A High Performance Memory Key-Value Database Based on Redis

Qian Liu*, Haolin Yuan
First Research Institute of the Ministry of Public Security of RPC, Beijing, China.

* Corresponding author. Email: liuqian1104@126.com
Manuscript submitted January 10, 2019; accepted February 20, 2019.
doi: 10.17706/jcp.14.3.170-183

Abstract: This paper proposes a high-performance memory key-value database Redis++. In the memory management mechanism, Redis++ can apply and release a fixed-size memory segment from the system. The data in each memory segment is stored consecutively, and the memory is reclaimed based on the profit evaluation value. Secondly, a cache-friendly hash index structure is designed and the structure uses two-level index which can solve the hash collision to complete per search which needs cache mapping only once if possible. In addition, using the SIMD instruction set to realize instruction-level parallelism, which speeds up the search efficiency of the secondary index. The experiments prove the effect of Redis++ on memory utilization, system latency, and throughput.

Key words: In-memory database, memory segment, profit evaluation model, two-level hash index.

1. 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 data-storage speed , higher throughput, and stronger concurrent access capability, which meets the fast response requirements of many applications. Memcached [1], MICA [2] and Redis [3] are among several representative products, but these products still have performance bottlenecks in memory fragmentation and cache miss.

In Linux system, all memory allocation must start at an address that is evenly divisible by 4, 8 or 16 (depending on the processor architecture) or is determined by the paging mechanism of the MMU (memory management unit). The memory manager can only obtain fixed-size memory pages when applying for memory from the system, so the extra space created by the difference between the size of the memory allocated and the actual size required by the applicant is called internal memory fragmentation. In addition, the frequent allocation and release of variable-length memory space can result in a large number of non-consecutive small memory segments interspersed among the allocated memory segments, produced the external memory fragmentation.

The latest version of Redis currently uses Jemalloc [4] as its memory management module by default. Jemalloc makes the size of allocated and released memory segments conform to certain rules and reduces the generation of external fragmentation by classifying the memory allocation request. However, this design is not optimized for the problem of internal fragmentation, and the common scenario of Redis is

caching. In order to process the writing and expiration of variable-length data, Redis needs to apply and release memory frequently to the system, which leads to increasing memory occupied by Redis and increasing fragmentation rate.

In order to test the actual memory usage performance of the memory allocator supported by Redis, related experiments were designed for verification. The test data set is designed as shown in Table 1.

Table 1. The Memory Utilization Dataset

Dataset	Before (Byte)	Delete	After (Byte)
W_1	100	N/A	N/A
W_2	100	0%	130
W_3	100	50%	130
W_4	100	90%	130
W_5	100-200	N/A	N/A
W_6	100-200	0%	200-1000
W_7	100-200	50%	200-1000
W_8	100-200	90%	200-1000

In this experiment, Redis's maximum memory usage parameter is set to 10 GB, and the elimination policy is set to allkeys-random. That is, when the data stored by Redis exceeds 10 GB, existing data will be randomly eliminated to ensure that valid data does not occupy more than 10 GB of memory. Each test is divided into three steps. The first step is to generate a test data set that conforms to the distribution shown in the *Before* column in Table 1. Such as the test data set for W_1 is composed of key- value pairs with a fixed size of 100 bytes. The W_5 test data set consists of key-value pairs with a value range of 100 to 200 bytes. The total size of each test data set is 10 GB, and then the data sets are written to Redis. The second step is to delete the data in Redis as shown in the *Delete* column. The third step is basically the same as the first step, but the value size distribution of the data set is modified as shown in the *After* column.

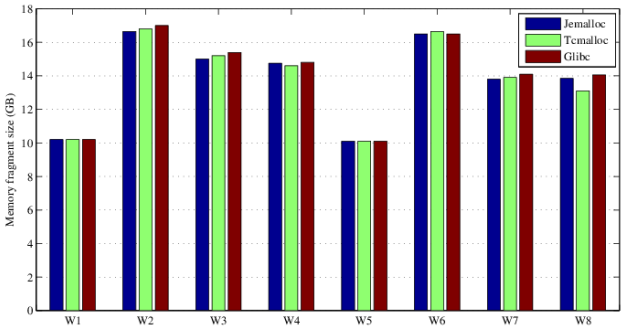

Fig. 1. The memory fragmentation comparisons of three memory allocators.

The test results are shown in Fig. 1. Jemalloc, Tcmalloc [5], and Glibc all have memory fragmentation problems. Jemalloc performed best, followed by Tcmalloc, and Glibc had more serious fragmentation problem. Jemalloc is only slightly inferior to Tcmalloc under W_4 and W_8, so Redis chooses Jemalloc as its memory allocator by default. The memory fragmentation rate under W_2 and W_6 is significantly higher than other tests. It is because other tests firstly perform some delete operations while no deletions are performed in W_2 and W_6. From this experiment, we can see that there are some memory fragmentation problems in current allocators when dealing with variable-length data. Therefore, it is necessary to design a new memory management mechanism to improve the memory utilization of Redis.

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 reads and writes to sequential reads and writes.

The Cache is located between CPU and DRAM. It is a memory with a small capacity but fast read/write speed. The Cache capacity is limited, so how to increase the Cache utilization is the key to improving the overall system performance. Redis currently uses the method of chain address to resolve hash collision. This method is simple to implement and easy to debug. However, with the increase of data in hash table, each query needs to traverse the entire linked list so that a large number of pointer random access seriously affect the cache hit rate. This paper uses the perf software to track the Cache-miss events during the running of Redis. The results are shown in Table 2.

Table 2. The Cache Miss Percentage 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 experiment, we can see that the Cache-miss event triggered by the hash table related function call accounted for about 77% of the total. This indicates that the current hash table design of Redis does not make full use of Cache, and how to reduce cache-miss effectively will become the key point to improve the overall performance of the system. To that end, the primary contributions of this paper are as follows:

a) A segmented memory management mechanism was proposed. Under this mechanism, Redis++ will only apply to system for memory space and release it in units of 8 MB fixed segments. It can avoid the problems of external memory fragmentation caused by the system's frequent allocating and recycling memory space of different sizes. At the same time, the use of the smart data structures and read-write strategies allows data to be stored consecutively in each memory segment, which avoids internal memory fragmentation. In addition, in order to improve memory utilization, an efficient memory reclaim strategy was proposed.

b) A cache-friendly hash table is designed as the index structure of Redis++ to solve the problem of low cache utilization. The index structure divides the 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, the latter 16 bits are used as secondary indexes in the hash bucket to support the specific positioning of index items in the bucket. Compared to the traditional hash index, the proposed hash index only needs one cache mapping. It greatly reduces the probability of cache misses and improves cache utilization.

The rest of the paper is organized as follows: Section 2 introduces the related works. Section 3 describes the memory management mechanism of Redis++. Section 4 presents the two-level hash index of Redis++. Section 5 shows our experimental results. Finally, the paper is concluded in Section 6.

2. Related Work

2.1. Memory Management

Memcached [6] uses a similar linux kernel's slab algorithm for memory management. It divides the applied memory space into multiple 1 MB slab. Each slab block 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 class. Memcached selects a class whose

chunk size is the closest to the current key-value data to store when inserting new data. This design can reduce the memory fragmentation and improve the memory utilization, but it cannot deal with the data skew.

Memc3 [7] uses the cuckoo hash algorithm to optimize the index structure of Memcached, and uses the idea of clock replacement algorithm to improve LRU (least recently used) strategy. The basic idea of the cuckoo hashing algorithm is simply to provide two index positions for each key value pair data 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 its 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 replacements 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 the waste of pointer 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. However, this pointer is not needed in the cuckoo hash index, and each value field of the hash table stores only the key value data.

MongoDB [8] is a documented NoSQL database. It 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 calls of file operations and the overhead of data copying between kernel space and user space.

RAMCloud [9] utilizes a log structure memory management mechanism. It only applies and releases a fixed 8 MB memory space, called Segment, and indexes these segments through a hash table. These Segments only allow supplementary write operations. The update operation is transformed into writing new data additionally with a higher version number. The delete operation is transformed into writing additional tombstone data to indicate that the corresponding data has been cleared. Under this limitation, all modification operations are converted to consecutive write operations which can improve the Cache hit rate. And the application and release of the variable-length memory space are avoided so that the memory fragmentation rate is reduced. However, this approach leaves a large amount of invalid data in the segment when the update and delete operations are frequent.

2.2. Index Structure

The common index structures are tree index structure, bitmap index structure and hash index structure [10]. 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 through hash map. However, bitmap index usually has weaker support for update operations. Athanassoulis *et al* [11] proposed the Upbit which adds an additional update vector for each bit vector. All update operations will be performed on the update vector, and bit vector will be updated when it is read in the next time. Although Upbit alleviates the overhead of update operations to some extent, bitmap indexes are still somewhat inefficient when dealing with write-intensive request load compared to other index structures. The tree index structure is not only widely used in the traditional hard disk database, but also has many optimization solutions in the in-memory database. FAST [12], HAT-trie [13] and ART [14] designed the nodes of the tree 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 (single instruction multiple data) instruction set to implement instruction-level concurrency efficiency. Mao *et al* [15] designed a multi-layer tree structure with multiple B+ trees coupled and used data pre-fetching techniques to improve tree traversal efficiency. The advantage of the tree index structure is that it can support efficient range

search operations and can save space and provide prefix-matching lookups when storing string data with a common prefix. However, the search operation of the tree index structure needs node traversal of the tree, so the single point search efficiency is usually lower than the hash index.

3. Segmented Memory Management Mechanism

In order to solve the problem of memory fragmentation problems when Redis handles frequent insertion and deletion of variable-length data, we propose a segmented memory management mechanism. In this mechanism, Redis++ only applies and releases a fixed 8 MB memory space from the system. This avoids the external memory fragmentation problems caused by the frequent application and release of variable-length memory space. The data structure of proposed mechanism is shown in Fig. 2.

We refer to each 8 MB 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 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 consecutively, so in order to provide in-place update functionality, it needs to check whether the size of the updated *Object* exceeds its initial size when the content stored on *Object* is updated, so as to avoid overwriting part data of the next *Object* after the update. 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 store the subscript of the hash bucket where the index information is located, and the last 16 bits store 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* 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.

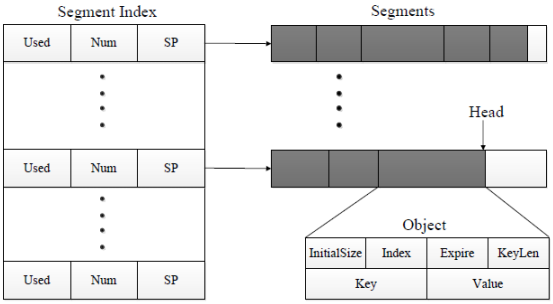

Fig. 2. The segment data structure.

The *Segment Index* is an array that stores the meta information of the *Segment*. Each array element contains three fields. The first field *Used* is an integer value that stores the number of bytes that have been occupied in the *Segment*. The second field *Num* is an integer value that stores the number of valid objects in the *Segment*. The third field *SP* is a pointer that strores the address of the *Segment*. When the occupied memory space of the *Segment* is released, instead of deleting the corresponding element in the *Segment Index*, we put the *SegmentID*, the array index in the *Segment Index* into the free 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 will be 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 operations are in append mode, so the cache can be better used to improve write performance. However, some delete 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 collection strategy to solve this problem. The memory collection strategy makes use of the fundamental idea of RAMCloud [9], and the *Segment* with invalid data is cleaned and reclaimed on a regular basis. When selecting the segment to be reclaimed, the profit of the reclaiming *Segment* will be evaluated first, and the segment with the highest profit will be selected for reclaim. The calculation formula of reclaiming profit is as follows:

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

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 average of the absolute value of the difference between *Expire* and the *Current* time. N is the number of valid *Object* in the *Segment*. We use *Lifetime* to measure the stability of data in a *Segment*. When cache elimination strategy is set to LRU, the value of *Expire* is the last time of the object was 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* means that the longer the data lives in the *Segment* and the better the stability, because the *Segment* with smaller *Lifetime* value will generate invalid data more quickly.

Redis++ will select the *Segment* with the highest profit to clean and 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*.

4. Two-Level Hash Index Design

We use the xxHash function in [16] as a hash function to build a hash table, which can map an arbitrary-length string to a 64-bit hash value. Xxhash is currently one of the fastest hash functions for hashing. Under the SMHasher test set [17], xxHash can achieve a full score of 10 points in hash quality and 13.8 GB per second calculation speed. At present, the widely used hash-function MurmurHash [18] is only 2.7 GB per second, although the score of hash quality is also 10 points.

In order to minimize the cache invalidation caused by traversing the linked list when performing search operation in the hash index, we design a new hash index structure shown in Fig. 3.

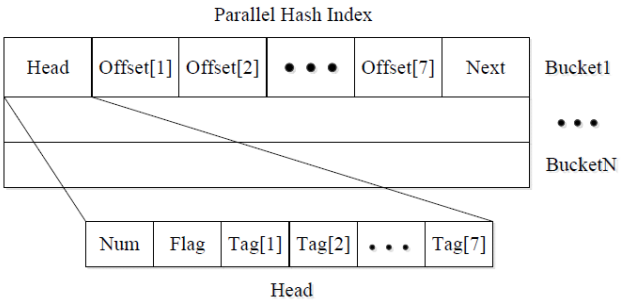

Fig. 3. The data structure of two-level hash index.

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 2 TB of memory at present, so the number of buckets in the hash index cannot exceed 2^{40}. Since we cannot 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++ uses the first 48 bits of the 64-bit hash value as the primary index to determine the bucket index of the object 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 *Head*, 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 value, 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 *Head* 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 *Head* 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 in a hash bucket through 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, 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, we uses the SIMD instruction set to speed up the lookup process. The total size of the *Head* 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 *Head* 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, 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 8 MB, 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.

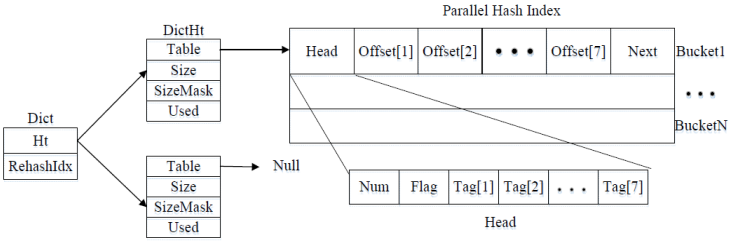

Fig. 4. The indexing process of two-level hash index.

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 buckets whose *ID* in 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 *Head* 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 Fig. 4.

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 designing the size of each hash bucket skillfully, it is exactly the same size as the cache mapping unit (64 bytes), to improve the cache utilization and speed up the hash index search efficiency.

5. Experiments

5.1. Experimental Settings

We analyze Redis++'s memory utilization, response latency and throughput. The experimental environment is shown in Table 3.

Table 3. The Experimental Environment

Hardware	Settings
CPU	Intel Xeon E5-2698 v3 2.30 Ghz (dual-16cores)
DRAM	DDR4-2133 128 GB
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 [19]. Therefore, we designed three test datasets with different data size distributions, as shown in Table 4. 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. We choose Redis 4.0.6 as baseline system.

Table 4. The Experimental Test Dataset

Scale	Key (Bytes)	95% Value (Bytes)	5% Value (Bytes)
Tiny	8	8-16	1024-10240
Small	16	16-128	1024-10240
Large	128	128-1024	1024-10240

5.2. Memory Utilization

We designed the four test datasets shown in Table 5 to show the memory utilization of Redis++.

Table 5. 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%	200-1000
W_4	100-200	50%	200-1000

We set the maximum memory usage limit of Redis and Redis++ to 10 GB, and set the memory elimination strategy to random elimination. When the memory of Redis and Redis++ reaches the upper limit and new data needs to be written, it will randomly evict data from existing data until there is enough space to store newly written data. At the same time, we activated the memory fragmentation cleaning function supported by Redis after version 4.0, and set the threshold for Redis and Redis++ on memory cleanup to start memory clean up when there is more than 10% of fragmentation in memory.

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 data set. All length of *Key* is fixed at 100 bytes, and the length of *Value* is shown in the *Before* column in Table 5. Then writing the *Key*-*Value* pairs of the test data set into the system in turn. The

second step is to delete the currently stored data according to the proportion value shown in the *Delete* column in Table 5. The third step is basically the same as the first step, except that the length of the generated test data set is shown in the *After* column in Table 5.

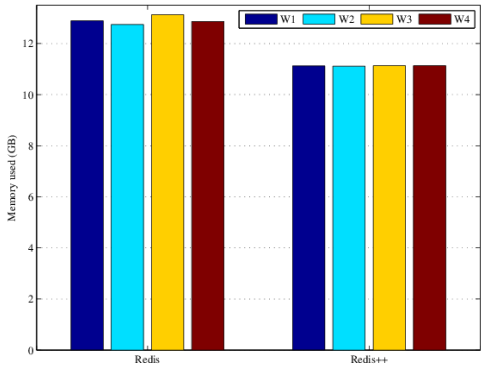

Fig. 5. The comparisons of memory utilization.

The experimental results are shown in Fig. 5. We can see that Redis++ achieves high memory utilization through memory reclaim mechanism. Whether the memory recovery mechanism is efficient depends on the appropriateness of the design of the return formula in (1). The formula can be divided into two parts: utilization ($(1-u)/u$) and stability (*Lifetime*). How to balance the weight of these two parts is important. If the weight of utilization is given too high, the memory reclaim will appear to be too greedy. And it is more likely to get the *Segment* with the local optimal reclaiming benefit for cleaning and 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 globally optimal solution if only considering the current utilization of the *Segment*. If the weight of stability is given too high, it will lead to the higher-utilization *Segment* will also be cleaned up and reclaimed, resulting in reducing the efficiency of reclaiming.

In order to achieve a good balance between utilization 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 (2), u_i is the memory usage of the *Segment* that is reclaimed, and N is the total number of the *Segment* that are reclaimed during the memory management. The smaller value of *ReclaimCost* indicates that the *Segment* selected for each cleaning and recycling more tend to be the globally optimal solution.

In this experiment we used two different modes of test loads. 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-Cold* load, that is, the occurrence number of 20% of *Keys* in the test dataset occurs 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 [20]. We will perform memory reclaim that use three different reclaiming revenue formulations under these two access modes. We define the *Original* mechanism as (1), the *Greedy* mechanism is to set the stability of *Lifetime* in (1) to 1, and the *Modified* mechanism is to use $\sqrt{Lifetime}$. The experimental results are shown in Fig. 6.

In Fig. 6, 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. Because *Original*'s stability in the calculation formula of recovery profit can easy interfere with the result of the final

calculation, resulting in the *Original* mechanism cannot handle the *Uniform* load well. 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.

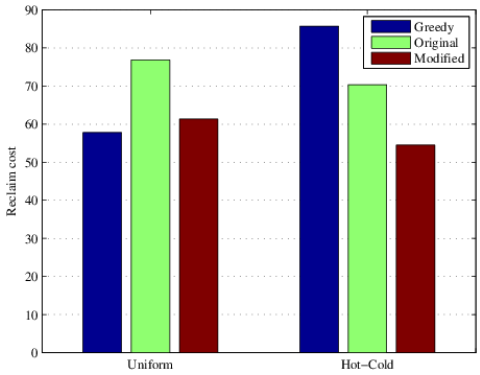

Fig. 6. The comparisons of memory reclaim strategies.

5.3. Response Delay

Request processing delay is an important indicator of system processing efficiency. In this paper, latency refers to the amount of time the client spends between sending a request to the server and receiving a reply 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.

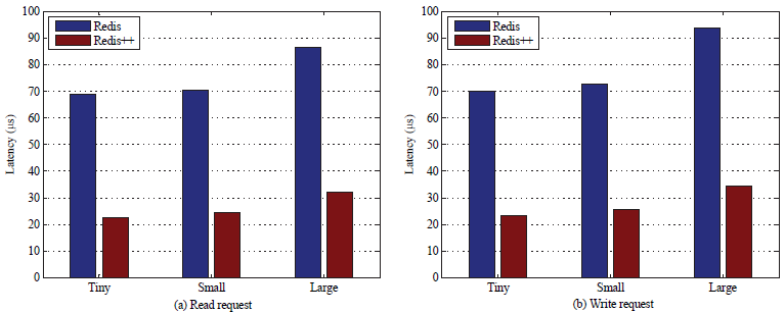

Fig. 7. The comparisons of response delay.

The delay of read requests is shown in Fig. 7 (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. The delay of write requests is shown in Fig. 7 (b). We can see that when dealing with *Tiny* data sets, the 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 Redis++ request processing efficiency can be maintained steadily about three times times over Redis.

5.4. Throughput

System throughput is another important indicator to measure the efficiency of request processing of a database system. In this paper, 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) [21] for testing. YCSB is an

open-source tool used by Yahoo to perform basic tests on 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 *Unif* access mode means that the *Key* of all requested are randomly generated with equal probability. In the *Zipf* 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.

In this experiment, we will run a single Redis++ and Redis server process, with 50 client processes sending requests to the server program.

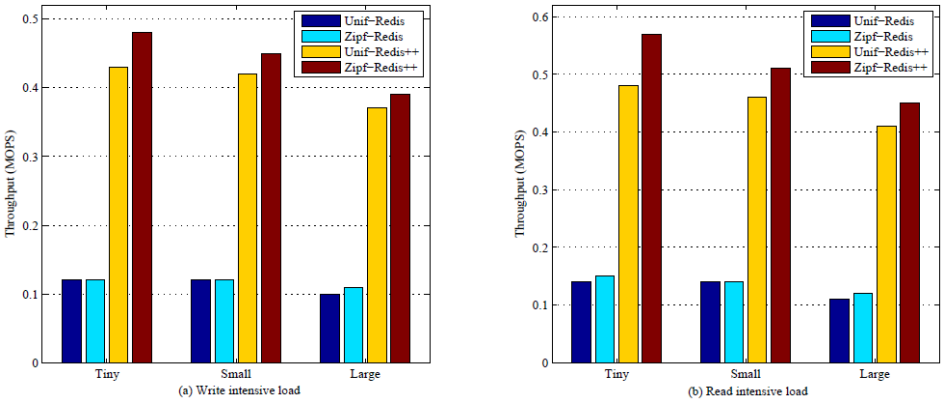

Fig. 8. The comparisons of throughput.

The experimental results of the write intensive load are shown in Fig. 8 (a). We can see that Redis++ can achieve throughput of 0.43 and 0.48 million requests per second in the *Unif* and *Zipf* access modes when processing *Tiny* datasets. And 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 steadily at a relatively high level.

The experimental results of read intensive load are shown in Fig. 8 (b). When dealing with *Tiny* datasets, Redis++ can reach throughput of 0.48 and 0.57 million requests per second respectively in *Unif* and *Zipf* access modes respectively. And Redis is 0.14 and 0.15 million requests per second, respectively. 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 *Unif* 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, the type of read/write intensive loads, and access modes, Redis++ can steadily achieve about 4 times the throughput performance of Redis.

6. Conclusion

In this paper, we select Redis that ranks the top one in market share as a benchmark, and analyzes its memory management bottlenecks. Then, 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 misses when using list, set, zset, hash, and other data structures. In the future, we will optimize the design of these data structures.

References

[1] Fitzpatrick, B. (2004). Distributed caching with memcached. *Linux Journal*, *2004(124)*, 5.

[2] Lim, H., Han, D., Andersen, D. G., & Kaminsky, M. (2014). Mica: A holistic approach to fast in-memory key-value storage. *Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation* (pp. 429-444). Seattle, WA, USA.

[3] Han, J., Haihong, E., Le, G., & Du, J. (2011). Survey on nosql database. *Proceedings of 6th International Conference on in Pervasive Computing and Applications* (pp. 363-366). Port Elizabeth, South Africa.

[4] Evans, J. (2006). A scalable concurrent malloc (3) implementation for free BSD. *Proceedings of the BSDCan Conference* (pp. 1-14). Ottawa, Canada.

[5] Lee, S., Johnson, T., & Raman, E. (2014). Feedback directed optimization of tcmalloc. *Proceedings of ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (p. 3). Edinburgh, English.

[6] Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). *Operating System Concepts Essentials*. John Wiley & Sons, Inc.

[7] Fan, B., Andersen, D. G., & Kaminsky, M. (2013). Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. *Proceedings of 10th USENIX Symposium on Networked Systems Design and Implementation* (pp. 371-384). Boston, USA.

[8] Banker, K. (2011). *MongoDB in Action*. Manning Publications Co.

[9] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., & Rosenblum, M. (2011). Fast crash recovery in ramcloud. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 29-41). Cascais, Portugal.

[10] Bajaj, P. L. (2015). A survey on query performance optimization by index recommendation. *International Journal of Computer Applications*, *113(19).*

[11] Athanassoulis, M., Yan, Z., & Idreos, S. (2016). Upbit: Scalable in-memory updatable bitmap indexing. *Proceedings of the 2016 International Conference on Management of Data* (pp. 1319-1332). San Francisco, USA.

[12] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., & Kaldewey, T. (2010). Fast: Fast architecture sensitive tree search on modern cpus and gpus. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (pp. 339-350). Indianapolis, Indiana.

[13] Askitis, N., & Sinha, R. (2007). Hat-trie: A cache-conscious trie-based data structure for strings. *Proceedings of the Thirtieth Australasian Conference on Computer Science* (pp. 97-105). Ballarat, Victoria, Australia.

[14] Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering* (pp. 38-49). Brisbane, Australia.

[15] Mao, Y., Kohler, E., & Morris, R. T. (2012). Cache craftiness for fast multicore key-value storage. *Proceedings of the 7th ACM European Conference on Computer Systems* (pp. 183-196). Bern, Switzerland.

[16] Collet, Y. (2016). *Xxhash-Extremeley Fast Hash Algorithm*.

[17] Appleby, A. (2016). *Smhasher.* Retrieved April 12, 2017, from https://github.com/aappleby/smhasher

[18] Appleby, A. (2008). *MurmurHash*. Retrieved from https://en.wikipedia.org/wiki/MurmurHash

[19] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., & Paleczny, M. (2012). Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review*, *40(1),* 53-64.

[20] Powers, D. M. (1998). Applications and explanations of zipf's law. *Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning* (pp. 151-160). Montréal, Québec, Canada.

[21] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, & R., Sears, R. (2010). Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing* (pp. 143-154). Indianapolis, Indiana, USA.

Qian Liu was born in Shandong province of China in 1984. She received the Ph.D degree in computer science and technology from Institute of Computing Technology Chinese Academy of Sciences in 2015. Her research interests include big data analysis, machine learning and named entity recognition.