Redis Hotspot Key Discovery and Common Solutions

Alibaba Clouder February 11, 2019

In this article, we will discuss the common causes of hotspot keys in Redis and propose several solutions to handle this issue effectively.

Whenever we have a database with a large number of users, it is not unusual to come across hotspots in the database. For Redis, frequent access of the same key in a partition is known as a hotspot key. In this article, we will discuss the common causes of hotspot keys, evaluate the impact of this problem, and propose effective solutions to handle hotspot keys.

Common Causes of Hotspot Keys

Reason 1: The size of user consumption data is much greater than that of production data, and includes hot items, hot news, hot reviews, and celebrity live broadcasts.

An unexpected event in your daily work and life, such: cut prices and promotion of certain popular commodities during the day of sticks, when one of these items is browsed or purchased tens of thousands of times, there will be a larger demand, and in this case it will cause hot issues.

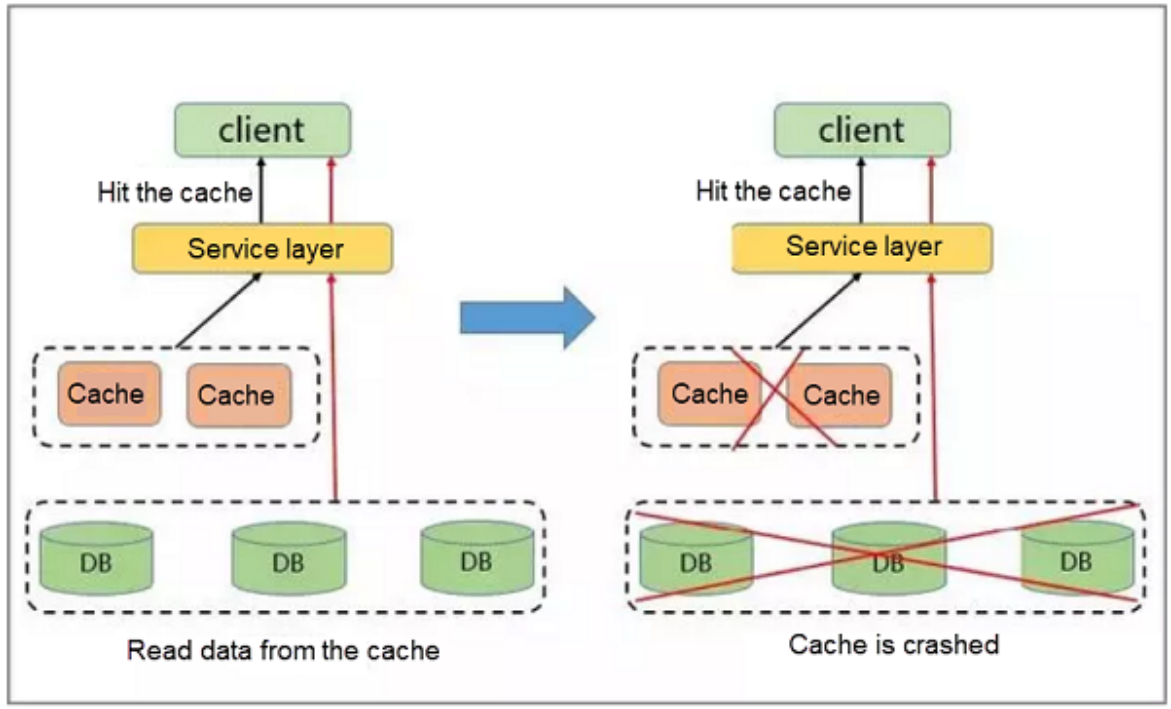
Similarly, it has been published and viewed by a large number of hot news, hot comments, Star live broadcast, and so on, these typical read-write-less scenes

also create hot issues.

Reason 2: The number of request slices exceeds the performance threshold of a single server.

When a piece of data is accessed on the server, the data is normally split or sliced. During the process, the corresponding key is accessed on the server. When the access traffic exceeds the performance threshold of the server, the hotkey key problem occurs.

Impact of the Hotspot Key Problem



1. Traffic is concentrated, reaching the upper limit of the physical network adapter.
2. Too many requests queue up, crashing the sharding service of the cache.
3. The database is overloaded, resulting in a service avalanche.

As mentioned earlier, when the number of hotspot key requests on a server exceeds the upper limit of the network adapter on the server, the server stops providing other services due to the excessive concentration of traffic.

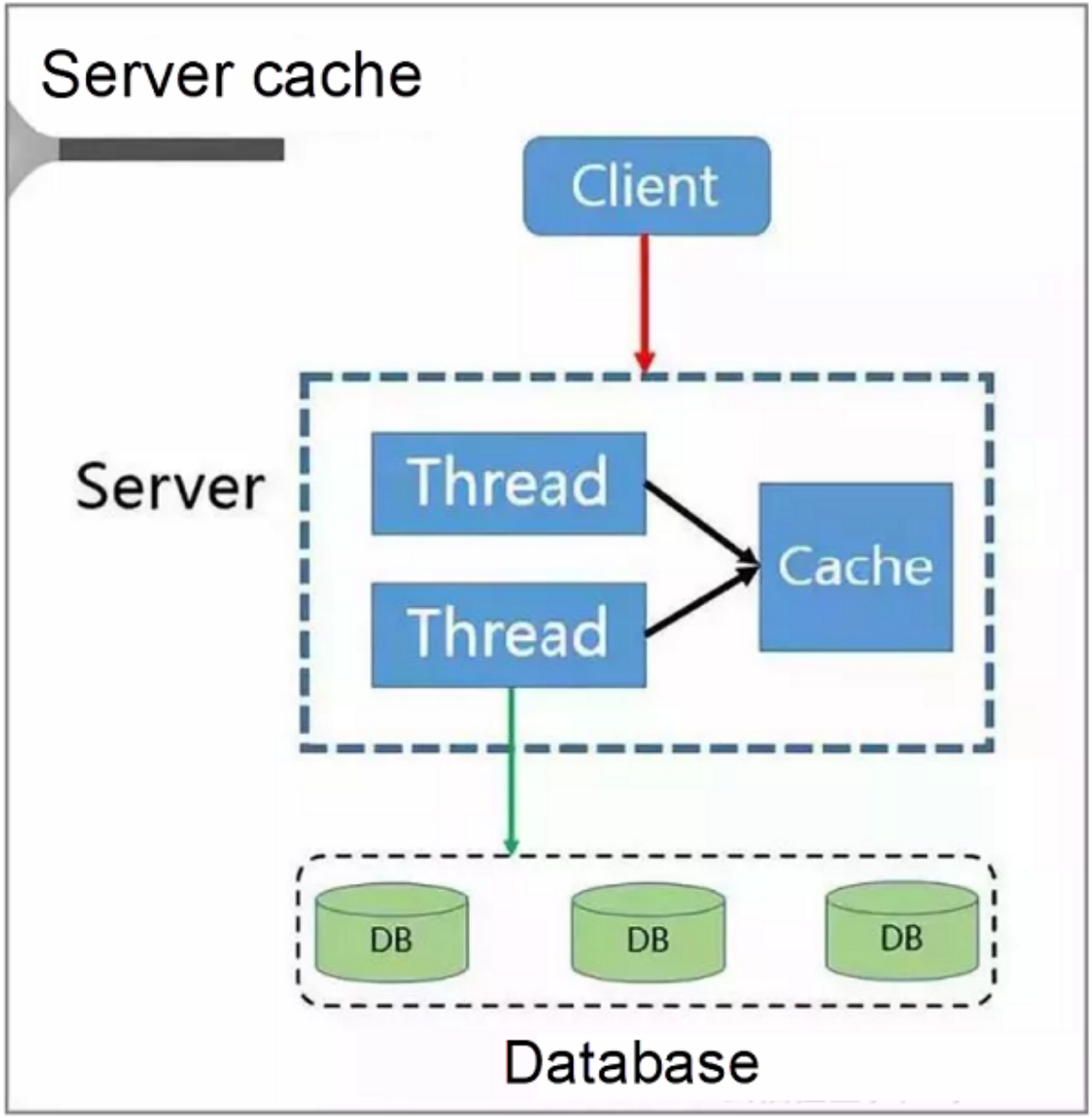
If the distribution of hotspots is too dense, a large number of hotspot keys are cached, exhausting the cache capacity and crashing the sharding service of the cache.

After the cache service crashes, the newly generated requests are cached on the background database. Due to the poor performance of this database, it is prone to exhaustion from a large number of requests, leading to a service avalanche and dramatic downgrading of performance.

Recommended Solutions

A common solution to improve performance is through reconstruction on the server or client.

Server Cache Solution



The client sends the requests to the server. Given that the server is a multi-thread service, a local cache space based on the cache LRU policy is available.

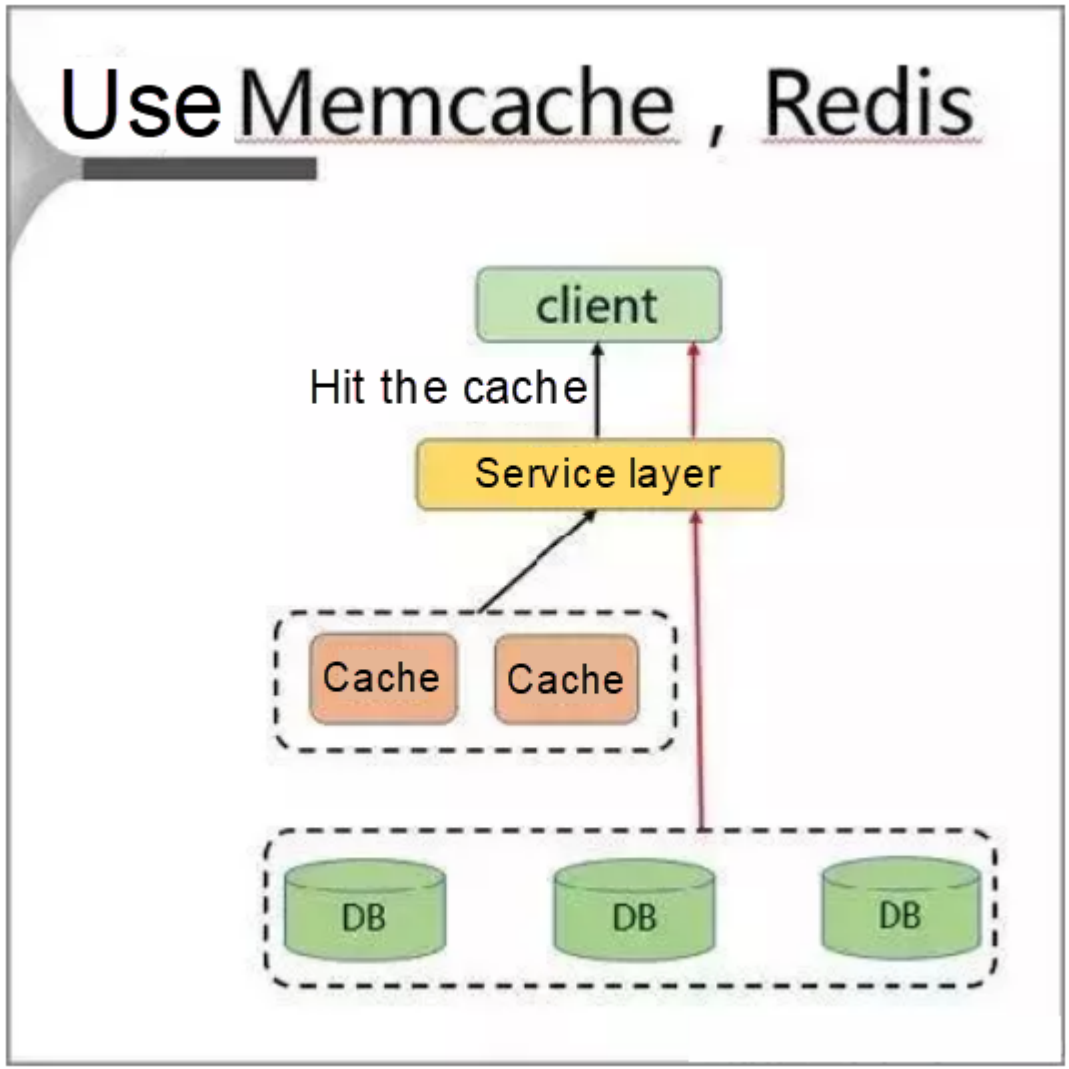
When the server becomes congested, it directly repatriates the requests rather than forwarding them to the database. Only after the congestion is cleared later does the server send the requests from the client to the database and re-write the data to the cache.

The cache is accessed and rebuilt.

However, the programme also has the following problems:

1. Cache building problem of the multi-thread service when the cache fails
2. Cache building problem when the cache is missing
3. Dirty reading problem

"MemCache + Redis" Solution



In this solution, a separate cache is deployed on the client to resolve the hotspot key problem.

When adopting this solution, the client first accesses the service layer and then the cache layer on the same host.

This solution offers the following advantages: nearby access, high speeds, and zero bandwidth limitation. However, it also suffers from the following disadvantages:

1. Wasted memory resources
2. Dirty reading problem

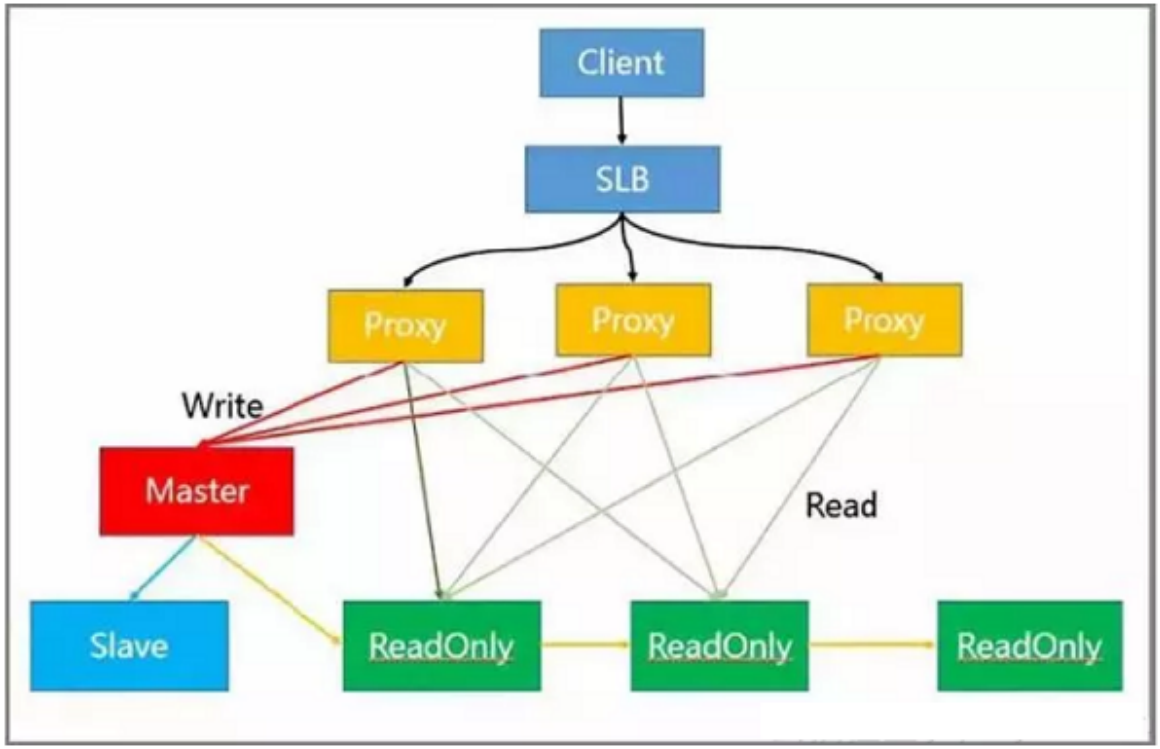
Local Cache Solution

Using the local cache incurs the following problems:

1. Hotspots must be detected in advance.
2. The cache capacity is limited.
3. The inconsistency duration is long.
4. Hotspot keys are incomplete.

If traditional hotspot solutions are all defective, how can the hotspot problem be resolved?

Read/Write Splitting Solution



This solution resolves the hotspot reading problem. The following describes the functions of different nodes in the architecture:

1. Load balancing is implemented at the SLB layer.
2. Read/Write separation and automatic routing are implemented at the proxy layer.
3. Write requests are processed by the master node.
4. Read requests are processed by the read-only node.
5. HA is implemented on the slave and master nodes.

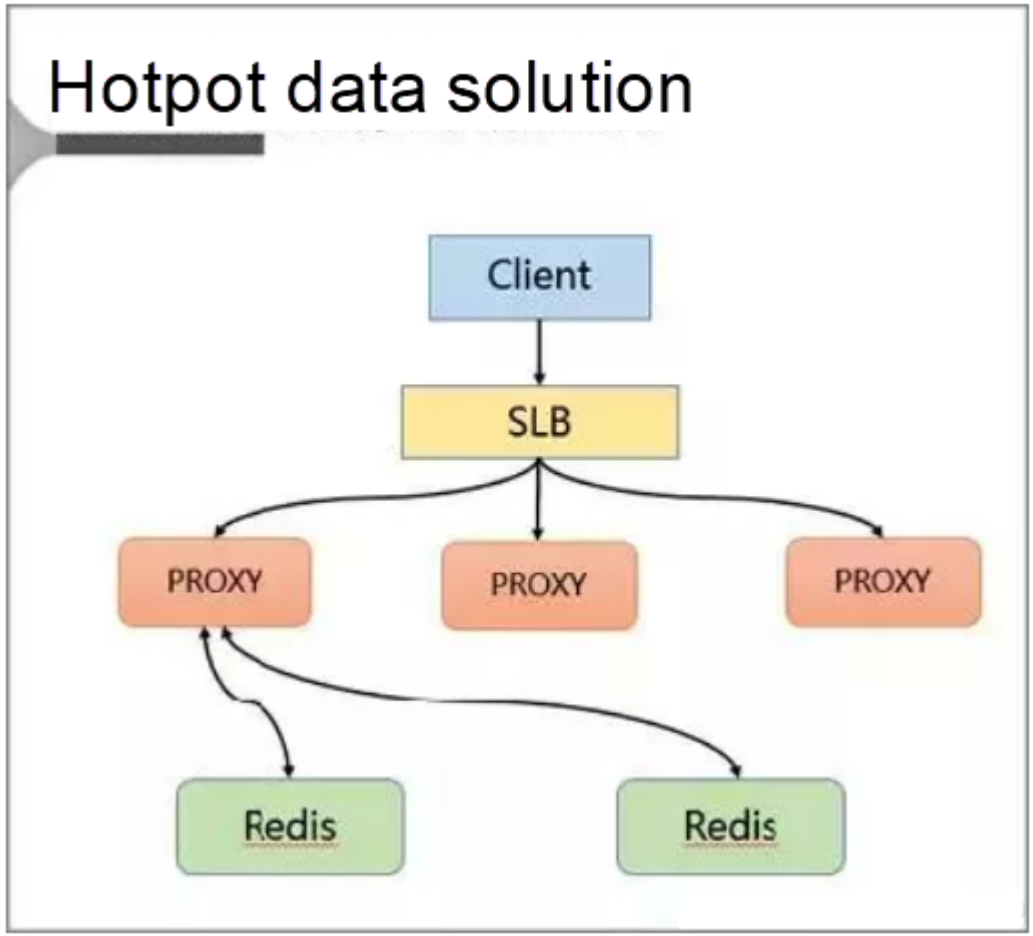
In practice, the client sends requests to SLB, which distributes these requests to multiple proxies. Then, the proxies identify and classify the requests and further distribute them.

For example, a proxy sends all write requests to the master node and all read requests to the read-only node.

But the read-only node in the module can be further expanded, thus effectively solving the problem of hot-spot reading.

Read and Write separation also has the ability of flexible capacity expansion reading hotspots, can store a large number of hotspots key, client-friendly advantages such.

Hotspot Data Solution



In this solution, hotspots are discovered and stored to resolve the hotspot key problem.

Specifically, the client accesses SLB and distributes requests to a proxy through SLB. Then, the proxy forwards the requests to the background Redis by the means of routing.

In addition, a cache is added on the server.

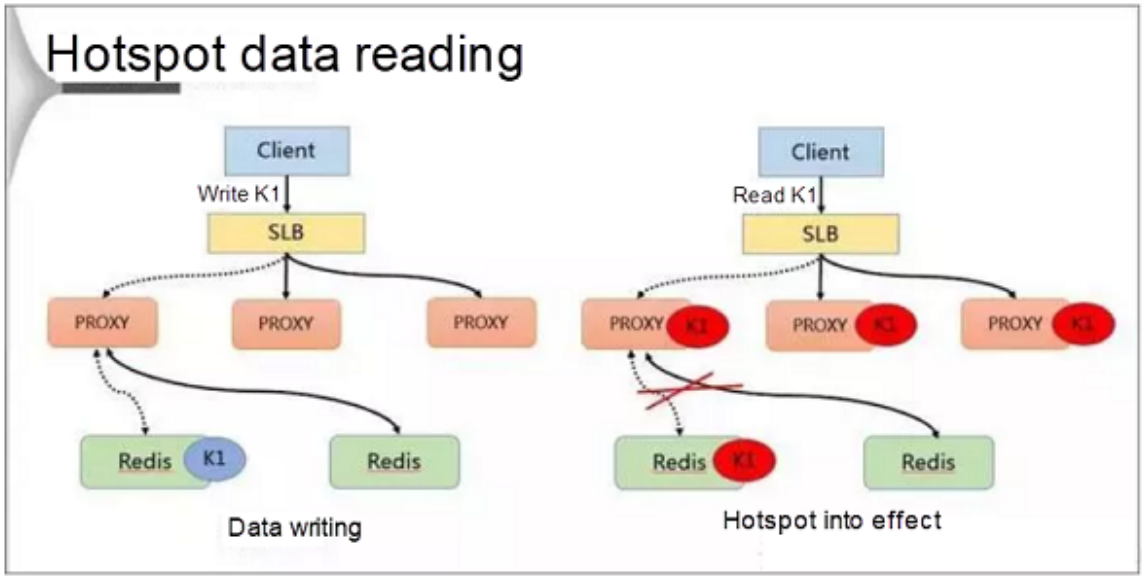
Specifically, a local cache is added to the proxy. This cache uses the LRU algorithm to cache hotspot data. Moreover, a hotspot data calculation module is added to the background database node to return the hotspot data.

The key benefits of the proxy architecture are:

1. The proxy locally caches the hotspot data, and its reading capability is horizontally scalable.
2. The database node regularly calculates the hotspot data set.
3. The database feeds the hotspot data back to the proxy.
4. The proxy architecture is completely transparent to the client, making efforts to impose compatibility unnecessary.

Processing of Hotspot Keys

Reading of the Hotspot Data

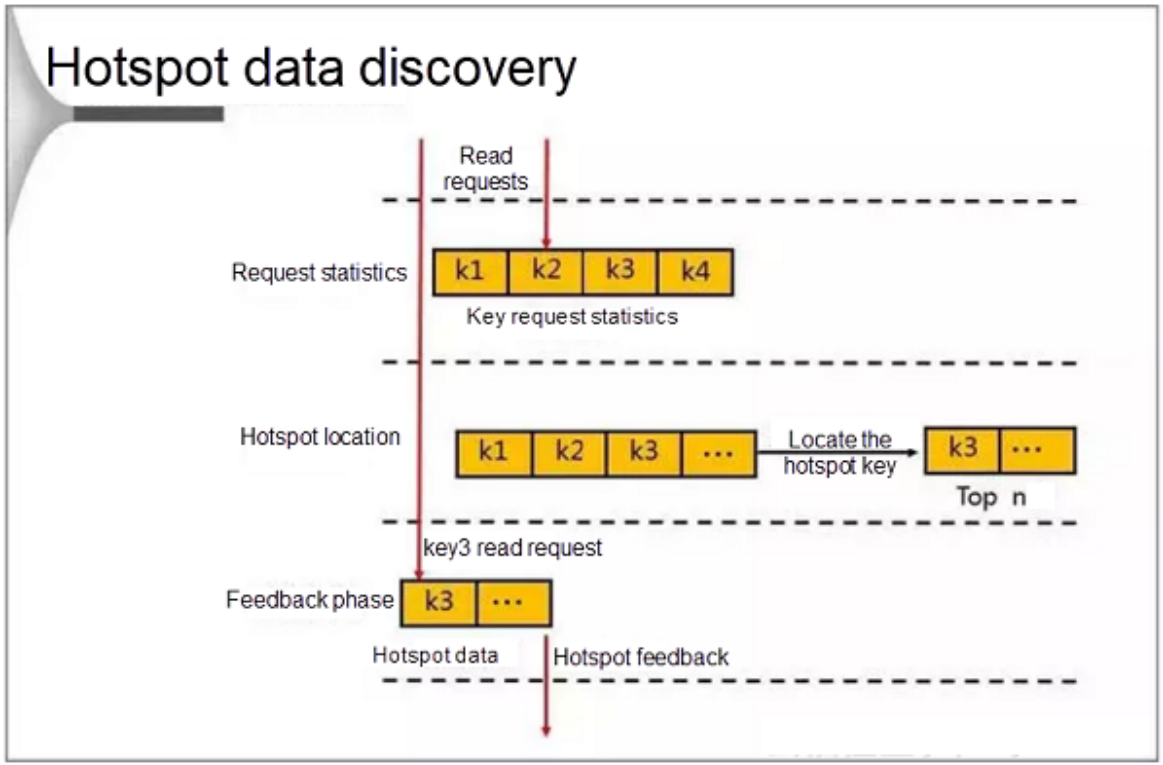


Hotspot key processing is divided into two jobs: writing and reading. During data writing, SLB receives data K1 and writes it to a Redis database through a proxy.

If K1 becomes a hotspot key after the calculation conducted by the background hotspot module, the proxy caches the hotspot. In this way, the client can directly access K1 the next time, bypassing Redis.

Finally, because the proxy can be horizontally expanded, the accessibility of the hotspot data also can be enhanced infinitely.

Discovery of Hotspot Data



During the discovery, the database first counts the requests that occur in a cycle. When the number of requests reaches the threshold, the database locates the hotspot keys and stores them in an LRU list. When a client attempts to access data by sending a request to the proxy, Redis enters the feedback phase and marks the data if it finds that the target access point is a hotspot.

The database uses the following methods to calculate the hotspots:

1. Hotspot statistics based on statistical thresholds.
2. Hotspot statistics based on the statistical cycle.
3. Statistics collection method based on the version number, which does not require resetting the initial value when being used.

4. Calculating hotspots on the database features minimal performance impacts and lightweight memory occupation.

Comparison of Solutions

From the preceding analysis, you can see that both solutions are improvements over traditional solutions in resolving the hotspot key problem. In addition, both the read/write separation and the hotspot data solutions support flexible capacity expansion and are transparent to the client, though they cannot ensure 100% data consistency.

The read/write separation solution supports the storage of a large hotspot data volume, while the proxy-based hotspot data solution is more cost-effective.