2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications

Redis++:A High Performance In-Memory Database Based on Segmented Memory Management and Two-Level Hash Index

Peng Zhang*, Lichao Xing*†, Ninggou Yang*†, Guolin Tan*†, Qingyun Liu*, Chuang Zhang*
*Institute of Information Engineering Chinese Academy of Sciences, Beijing, China
†University of Chinese Academy of Sciences, China

***Abstract*—Redis is an open source in-memory data structure store, used as a database, cache and message broker. However, there are two problems that will degrade its performance. One is the memory fragmentation problem, another is cache miss problem. For the purpose, this paper presents a high-performance in-memory database Redis++. In the memory management mechanism, Redis++ will allocate and deallocate a fixed-size memory segment from the system. The data in each memory segment are stored continuously, and the memory segment is reclaimed based on the profit evaluation model. Secondly, a two-level hash index structure is designed, the structure uses two-level index to complete only one cache mapping per query. In addition, instruction-level parallelism is implemented using the single instruction multiple data instruction set, which speeds up the query efficiency of the secondary index. The experiments prove the effect of Redis++ on memory utilization, response latency and throughput.**

***Index Terms*—in-memory database, memory segment, profit evaluation model, two-level hash index**

I. Introduction

With the rapid development of computer hardware technology and the continuous reduction of memory cost, it becomes feasible for database management system to put its working data set into memory completely. Compared with the conventional disk database, the in-memory database has faster read/write speed, higher throughput, and more efficient concurrent access, which meets the fast response requirements of many applications. Memcached [1], MICA [2] and Redis [3] are among several representative products. However, these products still have performance bottlenecks in memory fragmentation and cache miss.

Fragmentation can be thought of in terms of internal fragmentation and external fragmentation. Internal fragmentation is memory lost due to the allocator's policy of object alignment and padding. External fragmentation is memory lost because free memory is non-contiguous in a way that an allocation request cannot be satisfied even though the total amount of free memory would be sufficient for the request.

The latest version of Redis currently uses Jemalloc [4] as its memory management module by default. Jemalloc implements three main size class categories to reduce external fragmentation, *Small* allocation requests can be satisfied by allocating a single block, while larger ones require a possibly non-contiguous set of several blocks to access the required amount of memory. However, Jemalloc is not optimized for internal fragmentation, when Redis frequently allocates and deallocates memory to handle the insertion and deletion of variable-length object, both the memory occupied by Redis and the fragmentation rate keep increasing. The principle is as shown in Figure 1

Corresponding author: xinglichao@iie.ac.cn

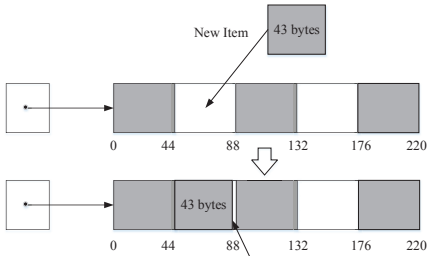

Fig. 1. The internal fragmentation of variable-length object allocator

In addition, the index structure is another key factor affecting the performance of the in-memory database. Because memory has better random read/write performance than disk, the index structure of the in-memory database is designed to improve cache utilization rather than increasing disk IO efficiency by converting random read/write to sequential read/write.

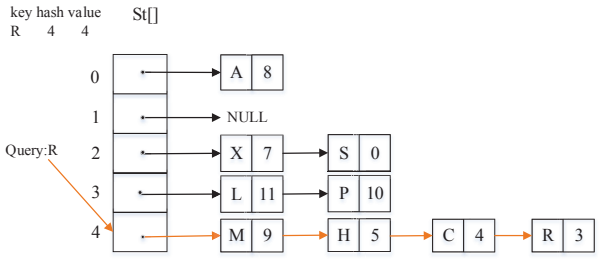

Fig. 2. The query processing by the separate chaining with linked lists

The cache is a memory with a small capacity but fast read/write speed. However, the cache capacity is usually limited, so how to increase the cache hit rate is the key to improve the overall system performance. Redis currently uses the separate chaining with linked lists to resolve hash collision.

978-1-7281-1141-4/18/$31.00 ©2018 IEEE
DOI 10.1109/BDCloud.2018.00125

IEEE computer society

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.

This method is easy to implement and debug. However, as the amount of data in hash table increases, each query needs to traverse the entire linked lists, and a large number of pointer random accesses seriously affect the cache hit rate as shown in Figure 2. This paper uses the performance analysis tool perf to track the cache miss during the running of Redis. The results are shown in Table I.

TABLE I
THE CACHE MISS PERCENTAGE DURING THE RUNNING OF REDIS

Function	Cache miss percentage (%)
siphash	22.5
dictFind	18.9
dictSdsHash	15.8
dictSdsKeyCompare	15.8
dictRehash	8.5
memcmp	6.0
other	12.5

From this experimental result, we can see that the cache miss triggered by the hash table related function call account for about 81.5% of the total, which shows that the current hash table design of Redis does not make full use of cache. How to effectively reduce cache miss will be the key point to improve the system performance.

Aimed at above problems, this paper designs and implements a high performance in-memory database Redis++. The contributions are as follows:

A segmented memory management mechanism is designed to reduce memory fragmentation. Under this mechanism, Redis++ will only allocate and deallocate the memory in units of 8MB fixed segment from the system, reducing the external memory fragmentation. At the same time, the use of the segment data structure and segment index allows data to be stored continuously in each segment, reducing internal memory fragmentation. In addition, an efficient memory reclaim strategy is proposed to improve memory utilization.

A two-level hash index structure is designed. The index structure divides 64-bit hash value into the first 48 bits and the last 16 bits. The first 48 bits are used as the first-level index to perform hash bucket mapping, and the last 16 bits are used as second-level index to support specific positioning in a hash bucket. The size of each hash bucket is designed to be a fixed 64 bytes, which is consistent with the cache line of current mainstream CPU. Compared to the traditional hash index, the new hash index only needs one cache mapping, which greatly reduces the probability of cache miss and improves cache utilization. In addition, the field size of the hash bucket header storing the secondary index is designed to be 128 bits, so that the SIMD (Single Instruction Multiple Data) instruction set provided by the Intel CPU can be used to execute comparison operations of the secondary index in parallel in a single instruction cycle.

The rest of this paper is organized as follows. Section II introduces the related works. Section III introduces the memory management mechanism of Redis++. Section IV introduces the two-level hash index of Redis++. Section 5 is devoted to the experimental results. Finally, the paper is concluded in Section 6.

II. RELATED WORK

A. Memory Management

Memcached uses a similar linux kernel's slab algorithm [5] for memory management. Memcached divides the allocated memory space into multiple 1MB slabs. Each slab is divided into chunks of a fixed size, and slabs with the same size of chunks belong to the same class. The slab algorithm uses the growth factor to determine the growth rate of the chunk size between adjacent classes. Memcached selects a class whose chunk size is the closest to the current data to store when inserting new data. This design can reduce the memory fragmentation and improve the memory utilization, but it can't deal with the data skew.

MemC3 [6] uses the cuckoo hash algorithm to optimize the index structure of Memcached, and draws on the idea of clock replacement algorithm to improve the LRU (Least Recently Used) elimination strategy to improve the efficiency of elimination. The basic idea of the cuckoo hashing algorithm is simply to provide two index positions for each key value pair through two hash functions. When a new data is inserted, it can select one of two index locations for storage. If both index positions are already occupied, it randomly selects one of the two locations for replacement. The replaced data is stored in other index position. If the position is also occupied, the replacement operation will be triggered again until an empty position is found or the number of replacement reaches the set threshold. If it is the latter, then the reconstruction of the hash table will be triggered. The cuckoo hash algorithm can make the load factor of the hash table, that is, the hash bucket usage rate as high as possible. In addition, this method can reduce memory space compared to using the open chain method to solve the hash conflict, because each value field in the open chain hash table will carry a pointer to point to the next node, but this pointer is not needed in the cuckoo hash index, and each value field of the hash table stores only the key value pair.

MongoDB [7] is a document-oriented storage. The data structure it supports is loose and is similar to json's bson format, which can store more complex data types. MongoDB uses a memory-mapping mechanism to manage data. It maps disk files to memory, and accesses the data on the disk through pointer access. This speeds up data access because it avoids system call for file operation and the overhead of copying between kernel space and user space.

RAMCloud [8] proposed a log structure memory management mechanism to perform memory allocation and recovery. Under this mechanism, RAMCloud only allocates and deallocates a fixed 8MB memory space from the system, called Segment, and indexes these segments through a hash table. These Segments only allow write-append operation. The update operation is transformed into appending a new data with a higher version number. The delete operation is transformed into writing tombstone data in append mode

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.

to indicate that the corresponding data have been deleted. Through this transformation, all data modification operations are transformed into continuous write operations to improve the cache hit rate, and the allocation and deallocate of the variable-length memory space are avoided, and the memory fragmentation rate is reduced. However, this design leaves a large amount of invalid data in the segment when the update and delete operations are frequent, resulting in a decrease in memory utilization.

B. Index Structure

The common index structures are tree index structure, bitmap index structure and hash index structure [9]. Bitmap indexing is the most memory-efficient indexing structure. This indexing structure allows each string to be equivalently replaced by a fixed-length binary expression to efficiently store key-value pairs. However, bitmap index usually has weaker support for update operation, so UpBit [10] is proposed to add an additional update vector for each bit vector. All update operations will be performed on the update vector, and bit vector will be updated based on the update vector when it is read in the next time. Although UpBit reduces the overhead of update operation to some extent, bitmap index is still somewhat inefficient when dealing with write-intensive requests compared to other index structures. The tree index structure is not only widely used in the traditional disk databases, but also has many optimization solutions in the in-memory databases. FAST [11], HAT-trie [12], and ART [13] design the nodes of the tree index structure in terms of the size of the cache mapping unit in order to improve the cache utilization. In addition, FAST and ART skillfully design data layout to use SIMD instruction set to implement instruction-level parallelism. Masstree [14] designs a multi-layer tree structure coupled with multiple B+ trees and uses data pre-fetching technique to improve the traversing efficiency. The advantage of the tree index structure is that it can support efficient range search operation and can save space and provide prefix-matching lookup when storing string data with a common prefix. However, since the search operation of the tree index structure needs to traverse the nodes of the tree, so its search efficiency is usually worse than that of hash index.

III. Segmented Memory Management Mechanism

In order to solve the problem of memory fragmentation when Redis handles frequent insertion and deletion of variable-length object, this paper designs a segmented memory management mechanism. In this mechanism, Redis++ only allocates and deallocates a fixed 8 MB memory space from the system, which avoids the external memory fragmentation caused by the frequent allocation and deallocate of variable-length memory space. The segment data structure used by this mechanism is shown in Figure 3.

We refer to each 8MB memory space as a Segment, and the smallest storage unit in each Segment is Object. Each Object contains 6 fields. InitialSize is an integer value, this field stores

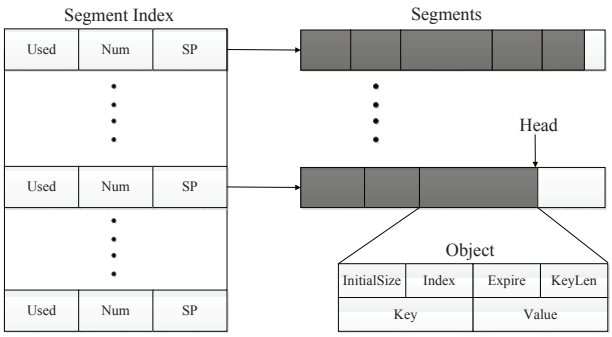

Fig. 3. The segment data structure

the initial size of the Object, which is the space occupied by the Object when it is inserted into the Segment. Because in each Segment, Object is stored continuously, so when the Object is updated, it needs to check whether the size of the updated Object exceeds its initial size to prevent from covering the data of the adjacent object. If the size of the updated Object exceeds the initial size, the original Object is deleted and the updated Object is inserted as new object at the place pointed by the Head pointer. The Head pointer always points to the end of the current Segment where it can be written. Index is a long integer value, which stores the index information of the object. The first 48 bits stores the index of the hash bucket where the index is located, and the last 16 bits stores the tag value of the index information, which will be detailed in the next section. When an object is deleted, Redis++ does not actually delete the object and deallocate the space it occupies. Instead, It sets the Object's Index field to 0 to indicate that the Object has been expired. Expire is a time type that stores the object's timeout elimination time. KeyLen is an integer value, which stores the Key string length of the Object, so that it can use a more efficient memcpy function instead of strcpy function to match the string. The Key and Value field stores the string of the Object's Key and Value, respectively.

The Segment Index is an array that stores the metadata of the Segment. Each array element contains three fields. The first field Used is an integer value that indicates the number of bytes that have been occupied in the Segment. The second field Num is an integer value that indicates the number of valid objects in the Segment. The third field SP is a pointer that points the address of the Segment. When the occupied memory space of the Segment is released, Redis++ does not delete the corresponding element in the Segment Index. Instead, Redis++ stores the segmentID of the Segment Index, that is, the element index in the Segment Index, into the idle queue. When the memory space pointed by the Head pointer is not enough to write new data, the SegmentID is preferentially taken from the idle queue, and the new segment is stored in the corresponding position of the Segment Index according to the SegmentID. When there is no free space in the Segment Index, the size of the array will be doubled.

In this memory management mechanism, all write opera-

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.

tions are in append mode, so the cache can be better used to improve write performance. However, some deletion and update operations will leave invalid data in the Segment. The accumulation of these invalid data will have a serious impact on memory utilization. Therefore, we need to design a memory reclaim strategy to solve this problem. The memory reclaim strategy makes use of the basic idea of RAMCloud, and regularly cleans the segments with invalid data. When selecting the segment to be reclaimed, the profit of the segment will be evaluated first, and the segment with the highest profit will be selected for reclaim. The profit evaluation model is as follows:

$$\frac{benefit}{cost} = \frac{(1-u) \times Lifetime}{u} = \frac{1-u}{u} \times \frac{\sum_{i=1}^{N}(Expire_i - Current)}{N} \tag{1}$$

In this formula, u is the usage rate of Segment, that is, the ratio of memory space occupied by valid data to the total memory space of the Segment, and Lifetime is the average survival time of Object in the Segment, that is, the mean value of the difference between Expire and the current time. N is the number of valid objects in the Segment. We use Lifetime to measure the stability of data in a segment. When cache elimination policy is set to LRU, the value of Expire is the time the object was last accessed plus the timeout time in the system configuration. When the cache elimination policy is set to TTL, the value of Expire is the timeout of Object. The larger the value of Lifetime, the longer the survival time of the data in the segment is, the better the stability is, because the Segment with smaller Lifetime value will generate invalid data more quickly.

Redis++ will choose the most profitable Segment to reclaim, and move its valid data in the Segment to the place pointed by the Head pointer, and deallocate the memory space occupied by the Segment.

IV. Two-Level Hash Index Design

This paper uses the xxHash function [15] as a hash function to build a hash table that can map an arbitrary-length string to a 64-bit hash value. XxHash is currently one of the fastest hashing functions for hashing. Under the SMHasher test dataset [16], xxHash can achieve a full score of 10 points in hash quality and 13.8 GB per second calculation speed.

In order to minimize the cache miss caused by traversing the linked list when performing search operation in the hash index, a two-level hash index structure is designed in this paper. The structure is shown in Figure 4.

The size of each hash bucket in the hash index is fixed at 64 bytes, which is exactly the same size as the cache mapping unit of X86 architecture. This ensures that each bucket is accessed only one cache mapping is required and there is no data access across the Cache line.

The size of a pointer in a 64-bit machine is 8 bytes. If a hash index has 2^64 hash buckets, the total space occupied by the hash index will exceed $8 * 2^64$ bytes. But a single general server is limited to 2TB of memory at present, so the number of buckets in the hash index cannot exceed 2^40. Since we

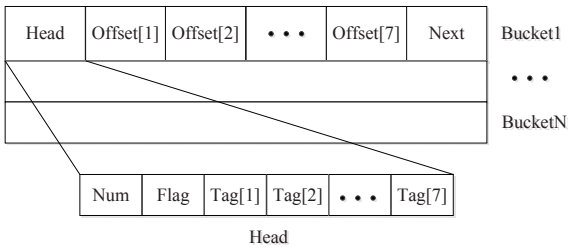

Fig. 4. The data structure of two-level hash index

can't directly use a 64-bit hash value as a subscript of the hash bucket in the hash index, we need to perform modulo operation so that the hash value falls within the number of hash buckets, so the hash collision rate is determined by the number of hash buckets instead of by the value range of the hash value. Therefore, Redis++ use the first 48 bits of the 64-bit hash value as the primary index to determine bucket subscript in the hash table which corresponds with the index information of the Object, and use the last 16-bit as a secondary index to perform the second search in the hash bucket to locate specific index information.

Each hash bucket is divided into three parts: head, Offset array, and expanded bucket ID. First, in the header, the Num field is an 8-bit bit vector. Its highest bit is always 0. The rest of the bits indicate the validity of data of the corresponding position of the tag array and the offset array in the header. The bit is set to 1 to indicate that the data stored in the corresponding position is valid, and the bit is set to 0 to indicate invalid. In the hash index, each index item is divided into two parts: Tag and Offset. Tag is the last 16 bits of the Object's Key hash, and Offset is a 48-bit vector that stores the position information of the Object in the Segment. For example, there stores two index entries in a hash bucket. The data of these two index entries are stored in Tag[1], Offset[1], Tag[3], and Offset[3], respectively. Then the binary expression of the Num field of the hash bucket is 00000101. So we can get the number of valid index in the current hash bucket by counting the number of digits set to 1 in the Num field. When the new index entry is inserted, the storage position of the new index entry is obtained by obtaining the lowest position of the zero in the Num field. When the binary expression of the Num field is 00000101, the new index entry is stored in Tag[2] and Offset[2]. The Flag field in the header is also an 8-bit bit vector. The highest bit is also 0. The remaining bits are used to indicate whether the index entries stored in the corresponding positions in the Tag array and Offset array are native. We will expand on the specific meaning of this field in the following sections of this section.

The Tag array in the header is used to store the Tag section of each index entry. Each Tag element has a size of 16 bits. When an Object is located through a hash bucket using the first 48 bits of the Key hash value as a primary index, and we use the next 16 bits as secondary index Tag to locate the specific Offset index information in the hash bucket. For example,

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.

when the last 16 bits of the Key hash value are equal to Tag[3] in the hash bucket, the Offset information of the Object corresponding to the Key will be stored in Offset[3]. In order to avoid the latency overhead of traversing the Tag array for comparison, Redis++ uses the SIMD instruction set to speed up the lookup process. The total size of the header field of each hash bucket is 128 bits, which is just in accordance with the SIMD operation specification, so that each 16-bit in the header field and the last 16 bits of Key hash value can be compared in parallel by using eight registers in one instruction cycle. By using the SIMD instruction to parallelize the comparison operation, the comparison operations are parallelized which improves the search efficiency.

The Offset array stores the specific information for each index entry. Each element in the Offset array is a 48-bit bit vector. The first 25 bits are used to store the SegmentID of the segment where the data of the Object is located, and the last 23 bits are used to store the offset of the Object's position in the Segment relative to the first address of the Segment. The reason for this design is that each Segment has a fixed size of 8MB, so we can just use the 23-bit bit vector to store the offset of each Object in the Segment relative to the first address.

Each hash bucket in the index can store up to 7 valid index entries. When more than 7 objects are mapped to the same hash bucket, we need to use the expanded bucket to store these index entries. The Next field in the hash bucket is a 48-bit bit vector used to store the expanded bucket ID. We use the square detection method to obtain the expanded bucket. For example, if the current bucket ID is b_1, we will select the hash bucket with the least number of positions in the Num field set to 1 as the expanded bucket of current bucket from the five bucket IDs $b_1 + 1^2, b_1 + 2^2, b_1 + 3^2, b_1 + 4^2, b_1 + 5^2$, and its bucket ID is stored in the Next field. The index entries that we place in the corresponding hash bucket after the first-level index mapping are called native index entries, which differentiate between these index entries and the index entries that are stored to the hash bucket through the expanded bucket mechanism. We use the Flag field in the header to indicate which index entries in the current hash bucket are native. For example, there are 3 index entries stored in a hash bucket, and the index entries stored in Tag[1], Offset[1], Tag[3], and Offset[3] are native and the index item stored in Tag[2] and Offset[2] is non-native, the binary expressions of its Num field and Flag field are respectively 00000111 and 00000101. When a hash bucket and its expanded bucket have been filled with 7 valid index entries and a new index entry needs to be stored in the hash bucket, a process similar to cuckoo hash will be triggered to make room for the new index entry. For example, if there are 7 valid index entries in the bucket, the value of the Flag field is 00110111. The new index entry will replace the native index entry in the bucket. For example, the index entries in the bucket that are stored in the Tag[1] and Offset[1], the replaced index item will be stored in its expansion bucket, which may trigger the replacement operation again. This replacement process will continue until there is free space to store the index entry that is replaced. When there is no native index entry in the expanded hash bucket or if the replacement operation has been executed more than the configured threshold value, a rehashing process is triggered to rebuild the hash table, and the hash table is expanded. The overall structure of the hash table is shown in Figure 5.

During the running of Redis++, we will maintain two hash table structures. The DictHt structure consists of four elements. The Table pointer stores the address of the hash index. The Size stores the number of buckets for the hash index. The value of SizeMask is Size-1, and we set the value of Size to the integer power of 2 so that the modulo operation at the first-level index searching can be transformed into the AND operation with SizeMask to improve the efficiency. Used stores hash buckets already used in the hash index. When it need to rehash, we will initialize the second hash table structure, and set the number of hash buckets of the hash index to the twice as much as the first hash index. Then the index entries in the first hash index are migrated to the second hash index. The Ht array in the Dict structure is used to store pointers which point two hash structures, and the RehashIdx field is used to store the running state of the rehash process. The initial value of RehashIdx is -1, indicating that no rehash process is currently performed. When the value is non-negative, it indicates that the hash index of the first hash index is currently migrated. During rehashing, the read operation of the hash index is performed on both indexes, and writing operation is performed only on the second hash index. After the rehashing is completed, Redis++ will reset RehashIdx to -1, and deallocate the memory occupied by the first hash table pointer, and then swap the two hash table pointers.

By making the size of the hash bucket consistent with the size of cache mapping unit, which are all 64 bytes, Redis++ improves the cache utilization and speeds up the search efficiency of the hash index.

V. Experiments

A. Experimental Settings

The following will test and analyze Redis++'s memory utilization, response latency and throughput. The experimental environment is shown in Table II.

TABLE II
THE EXPERIMENTAL ENVIRONMENT

Hardware	Settings
CPU	Intel Xeon E5-2698 v3 @2.30Ghz(dual-16cores)
DRAM	DDR4-2133 128GB
NIC	Intel® 82599ES 10 Gigabit Ethernet Controller

Facebook performs a statistical analysis of the dataset handled by Memcached cluster in a production environment, indicating that almost all requests in its online environment contain no more than 100 bytes of Key, and more than 95% of its requests Value is less than 1024 bytes in length [17]. Therefore, we designed three test datasets with different data size distributions, as shown in Table III.

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.

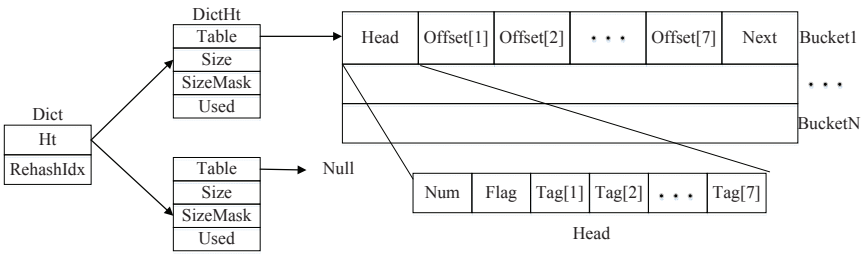

Fig. 5. The indexing process of two-level hash index

TABLE III
THE EXPERIMENTAL TEST DATASET

Scale	Key (Bytes)	95% Value (Bytes)	5% Value (Bytes)
Tiny	8	8∼16	1,024∼10,240
Small	16	16∼128	1,024∼10,240
Large	128	128∼1,024	1,024∼10,240

The Key and Value of the three test datasets are random strings, except that the 5% Value is between 1,024 and 10,240 bytes in length, and the remaining 95% of Value are less than 1,024 in length.

Since Redis++ is based on the Redis, the latest version of Redis (ver 4.0.6) will be selected as the reference system in the following experiments.

B. Memory Utilization

Memory utilization is a very important performance indicator for in-memory databases because all the data in the in-memory database is stored in memory, unlike the traditional disk database where only the index is stored in memory. We designed the four test datasets shown in Table IV to show the memory utilization of Redis++.

TABLE IV
THE MEMORY UTILIZATION IMPROVEMENT

Dataset	Before (Byte)	Delete	After (Byte)
W_1	100	0%	130
W_2	100	50%	130
W_3	100∼200	0%	200∼1,000
W_4	100∼200	50%	200∼1,000

In this experiment, we set the maximum memory usage limit of Redis and Redis++ to 10GB, and set the memory elimination policy to random elimination, so that when the maximum used memory of Redis and Redis++ reaches the upper limit and new data needs to be written, randomly eliminating data from existing data until there is enough space to store the newly written data. At the same time, we activate the memory fragmentation cleaning function supported by Redis after version 4.0, and set the thresholds to which there is more than 10% memory fragmentation.

The experiment is divided into three steps. In the first step, random strings with a total size of 20 GB are generated as the test dataset. The Key length of the test dataset is fixed at 100 bytes, and the Value length is shown in the Before column in Table 4, and then writing the Key-Value pairs of the test dataset into the system. The second step is to delete the currently stored data according to the proportion value shown in the Deletion column in Table 4. The third step is basically the same as the first step, except that the length of the generated test dataset is shown in the After column in Table IV. The experimental results are shown in Figure 6.

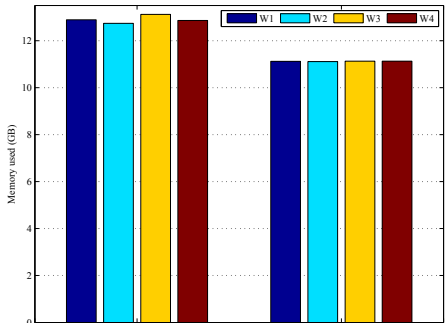

Fig. 6. The comparisons of memory utilization

Through this experiment, we can see that Redis++ has show better memory utilization than Redis under a variety of loads, the main reason is the memory allocation strategy and memory reclaim strategy used in Redis++ are more efficient than the Jemalloc used by Redis.

Redis++ achieves the highest memory utilization through segmented memory management mechanism. Whether the mechanism is efficient or not depends on the formula (Equation 1). The formula can be divided into two parts: utilization ($\frac{(1-u)}{u}$) and stability (Lifetime). How to balance the weight of these two parts is a key issue. If the utilization weight is given too high, the memory reclaim will appear to be too greedy, and it is easy to get the Segment with the local optimal value for reclaiming. Because the segment whose stability is poor will have its utilization reach the new lowest point faster, it is very likely to lose the opportunity to reach the global optimal solution if only the current utilization of the Segment is considered. If the stability weight is given too high, it will lead to the higher utilization of the Segment will be reclaimed, reducing the efficiency of memory management mechanism.

In order to achieve a good balance between utilization

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.

and stability, we make several variations of the formula and perform some tests. We define the memory reclaim cost in memory management as:

$$ReclaimCost = \sum_{i=1}^{N} u_i \tag{2}$$

In this formula, u_i is the memory usage of the segment that is reclaimed, and N is the total number of segments that are reclaimed during the memory management. The smaller the value is, the more the Segment selected for reclaiming is towards the global optimal solution.

In this experiment we used two different access modes. One is the random access mode. That is, the probability of occurrence of each Key is equal, and the test dataset is defined as the Uniform load. The other is the hot and cold access mode, where 20% of Keys in the test dataset occur 80% of the total, and the remaining 80% of Keys only occur 20% of the total. This test dataset is defined as the Zipf load [18]. We will perform memory reclaim that use three different formulas under these two access modes. We define the calculation formula of the Original mechanism as formula (1), the calculation formula of Greedy mechanism is to set the stability (Lifetime) in formula (1) as 1, and the calculation formula of Modified mechanism is to use $\sqrt{Lifetime}$ instead of the formula (Lifetime in formula 1), because in the formula (1) if simply multiplying or dividing a Lifetime by a constant, it cannot change the reclaim order for multiple Segments. The experimental results are shown in Figure 7.

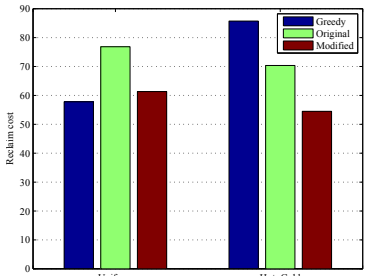

Fig. 7. The comparisons of memory reclaim strategies

From the experimental results, we can see that even though the Greedy mechanism performs the memory reclaim with the lowest cost under the Uniform load, it has the highest cost to perform memory reclaim under the Hot-Cold load because it only considers the Segment memory utilization and falls into local optimum trap. The Original mechanism does not handle Uniform load well because of its stability can easily interfere with the ReclaimCost calculation. Compared with the two mechanisms, the Modified mechanism performs best on the whole. This mechanism performs the memory reclaim with the minimum cost under Hot-Cold load, and the Hot-Cold access pattern is also more consistent with the real-world environment.

C. Response Delay

Response delay is an important indicator to measure the processing efficiency. Response delay refers to the length of time during which the client sent a request from the server to receive a response from the server. We will test write delay and read delay for Redis++ and Redis. In this experiment, we only run a single Redis and Redis++ process, then run only one client and the client sends 1,000,000 requests to two processes respectively. We perform many tests and calculate the average delay. The results are shown in Figure 8.

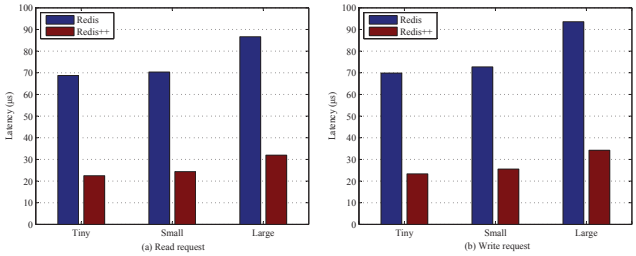

Fig. 8. The comparisons of response delay

The delay in processing read requests is shown in Figure 8(a). We can see that Redis++ has a delay of 22.47 microseconds when dealing with Tiny data sets, while Redis has a delay of 68.74 microseconds, and Redis++ is only a third of Redis. With the increase of the data size, the response delay in both systems will increase slightly.

The delay in processing write requests is shown in Figure 8(b). We can see that when dealing with Tiny data sets, the write request delay of Redis++ and Redis is not significantly different from the read request delay. With the increase of the data size, the processing efficiency of the write request is significantly lower than that of the read request, but overall, the processing efficiency of Redis++ can be kept stable at about three times that of Redis.

D. Throughput

Throughput is another important indicator to measure the database system. In this paper, the throughput is defined as the total amount of requests processed by the system in a unit of time. We use YCSB (Yahoo! Cloud Serving Benchmark) [19] for testing. YCSB is a Yahoo! open source tool for basic testing of NoSQL databases. In order to simulate the actual application scenario as much as possible, we will define two test loads and two access modes. A write-intensive load contains 50% of write requests and 50% of query requests, and read-intensive load contains 5% of write requests and 95% of query requests. The Uniform access mode means that all requested Keys are randomly generated with equal probability. In the Zipfian access mode, the distribution of all requested Keys is in accordance with the Ziff distribution, that is, about 20% of the Keys will appear in about 80% of the requests. This is in line with the actual application scenario.

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.

In this experiment, we will run a single Redis++ and Redis server process respectively, and send 50 client requests to the two server processes. The experimental results are shown in Figure 9.

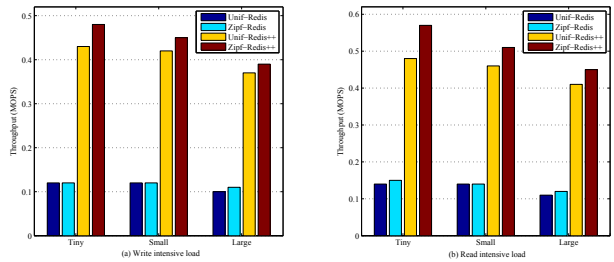

Fig. 9. The comparisons of throughput

The experimental results of the write intensive load are shown in Figure 9(a). We can see that Redis++ can achieve throughput of 0.43 and 0.48 million requests per second in the Uniform and Zipfian access modes when processing Tiny datasets, while Redis can only achieve throughput of 0.12 million requests per second, Redis++'s throughput performance can reach about 4 times that of Redis. With the increase of data size, although Redis++'s throughput will be slightly dropped, it can still be maintained at a relatively high level.

The experimental results for read intensive load are shown in Figure 9(b). When dealing with Tiny datasets, Redis++ can reach throughput of 0.48 and 0.57 million requests per second respectively in Uniform and Zipf access modes, while Redis is 0.14 and 0.15 million requests per second, respectively, and Redis++s throughput performance is still about 4 times that of Redis. The performance in the Zipf access mode is significantly higher than that in the Uniform access mode, mainly because the cache hit rate is higher in the Zipf access mode, so the processing efficiency is higher.

From this experiment, we can see that under a variety of data sizes, read/write intensive loads, and access modes, Redis++ can consistently achieve about 4 times the throughput performance of Redis.

VI. Conclusion

In this paper, we select Redis that ranks the top one in market share as a benchmark, and analyzes in depth its memory management bottlenecks. Next, we design and implement two optimization solutions, and develop a high performance in-memory database Redis++ based on Redis. The experiments prove the performance improvement of Redis++ over Redis on memory utilization, processing latency and throughput. However, the optimization solutions are mainly related to the String structure, there still exists memory fragmentation and cache miss when using List, Set, Zset, Hash, and other data structures. In the future, we will also optimize the design of these data structures.

Acknowledgment

The research work is supported by National Key R&D Program 2016 (Grant No.2016YFB0801300), National Natural Science Foundation of China (No. 61602474, No. 61602467, No. 61702552).

References

[1] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.
[2] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage." USENIX, 2014.
[3] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 2011, pp. 363–366.
[4] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.
[5] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.
[6] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing." in *NSDI*, vol. 13, 2013, pp. 371–384.
[7] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.
[8] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 29–41.
[9] P. L. Bajaj, "A survey on query performance optimization by index recommendation," *International Journal of Computer Applications*, vol. 113, no. 19, 2015.
[10] M. Athanassoulis, Z. Yan, and S. Idreos, "Upbit: Scalable in-memory updatable bitmap indexing," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1319–1332.
[11] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 339–350.
[12] N. Askitis and R. Sinha, "Hat-trie: a cache-conscious trie-based data structure for strings," in *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*. Australian Computer Society, Inc., 2007, pp. 97–105.
[13] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 38–49.
[14] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 183–196.
[15] Y. Collet, "xxhash-extremeley fast hash algorithm," 2016.
[16] A. Appleby, "Smhasher," Accessed 2017-04-12. URL: https://github.com/aappleby/smhasher, Tech. Rep., 2016.
[17] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.
[18] D. M. Powers, "Applications and explanations of zipf's law," in *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 1998, pp. 151–160.
[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

Authorized licensed use limited to: Indian Institute of Technology - Jodhpur. Downloaded on November 13,2024 at 13:40:26 UTC from IEEE Xplore. Restrictions apply.