Research Article

Open Access

Zhuo Zhang*

Dynamic load balancing algorithm for large data flow in distributed complex networks

<https://doi.org/10.1515/phys-2018-0089>

Received July 6, 2018; accepted August 19, 2018

Abstract: Information society brings convenience to people, but also produces a lot of data. Relational databases are not suitable for processing big data due to architecture defects. The most commonly used system to store and process large amounts of data is the NoSQL (Not only Structured Query Language) database. Obviously, it is very important to cooperate with these independent computers to accomplish processing tasks efficiently, which is the function of load balancing. This paper studies the commonly used NoSQL database and load balancing algorithms, and designs and implements a more efficient load balancing algorithm. By introducing the relationship between nodes and the children of their brother nodes, we reduce the height of the whole sorted binary tree. The time cost of the algorithm is reduced versus the commonly used weighted polling algorithm $O(N)$ to $O(\log N)$, while the spatial cost remains unchanged. The equalization algorithm synthetically utilizes the characteristics of big data processing systems and has good performance. At the same time, the algorithm can quickly find the sub-optimal nodes when the optimal nodes have been occupied, so it is very suitable for load balancing in highly concurrent systems. Finally, the effectiveness of the proposed load balancing algorithm is verified by simulation.

Keywords: NoSQL databases, load balancing algorithm, distributed Mass Data Storage System, sorted binary tree

PACS: 05.90.+m, 07.05.Tp

1 Introduction

For a long time, relational databases have been the main tools for data storage. Relational databases were first put forward by IBM researcher F. Codd. Their theoretical basis is relational algebra, which was then gradually developed

into a more mature relational database supporting transaction processing. Before the 1990s, the number of visits to a website was generally small, and mainly to static web pages. Relational databases became the mainstream storage technology [1].

With the rapid development of Web2.0, the amount of information generated by the Internet is growing rapidly. Search engine companies, such as Google, and social network companies have produced huge amounts of data information. However, the traditional relational database architecture cannot meet the needs of large quantities of processing [2]. First of all, the extensibility of the relational database schema is insufficient, and it is not suitable for big data storage. Because reading data from memory is much faster than reading data from disk, traditional relational databases speed up data access by putting indexes and caches into memory. Then a complex optimization algorithm is used to avoid reading the data directly from the disk and returning it to the client after receiving the user's request [3]. But this approach is often ineffective in big data processing, because big data indexes tend to be several orders of magnitude larger than the average server's memory capacity. Disk access is often the bottleneck of the whole system, which greatly reduces the response speed of the system. Second, although the use of mainframes or supercomputers can to some extent make up for the shortcomings of the above methods, they require high operating and maintenance costs [4]. It's hard for most free Internet applications to afford such a high cost. At the same time, having a mainframe as a centralized system does not provide as strong robustness and availability as distributed systems, and downtime accidents will seriously affect user experience. In the face of more and more homogeneous competition, this disadvantage can be said to be fatal.

Finally, big data processing systems tend to simply read and write data without the need for more complex features such as ACID (Atomicity, Consistency, Isolation, Durability), which provides these functions at the expense of performance degradation [5]. Since databases have become synonymous with storage systems, they are often referred to as NoSQL databases. In less than a decade, there have been dozens of NoSQL database products. Almost

*Corresponding Author: Zhuo Zhang: Department of Statistics, Heihe University, Heihe, China, E-mail: 19492884@qq.com

all large Internet companies use NoSQL database as one of their databases. Even Oracle, a relational database giant, has developed NoSQL database products based on Hadoop. The NoSQL database has become an important supplement to relational databases.

Load balancing is the distribution of workload between multiple computers, clusters of computers, or other resources to achieve optimal resource utilization, maximum throughput, minimum response time, and avoidance of overload. It is widely used in network systems and distributed systems. Load balancing is an important part of the NoSQL system [6]. How to design an efficient load balancing algorithm to deal with a large number of data requests is the key to the performance of the whole system. A good load balancing algorithm should select the most suitable node to handle the request according to the performance difference of each node in the system.

The common algorithms in commercial systems are the random method, polling method, and fastest connection method. In addition, even some artificial intelligence algorithms such as the genetic algorithm are still under study. However, such algorithms are either too complex or too difficult to implement, and the performance advantages are limited, or rough, failing to take into account the speed and performance of the node itself [7]. In this paper, we design a new algorithm to make the value of the largest weight always in the position of the first element without traversing the whole array to find the node with the maximum weight, which maintains the data structure while requiring less time cost.

2 A Survey of distributed mass data storage systems and load balancing algorithms

2.1 Summary of distributed mass data storage system

As mentioned above, the development of distributed mass data storage systems is the inevitable result of the development of the information age. Due to the limitations of hardware technology, the frequency of CPUs is closer to the theoretical limit of 4GHz under the existing architecture, and the space for performance improvement is not much while the cost is huge. Centralized systems face great challenges in dealing with rapidly expanding data storage and processing. Meanwhile, distributed systems have attracted much attention because of their high cost performance,

good fault tolerance and robustness. On the other hand, relational databases also face great challenges. For example, when Internet applications produce huge amounts of data which can't even be indexed into memory, traditional relational databases can't do anything about it. And Internet applications often surge in a large number of visits in a short period of time with social hot spots and festivals, but these applications do not have high requirements for consistency [8]. Poor support for scalability can significantly increase overhead and response speed in relational databases. Therefore, a new distributed mass data storage system emerges as the times require. It makes full use of the advantages of the distributed system, and achieves high response speed and availability through cooperation between independent nodes. Because databases have been synonymous with storage systems for decades, these distributed mass data storage systems are often referred to as NoSQL databases.

At present, there is no uniform definition of NoSQL database, which is generally considered as the general term for any non-relational database. A NoSQL database can be described as follows:

- 1) Extensible loosely coupled data model
- 2) Cross-node data distribution, horizontal dynamic expansion
- 3) Data persistence capability in memory or disk
- 4) Supporting non-SQL statement interfaces for data access

The results of the comparison of NoSQL databases which satisfy true distribution and automated fragmentation are shown in Table 1. Many NoSQL databases are not included because they do not meet these Ellis prerequisites.

Table 1: Classification of NoSQL

Data storage	Support for transparently added machines	Multi-data center support
Cassandra	No	No
HBase	No	Yes
Riak	No	Yes
Scalaris	No	Yes
Voldemort	Yes	Need some code

We believe that persistence design is important for good runtime of a database. Memory databases are fast but easily lost. This strategy is fast and avoids the vulnerability of pure memory databases. The B-tree can provide ro-

bust index support in the database, but it does not perform well due to the need to read and write more disks. The document database CouchDB uses a B-tree and only appends writes, thus avoiding excessive disk IO.

2.2 Distributed mass data storage system

Memcached is a distributed cache system developed by danga.com, which is used to dynamically reduce database load and improve system performance. It is widely used in NoSQL databases. Memcached implements network connections based on the libevent library and itself provides services as a stand-alone application or daemon. It uses a simple protocol for communication, built-in memory storage for storage [9]. The NoSQL data model and query API are given in Table 2.

Table 2: NoSQL data model and query API

Database	Data model	Query API
Cassandra	List	Thrift
CouchDB	Documents	Map/reduce view
HBase	List	Thrift, REST
MongoDB	Documents	cursor
Neo4j	Diagram	Diagram
Redis	set	set
Riak	Documents	Inline hash
calaris	Key/value	Get/put
Tokyo Cabinet	Key/value	Get/put
Voldemort	Key/value	Get/put

Table 3: Sorting NoSQL with persistent storage

Database	Durable design
Cassandra	Memtable / SSTable
CouchDB	Append B tree
HBase	Memtable / SSTable on HDFS
MongoDB	B tree
Neo4j	Disk link list
Redis	RAM
Riak	uncertain
calaris	RAM
Tokyo Cabinet	Hash or B-tree
Voldemort	Pluggable (mainly BDB MySQL)

Sorting NoSQL with persistent storage is given in Table 3. Memcached is usually used as a cache for the front end of the database, and is used only to manage data

in memory, not to include more time-consuming operations such as SQL parsing and disk operations. Therefore, it can provide better performance than direct reading of traditional relational databases, and it is widely used in mass data storage systems. In addition, Memcached is often used as a medium for sharing data between servers [10], shared by multiple applications. Because Memcached uses memory to store and manage data, other means are needed to persist the data. At the same time, memory is very expensive and storage space is extremely limited, so using this way to store data is not extensible.

A key-value storage database is a database where data is stored in memory and disk as keys and values. There is a common simple data model for key-value storage: a mapping/dictionary that allows clients to request and push values through keys. One of the more famous databases for key-value storage is AmazonDynamo. It was used for many purposes along with other databases at Amazon, and as one of NoSQL's earliest products had a significant impact on later NoSQL databases.

Dynamo is an extensible, highly available key-value storage system developed internally by Amazon. Instead of being exposed directly to the outside, Dynamo is used to support a portion of Amazon's web services. Many of the technologies used in Dynamo are based on the research of distributed systems and operating systems in the past few years. The techniques used by Dynamo and their advantages are shown in Table 4.

The advantages and disadvantages of Dynamo are summarized in Table 5.

2.3 Commonly used theory

2.3.1 Consistency theory

CAP means consistency, availability and partitioning fault tolerance. According to CAP theory, at most, only two of the three can be strictly satisfied. Consistency is whether and how the system is in a consistent state after performing an operation. If all read operations after a write operation can see their update operations in shared data resources, the distributed system can be considered consistent. Availability, or high availability, means that a system is implemented in a way that allows continuous operation, even if some nodes or hardware fail due to upgrades [11]. Partitioning fault tolerance refers to the ability of the system to support continuous operation in the case of network partitioning. This occurs when two or more isolated nodes are not connected. Others think that partitioning

Table 4: Dynamo Use Technology and Benefits

Problem	Solution	Advantage
Partition	Consistency hash	Improve scalability
High-availability write operations	Vector Clock	Version decoupling
Handling temporary failure	Loose election mechanism	High availability and durability
Recover from persistent failure	Anti-entropy	Background consistency replication
Failure detection	Heartbeat protocol	Avoid centralized registration

Table 5: The advantages and disadvantages of Dynamo

Advantages	Shortcomings
No master	Non-open source software
Supports high availability write operations	Clients need smarter
Read or write tuning	No compression
Simple	Do not support unified operation of columns
	Do not support range queries and batch operations

fault tolerance refers to the ability to dynamically add or remove nodes. CAP applied to the system is given in Table 6.

These applications often improve availability and reliability through a certain degree of redundancy.

2.3.2 Partition fault tolerance

If the data in a large-scale system exceeds the capacity of a single machine, then the system needs to consider replication to ensure reliability, load balancing and data partitioning to improve scalability. Depending on the size of the system and other factors, there are several different options [12].

This can be viewed as a partitioned memory database. These systems copy used data frequently into memory and then distribute it to the client, thus avoiding a large amount of disk IOs and significantly improving efficiency. In the case of Memcached, the memory cache contains a set of processes that have allocated memory, and these servers connect by configuring the network. The Memcached protocol, which can be implemented in multiple languages, provides a simple key-value storage API. It hashes objects that need to be stored into the configured in-memory cache instance. If a node is unable to connect due to a hardware failure, the object needs to be relocated through a secondary hash [13]. Because these are all in-memory operations, they are fast. Because memory bars are much more expensive than disks, they are expensive, or are not extensible at a certain cost. Database server clustering is another way of transparently partitioning data.

However, this approach can only be used based on distributed database management systems with poor performance [14].

2.4 Load balancing

Load balancing is a set of methods to adjust and control resources in a distributed system to improve the performance of the system. It has a significant impact on the performance of the system under the established hardware environment. Load balancing mechanisms are typically triggered during initialization and redistribution [15]. The load balancing algorithm can either separate the two or deal with them together. The mechanisms of the two methods are similar, and the redistribution phase requires more load balancing. This paper focuses on the re-distribution phase of the load balancing mechanism.

Load balancing can be classified differently according to different criteria. For example, according to the implementation method, it can be divided into software and hardware load balancing. According to the scope of load balancing, it can be divided into local and global load balancing. It can also be divided into link layer, transport layer and application layer load balancing [16].

1) Software and hardware load balancing

Software load balancing uses a software method to deal with load balancing, while hardware load balancing uses a hardware method. Software load balancing serves to distribute the traffic through a certain load balancing algorithm at the interface of the distributed software system to realize a reasonable flow of the traffic. Hardware

Table 6: CAP applied to the system

Select	Features	Example
Consistency, availability (sacrificing partition fault tolerance)	Two-phase commit, cache verification protocol	Single database, cluster database, LDAP, xFS file system
Consistency, Partition Fault Tolerance (Sacrificing Availability)	Negative lock, making a few partitions unavailable	Distributed Databases, Distributed Locks, Most Distributed Protocols
Availability, Partition Fault Tolerance (Sacrifice Consistency)	Conflict resolution, optimistic locking	Coda, DNS

load balancing by connecting a load balancer is responsible for forwarding and regulating network traffic. Because the hardware equipment has the characteristics of high speed, stability, reliability and so on, the backstage of a large website generally uses this load balancing method to control access.

2) Local and global load balancing

Technology for load balancing by region can be divided into partial load balancing and global load balancing. Partial load balancing consists generally of the dynamic distribution of load in LAN, and most of the load balancing belongs is of this type at present. Because of the high speed of local area networks, we can think of network delay as a secondary factor. At this time, the main consideration is the matching degree between the task and the server node itself. On the other hand, global load balancing is used to distribute the load at multiple sites in a cross-region.

3) Network layer load balancing

This method distributes different packets by parsing network protocol packets. At present, it is mainly divided into link layer, transmission layer and application layer balancing.

3 Design and implementation of load balancing algorithm

3.1 Load problem description

Due to the limitation of CPU frequency, memory reading speed and Von Neumann system structure, the performance of a single computer is limited to a certain extent, so people use multiple computers to improve performance. Distributed systems can be divided into three types: shared memory, shared disk, and no shared resources, as shown in Figure 1.

Multiple processors of a shared memory system are connected over a network and have access to common

memory. Shared disk storage systems are such that each processor has its own independent memory, but can directly access all hard disks over the network. Without a shared resource system, each processor has its own memory and disk, and there is no common resource, and all communication between different processors is done over the network. Shared memory and shared disk structure are prone to resource conflicts. With increase in number of processors, system performance improvement is limited. Non-shared resource systems are considered to be the best structure for scalable large-scale data processing systems, obtaining approximately linear performance improvement.

Load balancing is mainly responsible for adjusting the load between partition servers so that the resources managed by each load can be balanced as much as possible. When the load is not balanced, the task handled by the heavier node can be dynamically transferred to the lighter load node [6]. Load balancing plays an important role in distributed systems. Application scenarios typically focus on communication within the same data center, that is, connections between these servers over local area networks, regardless of the connection between servers over a wide area network when communicating across data centers. Therefore, it is generally believed that network transmission speed will not become the bottleneck of the system.

3.2 Design of load balancing algorithm

The key for load balancing algorithms is to design an algorithm to quickly and accurately find the node with light load and assign the task to such node. As mentioned earlier, static algorithms ignore the differences between nodes and the load tilt caused by task execution, which has obvious efficiency disadvantages compared with dynamic algorithms [8].

The balanced tree is a data structure commonly used in computer science. It usually adjusts the structure of the

Figure 1: Three common structures of distributed databases

general ordered binary tree when it meets some conditions by certain constraints to keep the relative equilibrium between its left and right subtrees. The time cost of operations such as insertion and deletion of binary tree is closely related to the height of the tree. Therefore, the key to improving the efficiency of the binary tree is to add appropriate constraints to maintain balance of the binary tree through simple and efficient operation.

In this paper, a constraint condition is introduced for binary trees by combining balanced binary trees with the Fibonacci sequence. The cost of time $O(\log N)$ is reduced, and only a few operations are required to make binary trees that do not satisfy this condition satisfy this constraint condition. The algorithm stores node information in the ordered binary tree, adds an element to the sorted binary tree to count the number of nodes, and then rotates the tree to the left and right, so that the binary tree always keeps the height of $O(\log N)$.

For the convenience of the following description, define as follows: T means node, $\text{left}[T]$ means left child of T , $\text{right}[T]$ means right child of T . $\text{size}[T]$ denotes the number of nodes in a subtree of which T is the root node. In addition to satisfying the properties of ordinary ordered binary trees, we also satisfy the following two properties:

$$\begin{aligned} & \text{size}[\text{right}[T]] \\ & \geq \max \{ \text{size}[\text{left}[\text{left}[T]]], \text{size}[\text{right}[\text{left}[T]]] \} \end{aligned} \quad (1)$$

$$\begin{aligned} & \text{size}[\text{left}[T]] \\ & \geq \max \{ \text{size}[\text{right}[\text{right}[T]]], \text{size}[\text{left}[\text{right}[T]]] \} \end{aligned} \quad (2)$$

Size $[T]$ and these two properties are very important for reducing the height of the ordered binary tree, which is the key to improving the efficiency of the algorithm. In the worst case, the common ordered binary tree is reduced to a linked list, and the time cost is the same as the time cost of polling in the worst case. It will be shown later that by introducing the above two properties, $O(\log N)$ can make

the sorted binary tree as high as possible in any case. The above two properties are not always satisfied in the establishment of ordered binary trees. Through the Maintain operation, the previously unsatisfied nodes satisfy these two properties. Because these two properties are symmetric, there are two situations:

1) Situation 1

$$\text{size}[\text{left}[\text{left}[T]]] > \text{size}[\text{right}[T]] \quad (3)$$

It is assumed here that T 's left and right children are satisfied with the above properties. Assuming that a node is inserted so that $\text{size}[A] > \text{size}[R]$ is in violation of Property 1. A node which is not satisfied with that property can be made satisfied by the right-hand rotation operation shown in Figure 2. If the T node or L node of the binary tree does not satisfy the above two properties, then the Maintain operation is performed on them again. The size values of the T node and L node are recalculated. It is clear that after the Maintain operation, all nodes satisfy the above two properties.

Figure 2: Right-handed operation of T node

2) Situation 2

Assuming that the insertion node causes $\text{size}[B] > \text{size}[R]$ to violate Property 1, then we first perform a left-handed operation on the L node and then a right-handed operation on the T node. The left-handed operation of the L node has been discussed in the case of 2) as shown in Figure 3. At this point, the B subtree may satisfy Property 1, and sometimes this Maintain operation needs to be performed on the B point. At this point, all nodes satisfy two properties.

Figure 3: Left-handed operation of L node

We organize the nodes into one of the above binary tree structures so that the most suitable node can be taken out at a time cost to execute the load balancing plan. When the power value changes, the binary tree can also be maintained over time by swapping with the left child or the right child. Adding or deleting nodes also takes time to complete.

At initialization, the binary tree is sorted in the normal way, and then the nodes that do not satisfy Properties 1 and 2 are rotated. When the data structure is set up, only a few nodes need to be adjusted, so it is not necessary to reorder all the nodes, but only to adjust the individual nodes. The time cost of reordering is maximum when using the old polling algorithm, but the whole time cost of this algorithm is what is required, and it may be much smaller than this worst time cost in practice.

3.3 Common operation of algorithm

The key for this algorithm is to count the number of nodes and limit the height of the binary tree by the constraint relationship between the number of nodes. The pseudo code

for the statistics node size is as follows:

```

if size[root] = NULL
    return 0
return size[left[root]] + size[right[root]] + 1

```

(4)

When the two introduced properties are not satisfied, the structure of the tree is changed by the Maintain operation to re-satisfy the condition. The operation is a recursive operation—that is, the operation is premised on the child nodes satisfying Properties 1 and 2. The pseudocode for left rotation and left rotation in Maintain operations is as follows:

```

node ← right[root]
right[root] ← left[node]
left[node] ← root
size[node] ← size[root]
size[root] ← size[left[root]] + size[right[root]] + 1
root ← node

```

(5)

The pseudocode for RightRotateroot to the right is as follows:

```

node ← left[root]
left[root] ← right[node]
right[node] ← root
size[node] ← size[root]
size[root] ← size[left[root]] + size[right[root]] + 1
root ← node

```

(6)

The pseudocode for maintaining operations, Maintainroot, for Properties 1 and 2 is as follows:

```

If size[left[left[root]]] > size[right[root]] then
    Right - Rotate(root)
    Maintain(right[root]), Maintain(root)
If size[left[right[root]]] > size[right[root]] then
    Left - Rotate(left[root]), Right - Rotate(root)
    Maintain(left[root]), Maintain(right[root])
    Maintain(root)
If size[right[right[root]]] > size[left[root]] then
    Left - Rotate(root)
    Maintain(left[root]), Maintain(right[root])
If size[right[left[root]]] > size[left[root]] then
    Right - Rotate(right[root]), Left - Rotate(root)
    Maintain(left[root]), Maintain(right[root])
    Maintain(root)

```

(7)

The pseudocode for looking for a precursor, FindPredecessorroot, looks like this:

```

If left[root] ≠ NULL
  while right[left[root]] ≠ NULL
    left[root] ← right[left[root]]
  return left[root]
y ← parent[root]
while y ≠ NULL and root = left[y]
  root ← y
  y ← parent[y]
return y

```

(8)

In high concurrency big data processing systems, it is also very important to find the sub-optimal and the third optimal nodes in order to improve the throughput when the optimal nodes have already been allocated. The pseudocode for finding the k maximum value FindRank(root, k) is as follows:

```

If  $k > \text{size}[\text{root}]$ 
  return NULL
If  $k \leq \text{size}[\text{right}[\text{root}]]$ 
  return FindRank(right[root],  $k$ )
else if  $k = \text{size}[\text{right}[\text{root}]] + 1$ 
  return root
else FindRank(left[root],  $k - \text{size}[\text{right}[\text{root}]] - 1$ )

```

(9)

In short, using the above operations, you can maintain a more balanced binary tree in time.

3.4 Implementation of load-balancing weighting algorithm

Load balancing evaluation index has an important impact on load balancing. A good evaluation index can accurately reflect the system load situation and processing ability. Common load metrics include processor occupancy, processor ready queue length, main memory usage, IO speed, and so on.

The load balancing algorithm is usually in the load balancer, where the trigger condition of load balancing is satisfied, and the load balancer finds out the responding node to schedule the load according to the load balancing algorithm. The communication between nodes can be done by means of TCP or UDP. when the client's read-write request arrives, the load balancer distributes the load in real time according to the load of the load server and the balancing strategy. The communication between the client and the load server is forwarded through the load balancer. The advantage of this method is that it is convenient for centralized management and easy to config-

ure. The weakness is that the requirements of load balancer are high and it's easy for the the equalizer to become the bottleneck of the system when the task is large. The client first communicates with the load balancer to obtain the address of the load server that can process its requests. The load balancer selects an appropriate balancing server based on the state information and load balancing policy of the maintained load balancer and returns the IP address and port number of the load server to the client. The client then establishes a communication connection directly with the load server. The advantages of this approach are ease of expansion, good robustness, and the lack of system bottlenecks in the architecture itself. Therefore, this method is the mainstream method to achieve load balancing; the test system of this paper also adopts this structure.

Figure 4 is a general frame diagram of the load balancing algorithm. When the load is tilted, the load balancing plan is executed, the optimal load node is selected to process the request, and finally the data transfer and meta-data modification are carried out.

Figure 4: Load balancing algorithm flow chart

4 Test and analysis

4.1 Test environment

The load balancing algorithm and memory optimization described in this paper are applied to the distributed mass data storage system developed by the Institute of Computing of the Chinese Academy of Sciences. In order to test the performance of the algorithm and memory optimization, a distributed system platform is built. The architecture of

the system is shown in Figure 5, and the IP addresses of the primary server and partition server are shown in Table 7.

Figure 5: Test System Architecture

Table 7: text

Name of server	LAN IP address
Master server	192.168.0.121
Partition server 1	192.168.0.134
Partition server 2	192.168.0.135
Partition server 3	192.168.0.136
Partition server 4	192.168.0.137

The whole distributed system is divided into three parts: server client, primary server and partition server. Metadata storage and distributed lock management is a cluster of primary server nodes in the metadata storage subsystem in Figure 5. This mainly provides a metadata persistence storage function, metadata consistency maintenance function, and metadata access management function based on the above mentioned metadata.

4.2 Improved binary tree performance test

The core of this algorithm is to maintain the balance of the binary tree by left rotation and right rotation operations, so as to ensure finding at $O(\log N)$ the node with high weight in any data application. One of the important differences between this algorithm and other balanced binary trees is the introduction of a simple constraint relationship so that the binary tree needs only a few operations to maintain a better balance. The balance of a binary tree can be roughly measured by the average depth and height. The binary tree test flow chart is given in Figure 6.

First, 2000000 random data are generated and inserted by the random number generator, and then the efficiency differences between the data structure designed by

Figure 6: Binary tree test flow chart

this algorithm and other classical binary tree data structures are compared, as shown in Table 8. You can count the number and average depth of rotation by setting global variables. Because rotation is a more complex operation, the more the rotation times are, the more unbalanced the binary tree is, so the algorithm with less rotation times is often more efficient. Because the time cost of many operations of a binary tree is related to its height, the smaller the height or depth of the tree is, the faster the query, insertion, and deletion of the binary tree is.

It can be seen that this data structure has some advantages compared with other classical data structures in terms of the height of the tree and the number of times of rotation. It is only slightly higher than the AVL tree in the number of times of rotation, and the overall efficiency is very high. The effect is satisfactory.

4.3 Memory optimization test

In the process of reading and writing the system, the memory detection tool Valgrind is used to detect the memory and find that there are a lot of memory fragments in Cell-Cache. The management of CellCache through memory pool can improve the efficiency of the system. The memory fragment size usually ranges from a few bytes to a few hundred bytes, so the test environment for this article changes the input data to a large number of small data repeated insert operations. A 40 tasks performance comparison chart is given in Figure 7.

After testing, there has been a significant increase in efficiency with the optimized memory pool rewriting configuration, as shown in Figure 8. The two lines in the fig-

Table 8: The proposed Binary Tree is compared with other binary trees

Types	Binary Tree	AVL tree	Treap	Random binary tree	Stretching tree
Average depth	19.35	19.58	26.54	25.68	38.2
height	25	26	51	56	81
Number of rotations	1.58 million	1.55 million	4.01 million	4.03 million	25.21 million

Figure 7: 40 tasks performance comparison chart

ure represent the default allocation of STL, the insert process under the four allocation modes directly using memory pool, and the comparison of memory usage. As you can see, memory fragmentation on the partition server has been significantly reduced through memory pool management of CellCache.

Figure 8: Comparison of results before and after memory optimization

As seen from Figure 7 and Figure 8, using the proposed load balancing algorithm can effectively reduce the system memory footprint. Simulation validates the effectiveness of the proposed method.

5 Conclusion

Based on the research into the NoSQL database and the current mainstream load balancing algorithm, this paper designs an efficient load balancing algorithm for mass data storage systems. On the basis of a polling algorithm and weighted algorithm, the algorithm adds information about the number of statistical nodes. The height of the sorted binary tree is maintained at $O(\log N)$ by rotation operations, so that the time cost of accessing the node can be reduced from the time cost of the node to $O(\log N)$, which is helpful to improving the throughput of the big data system. The algorithm can also recursively find sub-optimal nodes and their successors and support load balancing of high concurrent systems and parallel processing systems.

Acknowledgement: Supported by the Natural Science Foundation of Heilongjiang Province of China (Grant No.A201413).

References

- [1] Cui L., Yu F.R., Yan Q., When big data meets software-defined networking: SDN for big data and big data for SDN, *IEEE network*, 2016, 30, 58-65.
- [2] Kreutz D., Ramos F.M., Verissimo P.E., Rothenberg C.E., Azodolmolkly S., Uhlig S., Software-defined networking: A comprehensive survey, *Proceedings of the IEEE*, 2015, 103, 14-76.
- [3] Jo M., Maksymuk T., Strykalyuk B., Cho C.H., Device-to-device-based heterogeneous radio access network architecture for mobile cloud computing, *IEEE Wireless Communications*, 2015, 22, 50-58.
- [4] Shaofei W., A Traffic Motion Object Extraction Algorithm, *International Journal of Bifurcation and Chaos*, 2015, 25, 18-27.
- [5] Capitanescu F., Ochoa L.F., Margossian H., Hatziaargyriou N.D., Assessing the potential of network reconfiguration to improve distributed generation hosting capacity in active distribution systems, *IEEE Transactions on Power Systems*, 2015, 30, 346-356.
- [6] He X., Ai Q., Qiu R.C., Huang W., Piao L., Liu H., A big data architecture design for smart grids based on random matrix theory, *IEEE transactions on smart Grid*, 2017, 8, 674-686.
- [7] Hu F., Hao Q., Bao K., A survey on software-defined network and openflow: From concept to implementation, *IEEE Communica-*

- tions Surveys and Tutorials, 2014, 16, 2181-2206.
- [8] Mijumbi R., Serrat J., Gorricho J.L., Bouten N., De T.F., Boutaba R., Network function virtualization: State-of-the-art and research challenges, *IEEE Communications Surveys and Tutorials*, 2016, 18, 236-262.
 - [9] Yan Q., Yu F.R., Gong Q., Li J., Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues and challenges, *IEEE Communications Surveys and Tutorials*, 2016, 18, 602-622.
 - [10] Diamantoulakis P.D., Kapinas V.M., Karagiannidis G.K., Big data analytics for dynamic energy management in smart grids, *Big Data Research*, 2015, 2, 94-101.
 - [11] Jarraya Y., Madi T., Debbabi M., A survey and a layered taxonomy of software-defined networking, *IEEE Communications Surveys and Tutorials*, 2015, 16, 1955-1980.
 - [12] Syahputra R., Robandi I., Ashari M., Performance Improvement of Radial Distribution Network with Distributed Generation Integration Using Extended Particle Swarm Optimization Algorithm, *International Review of Electrical Engineering*, 2015, 10, 293-304.
 - [13] Liu Q., Cai W., Shen J., Fu Z., Liu X., Linge N., A speculative approach to spatial-temporal efficiency with multi-objective optimization in a heterogeneous cloud environment, *Security and Communication Networks*, 2016, 9, 4002-4012.
 - [14] Yang C., Zhang X., Zhong C., Liu C., Pei J., Ramamohanarao K., et al., A spatiotemporal compression based approach for efficient big data processing on cloud, *J. Comp. Sys. Sci.*, 2014, 80, 1563-1583.
 - [15] Chen T., Matimikko M., Chen X., Zhou X., Ahokangas P., Software defined mobile networks: concept, survey and research directions, *IEEE Communications Magazine*, 2015, 53, 126-133.
 - [16] Yi X., Liu F., Liu J., Jin H., Building a network highway for big data: architecture and challenges, *IEEE Network*, 2014, 28, 5-13.