Research and Application of Distributed Cache Based on Redis

Lanying Shi*, Hongming Qiao, Chengwei Yang, Yiquan Jiang, Kefeng Yu, Chunhua Chen
China Telecom Research Institute, Guangzhou, 510000, China.

* Corresponding author. Tel.: 13808871920; email: shily3@chinatelecom.cn (L.Y.S.)
Manuscript submitted May 4, 2023; accepted June 8, 2023; accepted August 10, 2023; published January 12, 2024.
doi: 10.17706/jsw.19.1.1-8

Abstract: As a traditional relational database, MySQL provides complete ACID operations, supports rich data types, powerful associated queries, where statements, etc., can easily establish query indexes, perform complex operations such as internal connection, external connection, summation, sorting, grouping, and support functions such as stored procedures. The product is highly mature and powerful. However, when databases face large-scale data access, disk I/O often becomes a performance bottleneck. In complex telecommunications business scenarios, high-frequency access to data such as product instances, sales product instances, product/sales product specifications, and accounts in telecommunications business can result in high database load and low efficiency if operated directly across tables or databases. Adopting distributed caching products can greatly improve performance and reduce database load. Therefore, this article proposes a distributed caching system to solve the disk I/O performance bottleneck caused by large-scale and high concurrency database access. It is a memory based, persistence, high-performance, highly reliable, and horizontally scalable distributed NoSQL memory database product. It supports memory management of sub databases and sub tables, provides disaster recovery, recovery, monitoring, migration and other capabilities, and supports transparent access for API provisioning.

Keywords: database, distributed, massive scale, high concurrency

1. Introduction

There are many open source cache products [1]. This article introduces and compares the memory storage products Redis and Memcached on mainstream cloud platforms.

Redis is a high-performance structured database [2, 3]. Using jemalloc or tcmalloc multi-level Memory pool mechanism, memory allocation performance is stable and allocative efficiency is better than Memcached [4]. Supports multiple data composite objects (lists, maps, collections). It can achieve local modification of large objects with low latency, and has the richest data structure and persistence characteristics. In addition, the cluster version provides data migration commands with high availability and multi copy mechanism [5]. However, due to Redis's use of a single threaded asynchronous IO model, it is not possible to fully utilize the characteristics of multi-core CPUs. At the same time, there is a lack of supporting facilities for operation and maintenance monitoring, automatic fault detection and recovery, and insufficient functional details in terms of availability, reliability, scalability, and disaster recovery, such as not supporting optimistic locking mechanisms or transparent access after data fragmentation, and it does not support automated deployment and online expansion [6].

Memcached is a distributed key value memory library implemented by client APIs [7]. The memory bucket based Memory pool model ensures stable data write performance [8]. Optimistic locking has been implemented to address the issue of concurrent data modification. The use of a multi-threaded model can fully utilize single machine resources, support optimistic locking, and solve the problem of concurrent data modification. However, Memcached does not support complex data objects and cannot meet the expression requirements of complex business object [8, 9]. Lack of supporting facilities for operation and maintenance monitoring, automatic fault detection, and recovery. Lack of data persistence backup and recovery tools, as well as data disaster recovery capabilities for application-level failures. It has weak scalability and does not support automated deployment and online expansion, nor does it support transparent access after data fragmentation [10].

Through comparison, it can be concluded that open-source products lack tools and means related to automated operation and maintenance, visual cluster management, and so on. In case of system exceptions such as disk and network, the processing mechanism is single. Inadequate support for data reliability and system availability in abnormal situations; Unable to quickly keep up with enterprise needs in terms of specific demand support.

2. Basic Concepts and Definition

Ctg-Cache: Distributed cache is a high-performance, highly reliable, and horizontally scalable distributed NoSQL memory database product that is compatible with the Redis protocol. It supports memory management of sub databases and sub tables, and provides disaster recovery, recovery, monitoring, migration and other capabilities.

Instance: The application uses a resource of distributed caching, with an instance containing an access cluster and a data storage cluster (Redis cluster).

Group: the alias of Db, which corresponds to the database of Redis one by one, similar to the table of relational database.

Access: Access machine is a middleware that connects clients and storage nodes. Applications do not directly connect to storage nodes, but rather connect through access machines. Used to mask the application's perception of distributed data management and provide certain load balancing functions.

Access SET: An access cluster composed of a set of access nodes, specifically serving a specific instance.

Redis data node: A highly available storage node deployed on two machines, one active and one standby, for data storage.

Data SET: Refers to a Redis cluster, a storage cluster composed of a set of data storage nodes, specifically serving a certain instance.

3. Design Constraints

3.1. Account Security Management

(1) Set menus, buttons, and data permissions for relevant tenants, users, and roles on the cloud platform.

(2) Delete all anonymous accounts and empty username accounts to prevent anonymous account connections and avoid security risks.

(3) Use tenants (users) to isolate data cross access between applications.

(4) Authorize access partitions and data partitions for users.

(5) The application side password configuration must be encrypted.

3.2. Network Security

(1) The storage layer and access layer are deployed inside the firewall, and unified access routes and

policies from the access layer to the application end are set in advance.

(2) Data nodes and access nodes are not open to public IP. The access diagram is shown in Fig. 1.

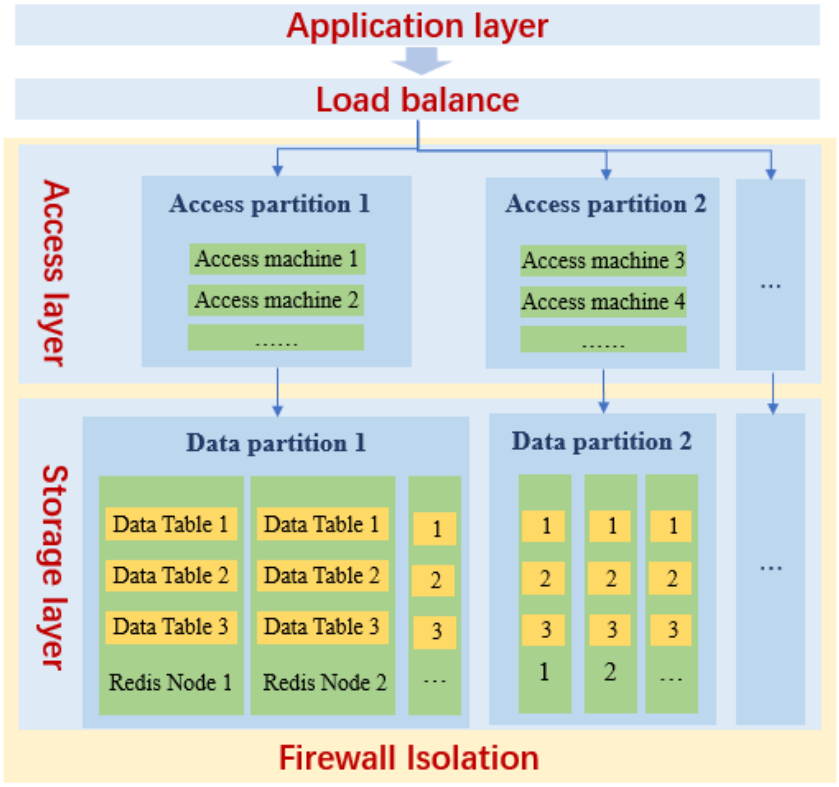

Fig. 1. Architecture diagram.

3.3. Data Desensitization

Data desensitization refers to the deformation of certain sensitive information through desensitization rules to achieve reliable protection of sensitive and private data. In the case of involving customer security data or some commercially sensitive data, real data shall be transformed and provided for test use without violating the system rules. Personal information such as ID number, mobile phone number, card number, customer number, etc. shall be desensitized.

For example, in a hot scenario 1, it is necessary to query customer information based on the user's card number using a cache index card number to customer relationship information. It is recommended to use md5 for card number information (md5 (card number + fixed mixed value 1), fixed value 2).

For example, in a hot scenario 2, it is necessary to verify whether the user's ID number is cached to index customer information, including sensitive information such as card numbers, based on the ID number entered after the user logs in. It is recommended to use MD5 for card number information (MD5 (card number + fixed mixed value 1), customer ID)

The usage security of scenario 2 is much higher than that of scenario 1.

3.4. Storage Constraints

(1) It is recommended to apply the dimension sub tenant design, with each tenant corresponding to one instance.

(2) Each instance corresponds to a data partition and an access partition.

(3) Each data partition supports 255 database (1-255) by default. If there is a larger database requirement, it can be configured by modifying the Redis template, and the size cannot exceed 10240.

(4) A single key value access message does not exceed 32K.

(5) Balances performance and reliability. Distributed caching does not guarantee 100% success for each command. Therefore, if the application has high requirements for success, it is necessary to retry the command for failure under a certain strategy.

4. Classic Scene Design and Application

There are two theories about the use of caching. The first one is caching, which temporarily stores data without the need for high reliability and allows for a certain amount of delay and misreading of cached data. This is called a hot usage scenario. Another type of cache that has gradually evolved into an important storage medium requires reliable data storage, known as reliable storage scenarios or NoSQL in memory database usage scenarios.

Hot data scenarios: Generally suitable for high read and low write data operation scenarios, such as static configuration caching, one-time data results such as bills, lists, etc. Please refer to usage examples. The main drawback of hot data scenarios is dirty reads caused by the delay between cached and business data, and maintaining consistency between business and cached data often increases the complexity of system design.

Reliable storage scenario: In times of peak business pressure or cache failure, there is often a difficult problem of cache penetration, which is usually solved by moving the entire hot data table to memory for storage, and related data services only fetch the cache.

Currently, Ctg-cache supports data persistent storage and disaster recovery to ensure data reliability. In transactions, optimistic lock data collision mechanisms (data multi version merging and conflict detection mechanisms) are supported to solve the problem of concurrent data modification conflicts. In the following cases, we will describe how to prevent overselling in a flash kill scenario and how to implement a distributed transaction transfer service.

4.1. Hot Data Scenarios

Business background: Generally, there are bottlenecks in existing systems, and hot query services are first migrated to the cache. Data changes are still in the database and are not transparent to the new business system.

Business requirements: Data is stored by other reliable storage devices, and cache only stores hot data. Ensure cache updates by setting the cache expiration time. Retrieve cached data if it does not exist. Add a data reconciliation tool to asynchronously refresh the cache (see Fig. 2).

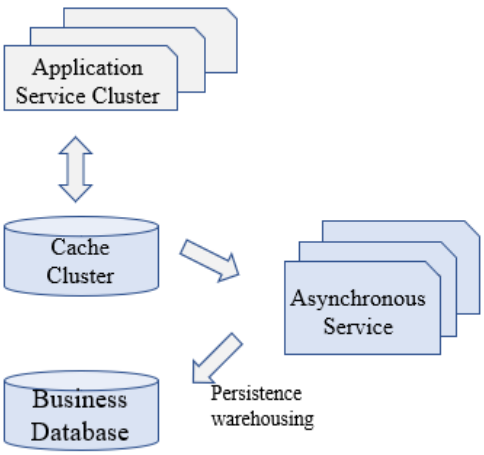

Fig. 2. Interaction diagram of hot data scenarios.

Demand analysis: Establish users, establish Redis data partitions, establish access partitions, and select a hot storage template (disk flushing strategy and master-slave synchronization strategy are different). After establishing the environment, create a group (table) and verify that the group is available according to the development example. You can refer to developing sample code for caching API usage. Use the API to set the cache key timeout. Cache usage loading can refer to the cache construction of hot keys. Use a scheduled task to scan the data source to obtain incremental updates to the data modification cache or update the cache at the business code where the data change event is sent.

Reliable storage scheme: If configuration data requires high data consistency or hot spots change frequently, a reliable storage scenario can be considered for design. However, this scheme requires transparent modification of business data and can be migrated to the cache as a whole. The design of the following cache sections will become simpler: Data is stored by the cache and cannot be lost. Configuration class data can be used as secondary hotspot data as a whole and stored in the cache, simplifying the system architecture scheme. Create a Redis data partition and select a persistent storage template (different disk flushing strategies and master-slave synchronization strategies). You can refer to developing sample code for caching API usage.

4.2. Seckill Inventory Control

Business Scenario: To sell a package or product, set the total quantity of products to be sold. During sales, there should be no oversold, that is, the remaining quantity of sales products cannot be negative.

Business requirements: The system has a high concurrency, and the rush purchase rate for extremely popular packages is only 1%. For example, 100 sales items are snapped up in a few seconds. In the front-end inventory judgment, it is necessary to support a 10w inventory quick judgment. Products cannot be oversold. Multiple merchants sell to ensure data consistency of the remaining product quantity (see Fig. 3).

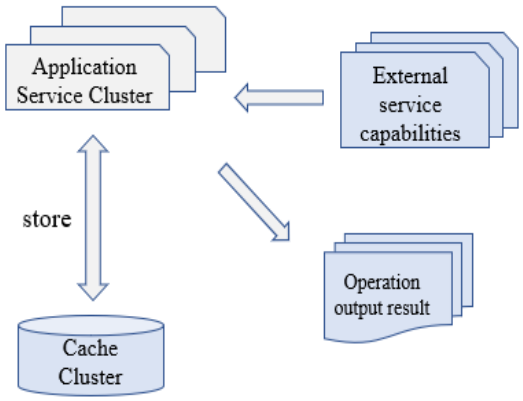

Fig. 3. Interaction diagram of seckill inventory control.

Demand analysis: Obtaining the remaining quantity of goods for condition judgment and updating data are two independent operations, with other applications potentially modifying data and causing conflicts. When a conflict occurs, users need to merge the latest data, and then perform conditional judgment and data modification. Packaging two independent operations into an atomic service ensures that the remaining quantity of the product is not negative after (remaining quantity - sales volume). The cache provides a vget operation that returns a version number and a data subtraction operation with a version number vdecrBy.

Example Scheme 1: First support atomic transaction operations with oversold risk

```
//There is no conflicting inventory reduction atomic function, and inventory judgment is being executed
public boolean decreaseFunction2(String key,long value, Jedis jedis,int retreyNumber){
```

```
Long remain = jedis.decrBy( key, 1);
return remain.longValue() > 0;
}
```

Example Scheme 2: Using the optimistic lock mechanism of the optimistic lock client to achieve.

```
// No conflicting inventory reduction atomic function
public boolean decreaseFunction(String key,long value, VProxyJedis vjedis,int retreyNumber){
    boolean flag = false;
    // When the deduction operation fails, retry the operation. The number of retries is retryNumber
    while ( (!flag) && (retreyNumber>0) ){
        try{
            // Atomic operation 1: Obtain the value and version number of the corresponding value of
the current key through vget
            VersionResult<String> result = vjedis.vget( key);
            // Judge whether the quantity of remaining goods is sufficient
            if((Long.parseLong(result.getValue())-value)>=0){
                // There may be other applications modifying data between operation 1 and operation 2
that generate inconsistent states
                // Atomic Operation 2: Product Remaining Quantity - Sales Volume, updated based on
the version number determined by current conditions
                vjedis.vdecrBy( key, value, result.getVersoion());
            }
            // The remaining quantity of goods has been deducted successfully. Set the operation success
flag flag flag=true
            flag = true;
        }catch(Exception e){
  // If another process modifies the data, a version number conflict will occur. Error code: ErrorMessage_
DiffVerison
            if(e.getMessage().equals(ErrorMessage_DiffVerison)){
      flag = false; // Set whether the operation succeeded flag flag=false
          retreyNumber--;// Subtract the number of retries and perform the operation through the while
loop
            }
        }
    }
    return flag;
}
```

5. Experimental Result

We plan a memory application scenario. Suppose the data volume is approximately 500G, and the storage format is map. The QPS is about 20w/s, and the TPS is about 2w/s.

Resource calculation formula

a) Access Set: (Requirement QPS+ Requirement TPS)/Single machine TPS)

Data - Set: max (D1, D2, D3) *2+1

D1=Demand QPS/Single machine QPS,

D2=Demand TPS/Single TPS

D3=demand data size/ (single machine memory * (70-90) % * loss coefficient)

Data Storage Data Security and Consistency:

AOF storage frequency: The benchmark is per second, with a 10% decrease in TPS per flash

Primary and backup synchronization: The benchmark is asynchronous replication, semi synchronous, with a 10% decrease in TPS

Backup copy: The storage layer benchmark is 2 primary and backup copies+1 empty buffer machine

T=Access-Set+ Data - Set

For cache instances of three host specifications, the following data was obtained through experiments. From the experimental data, it can be concluded that the TPS and QPS of the access layer and storage layer of the cache can meet the needs of the scenario.

Table 1. The TPS and QPS of access layer and storage layer

Host specifications	Access layer		Storage layer		
	Tps (w/s)	Qps (w/s)	Tps (w/s)	Qps (w/s)	Data(G)
2 * ten core 2.3GHZ CPU, 512GB memory, 4 * 600G, integrated 2 * 1000M power grid port, 1 * dual port 10GE optical network card	20	20	15	20-30	360-460G
C 10core, 768G memory, 2*300G SAS+5 * 900G SAS hard disk, RAID card, Cache, 6*1GE power	18	18	10	10~20	540-690G
4-way 8-core, 512GB memory, 6* 1GE power, SAS hard drive	20	20	12	10~20	360-460G

The cache usage of the Internet of Things system is detailed in Table 2 below. It mainly includes product configuration, in transit order flow data, database global index, cache, distributed session, Distributed Order lock, and so on.

Table 2. The cache usage of the experimental system

Cluster Name / Instances Number	Index Name	Peak value	Mean value	Occupy space
CRM Index/11	Global Index	20w/s write 10000/s query	2000/s	60G/220G
CRM application/7	session	400/s read and write	100/s	50m/140G
	Product configuration	100/s (local cache 5w/s)	100/s	1G/140G
	Order flow data	6000/s read and write	6000/s	35G/140G
	Order lock	7000/s write	7000/s	10m/140G
Billing Index/11	Global Index	20w/s write 20000/s query	6000/s	40G/220G

6. Conclusion

The distributed cache system proposed in this paper is a memory storage system with low latency, high performance, high availability, and high scalability, which supports the representation of complex business object. It greatly improves system performance by caching hotspot data in memory and quickly returning query results. Distributed deployment can horizontally expand cache capacity and further improve performance. By caching hotspot data, it is possible to reduce queries to the database, reduce database load, and avoid the database becoming a system bottleneck. Distributed cache nodes backup data to each other, and when a node fails, data can be restored from other nodes to improve system availability. It also supports fault isolation and automatic discovery and recovery of faulty nodes. In summary, the distributed caching system proposed in this article can meet the usage needs of various scenarios in large-scale production systems.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Lanying Shi and Chunhua Chen conducted the research; Chengwei Yang and Yiquan Jiang analyzed the data; Lanying Shi wrote the paper; Hongming Qiao and Kefeng Yu provided solution guidance; all authors

had approved the final version.

Acknowledgment

The first author thanks the finically support by key scientific and technological project form the China Telecom Research Institute (Project No. 23HQJ3YF0052-003). In addition, the first author would also like to thank the users of the Ctg-cache for their timely feedback and valuable suggestions.

References

[1] Pang, H., & Zhai, Z. L. (2011). On distributed database. *Database and Information Management*, *2*, 271–273

[2] Aghazadeh, S., & Leach, J. (2017). Evaluating cache memory performane: A case study on redis and memcached. *International Journal of Advanced Computer Science and Applications*, 23–28.

[3] Katsov, V. (2017). NoSQL data cache in ruby with redis. *Proceedings of the International Conference on Data Science, Technology and Applications (DATA)*

[4] Chhokra, D., & Sravani, G. (2018). Distributed caching techniques in cloud computing: A survey. *Proceedings of the 2018 International Conference on Smart City and Emerging Technology*.

[5] Liu, W., *et al*. (2019). A distributed cache consistency maintenance mechanism based on redis cluster. *Journal of Computer and Communications*, 61.

[6] Pająk, M., *et al*. (2019). Optimizing live migration of KVM virtual machines between data centers by compression and caching in Memcached. *Future Generation Computer Systems*, *101*, 555–568.

[7] Panchenko, I., *et al*. (2017). Prediction of load on virtual machines based on counts of queries to Redis database. *Proceedings of the 2017 20th Conference of Open Innovations Association (FRUCT)*.

[8] Gharaibeh, N. M., & Al-Naami, B. M. (2016). Caching policies for shared caches: New algorithms improving memcached performance. J*ordanian Journal of Computers and Information Technology (JJCIT)*, *2(2)*, 1995–6665.

[9] Li, G., *et al.* (2017). A distributed caching strategy based on topology in Memcached cluster. *Proceedings of the GLOBECOM 2017-2017 IEEE Global Communications Conference*.

[10] Yoon, D. H., *et al*. (2018). Efficient index caching in memcached. *Proceedings of the 2018 IEEE 34th International Conference on IEEE*.

Copyright © 2024 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0)