PAPER • OPEN ACCESS

Redis rehash optimization based on machine learning

To cite this article: Juan Zhang and Yunwei Jia 2020 *J. Phys.: Conf. Ser.* **1453** 012048

Redis rehash optimization based on machine learning

Juan Zhang[1*] **and Yunwei Jia**[2]

[1]Institute of information engineering, Anhui Xinhua University, Hefei, Anhui, 230088, China

[2]School of Communication and Information Engineering, Chongqing University of Posts and Telecommunications, Chongqing, 400000, China

[*]Corresponding author's e-mail: 12779302@qq.com

Abstract. This paper proposed an optimization for Redis rehash based on machine learning. Since Redis involves numbers of dimensions, only the process of Redis rehash was studied. In order to use Redis correctly, the principle of Redis rehash was analyzed firstly and then pointed out a feasible optimization scheme. Aiming at solving the problems that might exist in the rehash process, the scheme could predict the number of keys based on the time series model ARIMA. The result showed that the scheme was feasible in practice.

1. Introduction

With the rapid development of the Internet, more and more systems need to support high concurrency, so the caching system was invented and has become the standard in the Internet company system architecture. Among the various caching tools, Redis is the most popular and completely open source free one. It is a high-performance key-value database with BSD protocol [1]. Like other key-value cache products, Redis has three characteristics [2]: 1) Redis supports data persistence. It can save data in memory on disk and load it again when restarting. 2) Redis not only supports simple key-value type data, but also provides storage of data structures such as list, set, zset, hash, etc. 3) Redis supports data backup, i.e. master-slave mode, and cluster mode.

Recently, numbers of methods have been applied in this zone, and the machine learning optimization system is a research hotspot among them [3]. Researchers at Google and MIT [4] made a research on how to apply machine learning algorithms to index structures. It showed the exploration of the application of machine learning model and the core components of the data management system had a profound impact on future system design. However this work is only the tip of the iceberg for future development, and future applications of PostgreSQL, MySQL and redis will also be adopted [5]. The biggest selling point of Oracle, the world's largest database vendor, is autonomous database, an adaptive database that uses machine learning to automatically optimize the database to reduce DBA intervention. Major companies such as Facebook, Baidu, and Alibaba have their own Redis development teams, and have been optimizing Redis for their business scenarios. Academically, a new conference called SysML was established in the last two years, focusing on the intersection of machine learning and systems, not to mention more and more related papers [6], such as Carnegie Mellon University of Otter Tune.

Computer systems are full of empirical rules, and heuristics are used everywhere to make decisions. Using machine learning to learn the core parts of the system will make it better and more adaptive. This field is full of opportunities. At present, the industry has not found any application of machine learning on Redis. This paper will focus on this aspect. To optimize Redis, the knowledge of the core principles of Redis is necessary so that problems can be resolved in time. This paper proposed a feasible optimization scheme for the problems of Redis in the rehash process. And a software was established, which can not only monitor the running status and related parameters of Redis in real time, but also make early warning and artificially control the Redis parameters before Redis rehash occurs.

2. Principles of the Redis rehash mechanism

In order to fully grasp the principle of Redis rehash, this section first introduced the hash table, then explained how Redis implements the hash table, and finally introduced the principles of Redis rehash.

2.1. The hash table

The hash table (Figure 1) is a data structure designed to quickly access the corresponding data (value) through a key. It maps key codes to a position in the array through a hash function, and make use of the characteristics of fast search in the array, so as to have efficient search function. Since the length of the array is fixed, there will be different keys that may be mapped to the same position of the array, which will result in conflicts. Generally, open addressing method and zipper method are used to solve conflicts and zipper method is used in Redis. When a conflict occurs, the linked list is used to solve the conflict. When the hash function is better, all the keys will fall evenly to a certain position in the array. But when the number of keys increases, the probability of conflict will increase, and the length of the linked list will become longer, which will result in a decrease in query efficiency. In this case, the usual way to expand hash tables is to create a larger hash table than the original one, and then copy the data of the original hash table once. This process is also called rehash, which is used in many languages such as java.

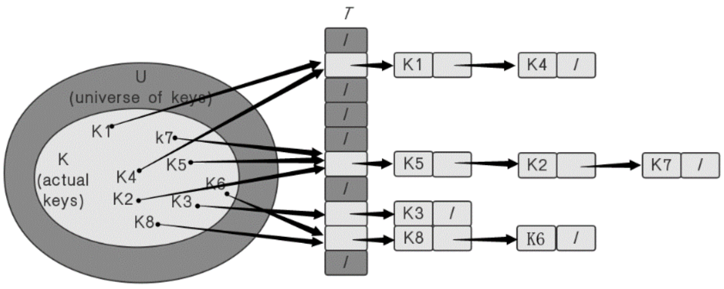

Figure 1. the hash table

2.2. Implementation of Redis Hash Table

To illustrate the process of Redis rehash, the data structure of the Redis hash table is necessary to learn firstly. The data structure of the Redis hash table is as follows:

```
typedef struct dictht
{
    dictentry **table;
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
} dictht;
```

Where dictentry is what we call an array, which can be seen as a one-dimensional array of dictentry*, size is the size of the array, sizemask is used to modulate the hash value, and used is the current number of elements in the current hash table. In the implementation of Redis hash table, the data structure is completed around. When there is a conflict, the linked list is used to resolve the conflict. The dictentry is a typical linked list node as follows:

```
typedef struct dictentry {
void *key;
union {
void *val;
uint64_t u64;
int64_t s64;
double d;
} v;
struct dictentry *next;
} dictentry;
```

This is a typical linked list node, with key and value, and a pointer next to the next node.

2.3. Redis rehash

The implementation of the hash table in Redis is introduced above. The general language also supports hash tables, such as java hashmap, python dict, etc. In the process of rehash, the principles described in Section 1 are generally the same. However, this method is not suitable to apply in Redis, because Redis is an online service. If all the data is copied at once in the process of rehash, it may take a long time and cause Redis blocking. Therefore, the process of progressive rehash is adopted in Redis. It is not a one-time copy of all, but a gradual migration. This method distributes the process of copying node data to subsequent operations instead of one-time copying. The so-called flattening to subsequent operations is to copy the node operations, such as reinserting, searching, deleting, and modifying.

To achieve this process, a hash structure must have the following fields: two hash tables and one representation for whether or not to rehash. When the first hash table ht[0] of the hash structure reaches the expansion condition, the rehash is started. At the same time, the new space will be re-adjusted and allocated, and the second hash table ht[1] points to this space. The data structure is as follows:

```
    typedef struct dict{
    dicttype *type;
        void *privdata;
        dictht ht[2];
        int rehashidx;
        int iterators;
    } dict;
```

The method of the Redis rehash process is to use two hash tables ht[0] and ht[1] for expansion and transfer. When a hash table reaches the condition of expansion, another hash table is allocated twice the size of the first hash table, and the element of the first index position of the first hash table is shifted to the second table. Then, every time the element is inserted and fetched, the elements in the next index position are sequentially migrated until the migration is completed. As can be seen from the above, when Redis triggers Resize, it dynamically allocates a block of memory, which is finally pointed to by ht[1].table. The dynamically allocated memory size is: Realsize*sizeof(dictentry*), table points to a pointer of dictentry*, in which the size is 8bytes (64-bit OS), that is, the memory size to be allocated by ht[1].table is: 8*2* 2^n (n is greater than or equal to 2).

Compare the hash table size with memory request size:

Table 1. Formatting sections.

Hash table size	Memory required by rehash
4	64bytes
16777216	256M
33554432	512M
67108864	1G

3. Problems with the Redis rehash Mechanism

Based on the above analysis, Redis temporarily allocates a bucket when rehashing, and the bucket size is twice as large. If there are more keys, it will occupy a large part of the memory space, which will be calculated into the total memory of the system. At this time, this part of the space can not be ignored, and it will cause Redis short-term unavailability in some specific scenarios, even Redis can not be used. Considering the following scenario, when Redis is used as a cache, the data clearing strategy will be applied. As the memory is insufficient, rehash occurs. At this point, the memory will exceed the maximum memory. Redis will expel the key to release the memory, so that the amount of memory is lower than the maximum memory. If the memory used in the rehash process is relatively large, then Redis will evict a large number of keys, as shown in Figure 2. Since redis is single-threaded, the eviction process is blocked, which will cause Redis to be unable to respond to client requests, resulting in online failures. If Redis is in master-slave mode at this time, it will cause Redis master-slave to switch. The process of switching will trigger the Redis master-slave replication process. If Redis memory is large, the master-slave replication will be a long process. which has been encountered many times in practice.

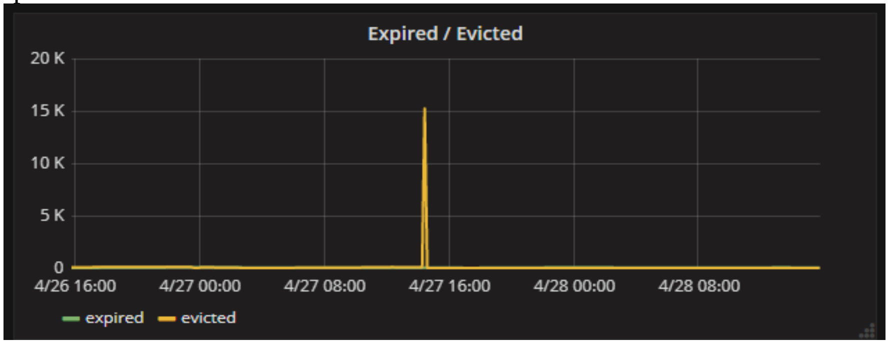

Figure 2. Redis jam

4. Optimization and Implementation

It can be seen from the analysis that in the near full or full state, if rehash occurs, Redis will be temporarily unavailable, affecting online services. The key to the problem is that when rehash happens, the memory has already exceeded the maximum memory set by Redis. Most companies solve this problem by modifying the source code nowadays. If there is insufficient memory when rehash occurs, forbid rehash, and wait for the appropriate time to restart rehash. This approach can solve the problem, but it also brings many other problems, requiring maintenance of Redis version. In addition, if the number of keys does not decrease, the performance of redis will decrease. Therefore this paper proposed a new scheme to predict the number of keys in Redis in advance and a time interval of a Redis rehash, then take corresponding measures based on previous experience, such as automatically increasing the memory, and recover the memory at an appropriate time, or notify the operators to make decisions

Based on the above analysis, this paper designed a system consisting of a client and a server (Figure 3). The client is installed on the machine where Redis is located. The number of keys in Redis is collected and uploaded to the server. The server is responsible for accepting the client's data and storing it. Then, the training information and the recommended results are recorded in the database through the machine learning model. Finally, the results are displayed to facilitate the maintenance personnel to view, or directly send instructions to the target Redis, dynamically control the parameters of Redis. The architecture of the whole system is showed as follows:

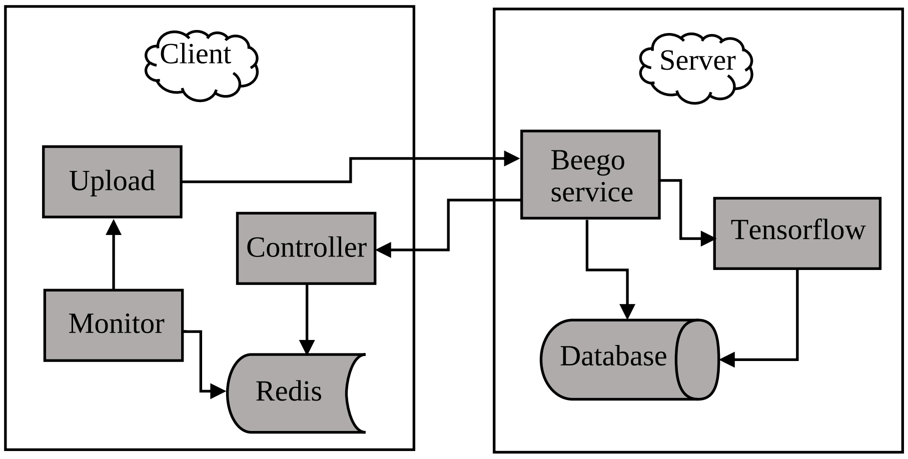

Figure 3. Redis optimization system

The whole system is developed by golang, and the client is mainly composed of monitol, upload and monitor. The monitor module mainly sends the info command to the Redis through the Redis client to monitor the current number of Redis keys, and interacts with the server through the upload module to periodically report the number of keys. The monitol module is mainly responsible for accepting commands from the server, and then controlling Redis related parameters to optimize the performance of Redis. The client design is very simple, and the biggest difficulty of the system is still on the server side.

Firstly, the server uses beego to build the website. Beego is a http framework for rapid development of Go applications. It can be used to quickly develop various applications such as API, Web, and back-end services. It is a RESTFul framework, and the main design is inspired by tornado. The main design inspiration comes from three frameworks: tornado, Sinatra and flask. But beego is a framework designed to combine some of Go's own features (interface, struct inheritance, etc.), which is a highly decoupled framework for quickly developing a background system.

Secondly, the system implemented tensorflow to train the machine learning model. TensorFlow is an open source software library using data flow graphs for numerical calculation. Nodes represent mathematical operations in the diagram, while the edges in the graph represent multidimensional data arrays that are interconnected between nodes, i.e. tensors. The flexible architecture allows you to deploy computing on a variety of platforms, such as one or more CPUs (or GPUs), servers, mobile devices, and so on. TensorFlow was originally developed by researchers and engineers from the Google Brain Group (part of the Google Machine Intelligence Research Institute) for research in machine learning and deep neural networks.

Finally, the key is to select an appropriate time series model. The system chooses ARIMA model (ARIMA model, differential integrated moving average autoregressive model, also known as integrated moving/sliding average autoregressive model, one of the time series predictive analysis methods). ARIMA consists of three parts, which is AR, I, MA. AR represents auto regression, i.e. autoregressive model; I represents integration, i.e. single integer order. Time series model must be stationary sequence to establish econometric model, ARIMA model is no exception as time series model. Therefore the unit root test should be carried out on time series at first. If it is a non-stationary sequence, it should be converted into a stationary sequence by difference. MA represents the moving average, i.e. the moving average. model. It can be seen that the ARIMA model is actually a combination of the AR model and the MA model. The difference between the ARIMA model and the ARMA model is that the ARMA model is a stationary time series model. The ARIMA model is modeled for non-stationary time series. In other words, to establish an ARMA model for a non-stationary time series, the first step is to transform into a stationary time series and then build an ARMA model to predict the number of keys, store them in the database, and finally display them through the page. The system has been used in practice, and has successfully predicted the time interval when the key occurs rehash, successfully alarmed and automatically adjusted parameters.

5. Conclusion

In this paper, a feasible solution was proposed for the problems caused by the Redis rehash process, and the feasibility of the scheme is verified by designing a complete system. The biggest innovation of the scheme is the combination of machine learning algorithms, which can also be used to solve other problems. This aspect is a research hotspot in the industry, which has been used in optimizing databases by many research institutions.

Acknowledgement

This work was supported by the Teaching Research Program (2018jyxm1107) and Brand Professional Cultivation Project (2018ppzyx03).

References

[1] Fu L, Zhang YJ. (2017) Redis DevOps. China Machine Press Publishing, Beijing.
[2] Wu pz. China. (2013) Storing hundreds of millions of simple key-value pairs in Redis. https://blog.csdn.net/wupangzi/article/details/13996785.
[3] Fonseca A, Cabral, Bruno. (2017) Prototyping a GPGPU Neural Network for Deep-Learning Big Data Analysis. J. Big Data Research, 8:50-56.
[4] Kraska T, Beutel A, Chi EH. (2018) The Case for Learned Index Structures. In: International Conference on Management of Data. Houston. pp. 489-504.
[5] Li CQ, Gu JH. (2019) An integration approach of hybrid databases based on SQL in cloud computing environment. J. SOFTWARE-PRACTICE & EXPERIENCE, 49（3）: 401-422.
[6] Cancer Research UK. (1975) Cancer statistics reports for UK. http://www.cancerreseark.org/aboutcancer/statistics.