Design of a Key-value Store

Learn about the functional and non-functional requirements and the API design of a key-value store.

We'll cover the following

Requirements

Let’s list the requirements of designing a key-value store to overcome the problems of traditional databases.

Functional requirements

Typically, key-value stores are expected to offer functions such as `get` and `put`. However, what sets this particular key-value store system apart is its distinct characteristics, explained as follows:

- **Configurable service**: Some applications might have a tendency to trade strong consistency for higher availability. We need to provide a configurable service so that different applications could use a range of consistency models.

We need tight control over the trade-offs between availability, consistency, cost-effectiveness, and performance.

> **Note:** Such configurations can only be performed when instantiating a new key-value store instance and cannot be changed dynamically when the system is operational.

- **Ability to always write (when we picked "A" over "C" in the context of CAP)**: The applications should always have the ability to write into the key-value storage. If the user wants strong consistency, this requirement might not always be fulfilled due to the implications of the CAP theorem.

> **Note:** The context of the problem determines what will be classified as a functional requirement and what will be classified as non-functional. For example, the ability to always write (high availability) is a functional requirement for Amazon's shopping cart application, while in other cases, high availability may be considered a non-functional requirement. Drawing inspiration from Amazon's Dynamo key-value store, we can categorize the ability to always write as a functional requirement.

- **Hardware heterogeneity**: We want to add new servers with different and higher capacities, seamlessly, to our cluster without changing or upgrading existing servers. Our system should be able to accommodate and leverage different capacity servers, ensuring correct core functionality (`get` and `put` data) while balancing the workload distribution according to each server's capacity. This calls for a peer-to-peer design with no distinguished nodes.

Non-functional requirements

The non-functional requirements are as follows:

- **Scalability**: Key-value stores should run on tens of thousands of servers distributed across the globe. Incremental scalability is highly desirable. We should add or remove the servers as needed with minimal to no disruption to the service availability. Moreover, our system should be able to handle an enormous number of users of the key-value store.
- **Fault tolerance**: The key-value store should operate uninterrupted despite failures in servers or their components.

Now that you understand the functional and non-functional requirements of a key-value store, what do you think are the key differences between key-value stores and traditional databases?

In what scenarios are key-value stores particularly advantageous?

Want to know the correct answer?

Differences between key-value stores and traditional databases

H1 H2 H3 | B I | | | | | |

Provide your answer here!

Point to Ponder

Question

Why do we need to run key-value stores on multiple servers?

Show Answer

Assumptions

We'll assume the following to keep our design simple:

- The data centers hosting the service are trusted (non-hostile).
- All the required authentication and authorization are already completed.
- User requests and responses are relayed over HTTPS.

API design

Key-value stores, like ordinary hash tables, provide two primary functions, which are `get` and `put`.

Let's look at the API design.

The `get` function

The API call to get a value should look like this:

```
get(key)
```

We return the associated value on the basis of the parameter `key`. When data is replicated, it locates the object replica associated with a specific key that's hidden from the end user. It's done by the system if the store is configured with a weaker data consistency model. For example, in eventual consistency, there might be more than one value returned against a key.

Parameter	Description

`key`	It's the `key` against which we want to get `value`.

The `put` function

The API call to put the value into the system should look like this:

```
put(key, value)
```

It stores the `value` associated with the `key`. The system automatically determines where data should be placed. Additionally, the system often keeps metadata about

educative

Parameter	Description
`key`	It's the `key` against which we have to store `value`.
`value`	It's the object to be stored against the `key`.

Point to Ponder

Question

We often keep hashes of the value (and at times, value + associated key) as metadata for data integrity checks. Should such a hash be taken after any data compression or encryption, or should it be taken before?

Show Answer

Data type

The key is often a primary key in a key-value store, while the value can be any arbitrary binary data.

> **Note:** Dynamo uses MD5 hashes on the key to generate a 128-bit identifier. These identifiers help the system determine which server node will be responsible for this specific key.

In the next lesson, we'll learn how to design our key-value store. First, we'll focus on adding scalability, replication, and versioning of our data to our system. Then, we'll ensure the functional requirements and make our system fault tolerant. We'll fulfill a few of our non-functional requirements first because implementing our functional requirements depends on the method chosen for scalability.

> **Note:** This chapter is based on Dynamo, which is an influential work in the domain of key-value stores.

← Back

Mark As Completed

Next →

System Design: The Key-value Store

Ensure Scalability and Replication