?

We won't be able to return the correct response if someone requests resources from this system, it appears. When developing a side project, we may overlook this difficulty. However, if we're serving millions of people from a large number of servers, this happens frequently, and we can't afford to restart the server manually every time. This is why, in today's distributed systems, availability is critical. So, how would we go about dealing with this problem?

Of course, we can write more robust code with test cases. However, our program will always have bugs. In addition, hardware issues are even harder to protect. The most common solution is a replica. By setting machines with duplicate resources, we can significantly reduce system downtime. If a single machine has a 10% of chance to crash every month, then with a single backup machine, we reduce the probability to 1% when both are down.

Replica VS Sharding

The replica appears to be very similar to sharding at first glance. So, what's the connection between these two? When developing a distributed key-value store, how would we decide between replica and sharding?

First and foremost, we must understand the aim of these two strategies. Because a single machine can only store so much data, sharding is used to spread data over numerous machines. The replica serves as a backup for the system in the event of a failure. With that in mind, a replica won't assist if a single machine can't store all of the data.

Consistency

We can strengthen the system by introducing replicas. Consistency, on the other hand, is a problem. Let's imagine we have replica A2 for machine A1. How can we be certain that A1 and A2 have the same information? When adding a new entry, for example, we must update both machines. However, one of them may fail the write operation. As a result, A1 and A2 may accumulate a significant amount of conflicting data over time, which is a significant issue.

There are a few options available here. The first option is for the coordinator to keep a local copy. When a resource is updated, the coordinator keeps a copy of the new version. If the update fails, the coordinator can retry the procedure.

The commit log is another option. If we've been using Git, we're probably already familiar with the concept of a commit log. Essentially, each node machine will record a commit log for each transaction, which acts as a track of all changes. So, if we wish to update an entry in machine A, we'll have to go through the commit log first. Then, a separate software will process all of the commit logs (in a queue). We can recover if an operation fails since we can look at the commit log.

Redis Architecture

Redis is a key-value data store that is widely used by major IT companies

around the world. It runs in memory and is supported by Amazon Elastic Cache, making it a powerful and useful tool for key-value data storage.

What is In-Memory Key-Value Store?

In the case of an in-memory key-value store like Redis, the key-value pairs are saved in primary memory (RAM). This means that Redis stores data in RAM as key-value pairs, where the value can be a string, list, set, sorted set, or hash and the key must be a string. Here are a few examples of key-value pairs in Redis:

- Key: "name", Value: "John"
- Key: "age", Value: "30"
- Key: "location", Value: "New York"

Redis allows for a variety of value types and stores data in primary memory, making it a fast and flexible tool for storing and retrieving data.

Advantages and Disadvantages of Redis over DBMS

In database management systems, all data is typically stored in secondary storage, which can make read and write operations slower. Redis, on the other hand, saves everything in primary memory (RAM), which is much faster for data read and write operations. However, it is important to note that Redis has limitations due to the limited size and cost of primary memory compared to secondary storage.

Redis cannot store large files or binary data and is best suited for small amounts of textual data that need to be accessed, changed, and inserted quickly. If we try to write more data than the available memory allows, we may encounter errors. Overall, Redis is a useful tool for fast read and write operations with small amounts of textual data, but it is important to consider its limitations when using it to store and access data.

Thanks to Navtosh Kumar for his contribution in creating the first version of this content. Enjoy learning, Enjoy algorithms, Enjoy system design!