Bucket-Based Expiration Algorithm: Improving Eviction Efficiency for In-Memory Key-Value Database

Guochao Xie
guochaoxie@link.cuhk.edu.cn
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China

Yeh-Ching Chung
ychung@cuhk.edu.cn
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China

ABSTRACT

Evicting expired keys for an in-memory key-value database is essential to save its memory resources and control its memory usage not exceeding its memory limit. Existing randomized expiration algorithms randomly sample keys from the key space associated with expiration, evict all expired ones and proceed to the next iteration if the proportion of expired keys exceeds a pre-defined threshold. The randomized algorithm has been adopted by Redis currently.

In this paper, we present a novel approach, a hybrid algorithm combining a deterministic one using buckets and a randomized one inherited from Redis expiration algorithm, to improve the efficiency of eviction of expired keys. For the main part, we adopt a deterministic algorithm to discretize the expiration timestamps into buckets and evict keys bucket by bucket; if time permitted, we also run the Redis expiration algorithm after finishing the deterministic part. Furthermore, our experiment using Redis randomized algorithm as the baseline shows that our algorithm is more effective in reducing memory usage with an acceptable impact on the overall throughputs.

CCS CONCEPTS

• **Information systems** → **Data management systems**; **Information storage systems**.

KEYWORDS

Eviction algorithm, Expiration, Redis, Key-value Database, In-Memory Storage

ACM Reference Format:
Guochao Xie and Yeh-Ching Chung. 2020. Bucket-Based Expiration Algorithm: Improving Eviction Efficiency for In-Memory Key-Value Database. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3422575.3422797

1 INTRODUCTION

With the rapid growth of big data, in-memory key-value databases are becoming popular to serve as a temporary cache between web applications and persistent databases. A key-value database is a type of NoSQL database where each object stored is accessed by a single key and no relationship between objects [11]. Because of its excellent scalability, a key-value database can provide very fast random access via key and ease of data partitioning [8], which meets requirements of high throughputs for modern web applications. Among all key-value databases, on-disk and in-memory are two main types. For the on-disk key-value database, MongoDB is a popular one [1, 3]; for in-memory ones, Redis [5], [14] and Memcached [10] are two representatives utilizing RAM for data storage. Comparing these three popular NoSQL databases using YCSB benchmark tool [7], Redis has the best efficiency for loading and executing workloads [21], because its in-memory design takes advantage of RAM for faster operations. Furthermore, Redis supports clustering and current researches show great interest in using Redis as the implementation base of algorithms for in-memory NoSQL database, like a more scalable and reliable Redis [6], Distributed Dynamic Cuckoo Filter System [15], and abnormal payment transaction detection scheme [13]. Therefore, in our current project, we use Redis as the representative of the in-memory key-value database for implementation and analysis.

For a temporary cache, keys are associated with expiration time for two main purposes: limiting memory usage and maintaining data security. Because an in-memory key-value database utilizes memory for storage to achieve high throughputs, its limited memory resource becomes precious and eviction is essential to keep the total memory usage within a limit to avoid memory swaps that could significantly affect the performance. With an expiration time attached for each key, it follows the eviction policy that the earlier expiration time indicates the higher eviction priority and that those expired keys are with the highest eviction priority. By evicting expired keys efficiently, we can save a greater amount of memory resources and store more keys to enlarge the total capacity of our database. As for the other purpose of maintaining data security, setting an expiration time is beneficial for use cases like user session storage [14], where a user's connection is expired after a long time's inactivity to protect users' information security and lower the probability of security issues for using a public computer [22]. Setting expiration information and let the database manage expiration automatically is therefore an easy and popular approach for temporary cache usage.

To evict expired keys efficiently, randomized expiration algorithms are discovered. A randomized web-cached eviction algorithm is proposed by Psounis and Prabhakar [19] in 2001 to approximate the performance of some deterministic algorithms with no extra data structure. Redis adopts a modified version of the randomized algorithm in practice [14]. Furthermore, Hyperbolic caching [4] is proposed in 2017 to decouple item priorities from eviction

MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA

ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422797

data structures and adopts a randomized algorithm for eviction. Besides, randomized eviction algorithm can also be applied to a sampling-based garbage collection algorithm [9]. Although randomized expiration algorithms are simple to implement, there still exists much space to improve their eviction efficiency.

We propose an innovative Bucket-Based expiration algorithm to improve the eviction efficiency with relatively low overhead. Our Bucket-Based expiration algorithm uses an array of buckets (hashmaps) to discretize the expiration information. We store keys to be expired in the same period of seconds or milliseconds into the same bucket. Later for the eviction procedure, we scan through buckets and evict keys in buckets first to efficiently evict expired keys. Only if time permitted, we continue with Redis randomized expiration algorithm to further reduce memory usage. Furthermore, our analysis and experimental results using Redis expiration algorithm as the baseline show the efficiency of our algorihtm.

The rest of the paper is organized as follows. In section 2 we present a review of randomized expiration algorithms. In section 3 we introduce our Bucket-Based expiration algorithm and required data structure in detail. We analyze the complexity for all operations of bucket maintenance and show a comparative analysis with Redis expiration algorithm. In section 4 we provide the experimental evaluation using YCSB [7], a benchmark tool to evaluate NoSQL databases. Experiments with differentsettings are designed and run. From the experimental results, we find our algorithm significantly reduces memory usage with an acceptable impact on throughputs. section 5 concludes the paper and provides some aspects of future works.

2 BACKGROUND AND RELATED WORKS

In this section, we will describe randomized expiration algorithms currently adopted in Redis, which is the referenced benchmark for our Bucket-Based expiration algorithm. The following subsections include the introduction of an original randomized algorithm by Psounis and Prabhakar [19], Redis expiration algorithm [2], and analysis of Redis expiration algorithm.

2.1 Randomized Algorithm

A randomized expiration algorithm proposed by Psounis and Prabhakar [19] is shown in Algorithm 1. It aims to adopt a simple algorithm to minimize the error rates of eviction. Here, an error is defined as the evicted one that *does not* belong to the least useful *n*th percentile of all keys.

Algorithm 1: Randomized Expiration Algorithm

```
if first_iteration:
  sample(N)
  evict_least_useful()
  keep_least_useful(M)
else:
  sample(N-M)
  evict_least_useful()
  keep_least_useful(M)
```

The randomized expiration algorithm maintains a sample pool, which is an array of size **N**. The algorithm is repeated every active expiration cycle. Within each cycle, it first initializes the sample pool and fills it by sampling **N** keys. Next, it evicts the least useful one and keeps the least useful **M** for the next iteration of eviction. In this algorithm, **n**, **N**, and **M** are hyperparameters that affect its performance. Through analysis and experiments, the original randomized expiration algorithm is claimed to be simple, which does not require complex data structures, and efficient with low error rates. Furthermore, Psounis and Prabhakar [20] analyze the optimal parameters for **N** and **M**.

2.2 Redis Expiration Algorithm

Redis adopts a modified randomized expiration algorithm as shown in Algorithm 2.

Algorithm 2: Redis Randomized Expiration Algorithm

```
def RedisExpiration(start_time):
  for i in range(n):
    while current_time()-start_time > \
        TIME_LIMIT:
      sample(N)
      num_expired = delete_all_expired()
      if num_expired / N < DELTA:
        return
```

Redis Expiration Algorithm functions as follows. Before eviction, the expiration information of all keys are stored in a hashmap **expires**, whose keys and values are the key names and the expiration time correspondingly. At the beginning of the algorithm, it retrives the **start_time** as the input. For every expiration iteration, it samples **N** keys from the set of keys with expiration information storing in the hashmap **expires**. Next, it deletes all expired keys within the **N** samples and stores the number of expired keys as **num_expired**. There are two cases that an iteration will break: either **num_expired** is smaller than **DELTA** (δ) percentage of **N** or the time for expiration has exceeded the **TIME_LIMIT**. Otherwise, the active expiration will proceed.

As for the data structure required by Redis, the hashmap **expires** is essential to store the expiration information. Furthermore, it serves as the sample pool for eviction, so that the sampling is done in a smaller key space rather than the complete one. In our algorithm discussed in the next section, its **expires** is directly inherited from the Redis one.

Redis Expiration Algorithm is similar to the Randomized Expiration Algorithm if we set the *usefulness* to be the time to evict. The earlier expiration time means the lower value, which has a greater priority to be evicted. Specially, we can define all expired keys to have zero value, which has the same priority that can be evicted simultaneously, while for those keys have not reached their expiration time, their value is infinite and must not be evicted under our current settings.

More specifically, Redis sets the following parameters:

- $N = 20$, which means we scan 20 keys per iteration.
- $n = 10$, which means it has 10 iterations per round.
- $\delta = 0.25$, which means the threshold for the proportion of expired keys is 25%.

2.3 Analysis of Redis Expiration Algorithm

Redis expiration algorithm has approximately **O(N)** time complexity for the eviction procedure. Considering the worst case where the samples are poorly sampled, it will continue evicting as long as δ of sample keys are expired. Therefore, to evict **n** expired keys, at most $\frac{1}{\delta}n$ scans are required, which show it has linear time complexity.

Concerning the result of it, although Redis expiration algorithm can guarantee that, after 10 iterations of expiration, it is very likely that the proportion of expired keys is lower than δ, which is 25%, it still has the drawbacks of low efficiency which means it may waste redundant computation resources on scanning the keys which are non-expired.

For one iteration, suppose the proportion of expired keys before an iteration is p, which means on average only pN keys are expired. Then, while it evicts on average pN expired keys, it also redundantly scans $(1-p)N$ non-expired keys at the same time. For example, when $p = 40\%$, there are 8 expired keys and 12 non-expired keys within the 20 samples on average. Those redundant scans are of low efficiency, which occupies precious computation resources, and furthermore, it may reduce throughputs and the overall performance of the system.

Consider the whole procedure of the algorithm, a disappointing consequence is observed, that is when p, the proportion of expired keys, approaches δ, the eviction efficiency would be significantly reduced. As we can see from the analysis, we will waste a $(1-p)$ proportion of computation resources. Initially, the efficiency is $p_{init} \geq \delta$, where p_{init} represents the initial proportion of expired keys, but as the algorithm continues, p decreases. And when p decreases, this proportion increases, showing a negative effect on the eviction effectiveness. This negative effect is more critical when p approaches δ when the efficiency is as low as δ. For example, in the current setting of Redis, δ is 25%, which means when p approaches 25%, the efficiency will approach 25% and there will be about tripple redundant scans. As a consequence, the overall efficiency is between δ and p_{init}.

Therefore, Redis Expiration Algorithm which is based on the randomized algorithm has relatively low efficiency for expiration, showing the potential of a more efficient expiration algorithm.

3 BUCKET-BASED EXPIRATION ALGORITHM

In this section, we will show our Bucket-Based expiration algorithm. First, we will describe our operations and algorithm. Next, we will illustrate our algorithm with an example. Finally, we will provide a comparative analysis of our Bucket-Based Expiration Algorithm against the randomized expiration algorithm described in subsection 2.2.

3.1 Algorithm

Bucket-Based Expiration Algorithm is a deterministic algorithm based on the idea of separating keys into buckets associated with discretized values (expiration times). It is not only designed to be simple, which uses only basic data structures of arrays, sets, and hashmaps, but also with relatively high efficiency in expiration.

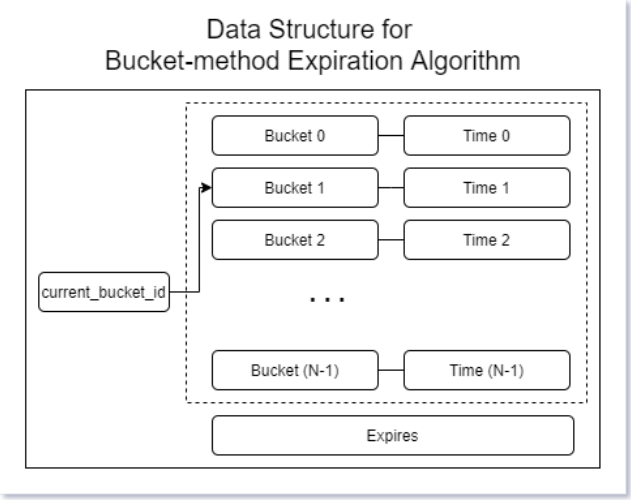

Figure 1: Data Structure for Bucket-method Expiration Algorithm

3.1.1 Data Structure. As shown in Figure 1, the data structure used in our Bucket-Based Expiration Algorithm is composed of **buckets** (an array of sets), **times** (an array of integers), **current_bucket_id**, and **expires** (a hashmap inherited from the original Redis). Moreover, it uses 2 parameters **BASE** (the divisor for discretizing values) and **NUM_BUCKETS** (the number of buckets). Each bucket is a set of keys and it is associated with a specific **time**, which is the quotient of the actual expiration timestamp (a Unix timestamp in millisecond) dividing the **BASE**. For example, when we set the **BASE** to be *1000*, an expiration timestamp *1590500917012* would have time *1590500917*, which can be interpreted as the timestamp in second. However, we also try other **BASE**s and our further experiments show that different **BASE**s would have significantly different performances. The **current_bucket_id** records the ongoing bucket. In addition, the hashmap **expires** stores all expiration information using key-value pairs, where keys are key names with expiration and values store the expiration times.

In our algorithm, the **Single-bucket constraint** is essential in the maintenance of **buckets**. It requires that every key in **expires** can be stored in **at most one** bucket, which must also be the corresponding bucket with its latest expiration time. We set this constraint for the following reasons. First, it can save space for maintaining buckets by eliminating redundant stores for the same key in different buckets. Second, it can make our operations **setExpire**, **removeExpire**, and **activeExpiration** much simpler. More importantly, since one bucket can only be used for a single time, if we do not remove the previous record from the previous bucket, its previous bucket will be unavailable to reuse, degrading the performance of our algorithm.

3.1.2 Main Algorithm. In this section, we will describe our main bucket-based expiration algorithm shown in **Algorithm 3** and **Algorithm 4**. Our **active Expiration** is a hybrid algorithm taking advantage of both buckets and the randomized **RedisExpiration** algorithm. First, it gets the current timestamp and stores it as the

current_timestamp. Then, it tries to go through all buckets starting from the **current_bucket_id** and proceeding in a round-robin manner. For each bucket, we attempt to evict it by calling **expireBucket** operation. If the status is **EXCEED_TIME_LIMIT**, which means we have reached the time limit of expiration, we directly break the loop. Otherwise, we will move the **current_bucket_id** forward to the next one until we have reached our initial position. If we have not reached our **TIME_LIMIT** after checking all buckekts, we will continue with the **RedisRandomizedExpiration**, which is exactly the same approach discussed in subsection 2.2. By utilizing the remaining available computation resources to evict expired keys missing from buckets, we are able to further save more memory resources if time permits.

The **expireBucket** operation takes a **bucket_id** and the current timestamp as its input. If the bucket is expired, so as all its keys, then we try to evict all keys in a bucket by iterating all keys only once, resulting in **O(N)** time complexity where **N** is the number of keys in a bucket. Four status codes are EMPTY, EXCEED_TIME_LIMIT, SUCCESS, and TIME_UNREACHED to prompt our **activeExpire** algorithm the current condition.

Algorithm 3: **activeExpiration**

```
def activeExpiration():
  current_timestamp = get_timestamp()
  for i in range(NUM_BUCKETS):
    status = expireBucket(
      current_bucket_id,
      current_timestamp
    )
    if status == EXCEED_TIME_LIMIT:
      break
    current_bucket_id=(current_bucket_id \
      + 1) % NUM_BUCKETS

  RedisExpiration()
```

Alorithm 4: **expireBucket**

```
def expireBucket(bucket_id, timestamp):
  if times[bucket_id] == 0:
    return EMPTY
  time = timestamp // BASE
  if time > times[bucket_id]:
    for key in buckets[bucket_id]:
      buckets[bucket_id].remove(key)
      expires.remove(key)
      data.remove(key)
      if get_timestamp() - \
        timestamp > TIME_LIMIT:
        return EXCEED_TIME_LIMIT
    return SUCCESS
  return TIME_UNREACHED
```

Two extreme situations are either the system is too busy or there exist too few keys in buckets. When the system is extremely busy, we may not have enough time to evict all keys in our first bucket. Then, we will try to evict as much as possible and keep the **current_bucket_id** unchanged, so that we will continue the eviction in the next expiration cycle. On the other side, when expired keys are few, we will go through all buckets once. Then, we will continue with the original **RedisRandomizedExpiration** algorithm. We partially degrade our algorithm to the original one to tradeoff **NUM_BUCKETS** and expiration efficiency, because the number of buckets have to be finite and some keys may have to be stored merely in **expires** but not **buckets**.

Following the practice of Redis, **activeExpiration** is called every 100 milliseconds by the main thread. Because the main operations are completed only in the main thread in Redis, the implementation does not require thread safety.

3.2 Bucket Maintenance Operations

To maintain our buckets, we have the following 4 operations: **timestampToBucketId**, **removeFromBucket**, **setExpire**, and **removeExpire**. All those operations are in **O(1)** time complexity on average.

For **timestampToBucketId**, we calculate the bucket id for a timestamp in two steps. First, we divide it by **BASE** and take the quotient as **time**. After obtaining the time, we divide it by **NUM_BUCKETS** and take the remainder, which is a simple hash for the time, mapping time into a **bucket_id** by adopting the modulo formula as the hash function. For example, when we set the **BASE** to be *1000*, an expiration timestamp *1590500917012* would have time *1590500917*, and with **BUCKET_NUM** being 60, its corresponding bucket id would be 1590500917012%60 = 52.

RemoveFromBucket tries to remove a key from a given bucket id. We first try to remove the key from the bucket's hashmap. Next, if the hashmap is empty, we reset its corresponding **times[bucket_id]** to be 0.

SetExpire is an operation to store the expiration information and maintain buckets. First, we obtain the previous timestamp from **expires** which stores all keys' expiration information. Next, we will remove the record from the previous bucket by calling **RemoveFromBucket** and **timestampToBucketId**. Finally, we will update **expires** and store the key to the new bucket if the bucket's time equal to our new time.

RemoveExpire is the opposite of **SetExpire** to remove the expiration information. It first gets and removes the expiration timestamp from **expires**. Using the timestamp, it checks the corresponding bucket's time and tries to remove the key from its corresponding bucket.

In short, we store all expiration information in a hashmap **expires**. Using the stored timestamp, we can calculate the corresponding bucket id to access its bucket and bucket time. All those operations are **O(1)** on average because the get and set operations for a hashmap is **O(1)** on average.

3.3 An Illustration

In this section, we will illustrate Bucket-Based Expiration Algorithm using a specific example shown in Figure 2. We assume the following constants of configuration:

- **NUM_BUCKETS**: 3
- **BASE**: 1000
- Period of **activeExpiration**: 1 second
- Initial **current_bucket_id**: 0

We also assume we have the computation ability to evict at most 2 keys per **activeExpiration** in order to illustrate the situation that too many keys expired.

And we have the following operations:

- 00:00 : **setExpire(a, 1023)**, **setExpire(b, 2100)**, **setExpire(c, 2020)**
- 00:01 : **setExpire(a, 2023)**, **setExpire(d, 2450)**, **removeExpire(c)**
- 00:02 : **setExpire(e, 5015)**
- 00:03 : **evict a, b**
- 00:04 : **evict d**

Initially, we have 3 empty buckets, **times** filled with zeros, and an empty **expires**. The **current_bucket_id** points to **bucket 0**. At time **00:00**, we set the expiration for **a**, **b**, and **c**. As a consequence, **bucket 1** stores **a**, **bucket 2** stores **b** and **c**, and **expires** stores the expiration time for all. At time **00:01**, we overwrite the expiration of **a** to be 2023, so **a** is first removed from **bucket 1**, then added to **bucket 2**, and its record in **expires** is also updated to be the newer one. At the same second, we also set the expiration for **d**, which is added to **bucket 2**. Moreover, we remove the expiration of **c** by removing it from both the **bucket 2** and **expires**. At time **00:02**, we set the expiration for **e**. The corresponding bucket for **e** should be **bucket 2** since *(5015 // 1000) % 3 = 2*. However, by checking **times[2]** we observe that it currently stores keys at time 2. Therefore, we cannot add **e** to **bucket 2** but merely stores it in **expires**. At time **00:03**, keys in **bucket 2** have been expired and we will evict them. However, because of our assumption of the computation limit, we can only evict two of them, and then we will time out. At time **00:04**, we will continue to evict the remaining one in **bucket 2**. Finally, we leave key **e** with expiration information stored in **expires**. The only way for it to be expired is when the system is not so busy that the original Redis Expiration Algorithm is invoked to evict it.

3.4 Performance Analysis

In this section, we will first analyze the performance of different operations in Bucket-Based expiration algorithm. Furthermore, we will analyze the performance of Bucket-Based expiration algorithm and make a qualitative comparison against Redis expiration algorithm.

3.4.1 Analysis of Operations. We design our bucket as the data structure *set* and implement it using a *hashmap*. For both *set* and *hashmap*, the *add*, *set*, *get*, *remove*, *check_in*, *check_empty* operations are all **O(1)** for time complexity on average, making our operations comparatively fast. Concerning the space complexity, operations above used only **O(1)** temporary space, and storing a *hashmap* requires **O(N)** in total for N keys.

Four operations which are **O(1)** for both time and space complexities include **removeFromBucket**, **timestampToBucketId**, **setExpire**, and **removeExpire**. For **removeFromBucket**, it consists of one *remove* and one *check_empty* which are both **O(1)** for time complexity and space complexity. **TimestampToBucketId** takes only two divisions and thus it is also **O(1)** for both complexities. **SetExpire** consists of one *get*, one *add*, one *set*, two **timestampToBucketId** operations, and one **removeFromBucket**, which are all **O(1)** for both complexities. **RemoveExpire** calls a *get*, one or two *remove*s, and a **timestampToBucketId**, and thus it is also in **O(1)** complexities.

ExpireBucket is a more complex operation that is in **O(N)** time complexity and **O(1)** space complexity, where **N** is the number of keys in a bucket. It will repeat the following actions at most once for all keys in it. The actions include three *removes*, and one *get_current_time()*, which are all in **O(1)** time complexity. Concerning the space complexity, those repetitions share the same **O(1)** space. Therefore, it is in **O(N)** time complexity and **O(1)** space complexity.

Additionally, since **activeExpire** is the key operation that invokes basic operations above to achieve our algorithm, we consider it the same analysis with the overall Bucket-Based expiration algorithm, which is presented in the next section. It has **O(**M**)** time complexity and **O(**K**)** space complexity. The next section will show the detail of its analysis.

3.4.2 Analysis of Bucket-Based Expiration Algorithm. For the analysis of Bucket-Based Expiration Algorithm, we will concern time complexity for one call of **activeExpiration** and space complexity for maintaining all expiration information at a moment. Suppose we have M keys to expire and K keys having expiration information at a moment, Bucket-Based Expiration Algorithm has **O(**M**)** time complexity and **O(**K**)** space complexity.

First, let $M = M_1 + M_2$, where M_1 is the number of keys in buckets, and M_2 is the number of those only stored in **expires** but not buckets. For those in buckets, we evict them by scanning through buckets and every key is scanned and evicted once, so the time complexity for this part of keys is $O(M_1)$. As for the other M_2 keys, they will be expired using Redis expiration algorithm, which is also $O(M_2)$*. Combining two components, the overall time complexity is $O(M)$.

Next, we apply the same method to analyze the space complexity. Let $K = K_1 + K_2$, where K_1 keys are in some buckets and K_2 keys are not in any buckets. For K_1 such keys, they require a record in buckets and **expires** for each, so the space complexity is $O(2K_1) = O(K_1)$. For the other K_2 keys, they only require one record in **expires** and the space complexity is also $O(K_2)$. Therefore, the overall space complexity is $O(K)$.

3.4.3 Comparison Between Bucket-Based Expiration Algorithm and Redis Expiration Algorithm. As we have seen in subsection 2.2 and section 3, Redis expiration algorithm is a randomized algorithm and Bucket-Based Expiration Algorithm is a hybrid algorithm of a deterministic one and Redis expiration algorithm. Because Redis expiration algorithm is a randomized algorithm, it has a relatively lower efficiency because a lot of non-expired keys may be checked, which wastes much computation resources. Moreover, it can only guarantee an asymptotic result that the final proportion will be lower than a pre-defined parameter *n%*, but it is hard to approach zero.

In contrast, our Bucket-Based Expiration Algorithm achieves better eviction efficiency by requiring a small proportion of memory for storing expiration information of key space and reducing more memory for the expired value space.

Our algorithm is composed of a deterministic part and a randomized part which is the same as Redis expiration algorithm, and the deterministic one plays a significant role in improving the efficiency

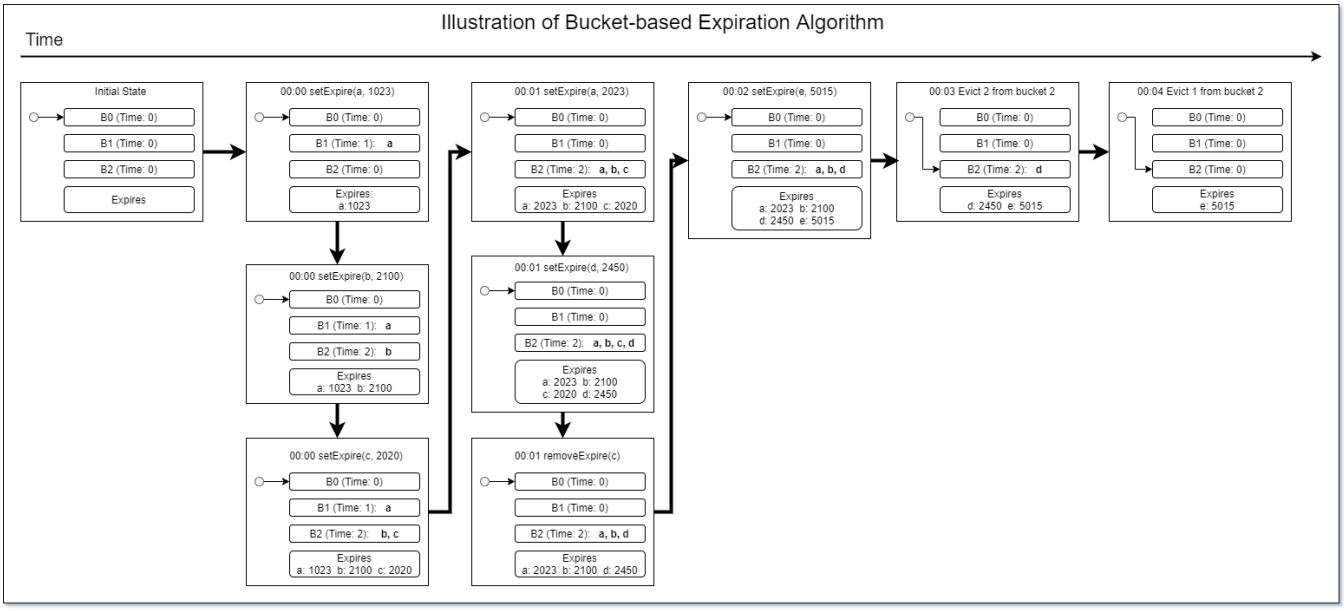

Figure 2: Bucket Illustration

of eviction. Although both algorithms are $O(M)$ for time complexity, the coefficient differs a lot. For the deterministic algorithm, since every key scanned must be within an expired bucket, if we scan M keys, it can evict M, which are all of them. Comparatively, for Redis expiration algorithm, we can only evict pM of them on average where p is the proportion of expired keys, so our deterministic part is $\frac{1}{p}$ times efficiency than Redis expiration algorithm. As eviction proceeds, p decreases, and the efficiency of Redis expiration algorithm decreases as well, while the deterministic one will not be affected, enlarging the gap of performance between them. When all expired buckets have been evicted, our algorithm will degrade to Redis expiration algorithm. It is supposed to be a rare case when the key space contains mostly non-expired keys, so it will finish shortly and only occupy a small proportion of the overall algorithm. Since the deterministic part is more efficient than Redis expiration algorithm and the randomized part is the same, the overall eviction efficiency of Bucket-Based Expiration Algorithm is better than Redis expiration algorithm.

To achieve higher eviction efficiency, it takes more space to store the expiration information for keys. However, memory for key space is usually smaller than it for value space with one order of magnitude, and our reduction for value space will overshadow our increase of memory for key space. Our algorithm requires two arrays with size **NUM_BUCKETS** to store the hashmap bucket and integer time, which is constant and relatively small. Moreover, every key will have at most one occurence in a bucket, which at most double the memory for key space. But on the other side, we can evict expired keys more efficiently and keep the value space much smaller, which overshadows the increase of key space memory. Therefore, the overall memory usage will be much smaller for our algorithm.

Besides the effect of low memory usage, our algorithm can perform better or similar throughputs against Redis expiration algorithm by balancing the tradeoff of the huge number of redundant expired keys and the small cost of maintaining buckets. Although our algorithm takes more overhead to maintain buckets, the increased eviction efficiency can reduce the key space and thus increase the efficiency of other operations. As a result, our algorithm performs better than or close to Redis expiration algorithm in throughputs. More experimental results will be shown in section 4.

The following example can help understand the comparison of differences in performance between them. Assume there are 100 keys in total with 40 of them are expired and we have the computation power to scan 20 of them. Using Redis expiration algorithm, we can only evict $20 \times 40 \div 100 = 8$ and leave 32 (80%) not evicted on average. However, if we use Bucket-Based Expiration Algorithm, we can fully utilize the computation power to evict 20 expired keys, and leave only 20 (50%) not evicted. As we can see from this example, when the proportion of expired key is $p = 40\%$, Bucket-Based Expiration Algorithm can evict $\frac{1}{p} = 2.5$ times keys over Redis expiration algorithm and save $1 - \frac{20}{32} \times 100\% = 37.5\%$ space storing expired keys.

In short, Bucket-Based Expiration Algorithm sacrifices **O(N)** more key space to improve the eviction efficiency and reduce the overall memory usage.

3.4.4 Analysis of SIZE, NUM_BUCEKTS, and BASE. In our algorithm, **NUM_BUCKETS** and **BASE** are used to determine the **size** of buckets. The **size** is defined as the product of them, which represents the total capacity of all buckets can hold. The greater size means our buckets can hold keys expired in a longer time interval, and if the size is smaller or merely equal to one TTL, our algorithm will suffer from the insufficiency of buckets.

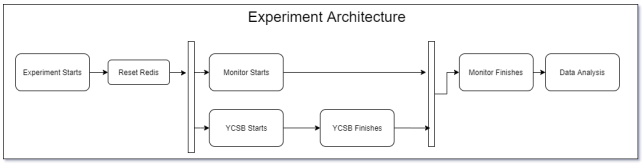

Figure 3: Experiment Architecture

With fixed size, BASE and NUM_BUCKETS are used to control the degree of discretization. With more buckets, our algorithm requires more memory to store more buckets. Meanwhile, the BASE will be smaller, and thus fewer keys will be stored in a bucket, which will result in lower overhead for bucket operations like *add*, *remove*, and *check_in* and higher throughputs.

Choosing the optimal parameters is about the tradeoff between memory usage and overhead of bucket maintenance. In subsubsection 4.5.2, we will analyze the effect of size experimentally. In subsubsection 4.5.6, we will analyze the effect of NUM_BUCKETS and BASE. Last but not least, we will provide the optimal choice of parameters in subsubsection 4.5.7.

4 EXPERIMENTAL EVALUATION

In this section, we study the performance of Bucket-Based expiration algorithm experimentally, choosing the original Redis expiration algorithm implemented in Redis 5.0 as our benchmark for comparison. We first implement our algorithm based on Redis 5.0 with different configurations. Next, we design and implement our experiment platform. Using our experimental system, we repeat our experiments in different settings and workloads to show our performance.

4.1 Performance Metrics

In our experiment, we mainly focus on 2 metrics: **memory usage** and **throughputs**. First, memory usage is our algorithm's objective. Our Bucket-Based expiration algorithm takes advantage of the distribution of TTL to improve eviction efficiency, which aims to significantly reduce memory usage, especially the peak memory usage of the NoSQL database. Meanwhile, we may sacrifice some computation resources because of the overhead of bucket operations. However, we want such negative effects to be minimum or even provide positive effects. Therefore, we also consider throughputs as the other metrics to evaluate the tradeoff between the benefits of memory and its required computation resources.

4.2 Experiment Platform Setup

To evaluate the performance of our Bucket-Based expiration algorithm, we build an automatic experiment platform and use it to run our experiments for different configurations. Our experiment architecture for a single experiment is shown in Figure 3. Our platform integrates a modified version of YCSB, a Redis Monitor, and a data visualization tool implemented in Python. The main modification of YCSB is to support multiple TTLs and their distributions by randomly choosing a TTL for each **Insert**, **Update**, and **ReadModifyWrite** operation and appending an **EXPIRE** command after it.

As for the Redis Monitor, it will call **INFO MEMORY** using *redis-py* [16] interface for every second.

For the procedure of one experiment, we will first reset Redis instances using **FLUSHALL**. Next, our experiment monitor and modified YCSB will start running. After our YCSB finishes, our system will wait for one more maximum TTL and then send a termination signal to our Redis Monitor to stop tracking and save all records in log files. In the final step, we will analyze our records by parsing logs and use Python's *NumPy* [18], *pandas* [17], and *matplotlib* packages [12] to visualize the results.

Besides the single-experiment mode, we can also run multiple experiments simultaneously as long as we run different Redis instances and the CPU and memory resources are within the capacity to avoid interference between concurrent experiments. In our practice, we run as many as 4 experiments to improve the experiment efficiency.

4.3 Experiment Environment Setup

For our experiment, we use a server with the configuration shown in Table 1. We set the maximum memory limits for Redis as 10GB per instance to avoid memory swap. In total, we have *7* instances (*1* original **Redis 5.0** and *6* modified ones with different configurations of **NUM_BUCKETS** and **BASE**) running on the server, but we divide our experiments into batches to avoid the influence of interference.

Table 1: Experimental Server Parameters

Parameter	*Value*
OS	Ubuntu 18.04.4 LTS x86_64
Kernel	5.3.0-53-generic
CPU	Intel Xeon E5-2640 v4 (40 Cores) @ 3.400GHz
Memory	193326MiB (188GiB)

4.4 Experiment Parameters

Table 2: YCSB Parameters

Parameter	*Fixed?*	*Range*
Key Size	YES	8 bytes
Value Size	YES	1000
Number of Operations	NO	3,000,000 (small) 10,000,000 (medium) 30,000,000 (large)
Type of Operations	NO	Read, Update Insert, ReadModifyWrite
Distribution of Operations	NO	Depend on the types of workloads.
TTLs	NO	{60}, {30, 60} or {15, 30, 45, 60} (seconds)
Distributions of TTLs	YES	Uniform
Number of Client Threads	NO	1 (Light Load) 4 (Heavy Load)

Table 3: Types and distributions of workloads for different workloads

Workload	Read	Update	Insert	ReadModifyWrite
workload a	50%	50%	0	0
workload f	50%	0	0	50%
workload i	10%	0	90%	0

4.4.1 YCSB Parameters. As shown in Table 2, we consider 8 parameters for YCSB including the key size, the value size per key, the number of operations, the types of operations and their distributions, the TTLs, and the number of client threads.

In our experiments, we consider 3 types of workloads as shown in Table 3, which represent 3 typical usages of expiration. *Workload a* is the same as YCSB workload a, which consists of a half **Read** and a half **Update**. It represents the usage of session storage; *workload f*, representing a user database that frequently updates user status or records user activities, is also the same as YCSB workload f with a half **Read** and a half **ReadModifyWrite**; a newly designed *workload i* is used to test the performance of **Insert** dominant workload which consists of 10% **Read** and 90% **Insert**.

Concerning expiration TTL, we choose TTLs from 60 seconds, 30 and 60 seconds, or 15, 30, 45, and 60 seconds. For multiple TTLs, we will randomly choose one with equal probability. We choose 60 seconds to be our maximum TTL because it is large enough to compare our special characteristics and properties.

Last but not least, the number of client threads of YCSB is from 1 to 4, where the more client threads represent the heavier loads. The maximum number of client threads in our experiment is 4 client threads because it is large enough to reach the maximum CPU limit for the stand-alone Redis.

Table 4: Parameters of Redis Instances

Name	NUM_BUCKETS	BASE	SIZE
Ref (**Redis 5.0**)	N/A	N/A	N/A
120-1000	120	1000	120 seconds
60-1000	60	1000	60 seconds
240-1000	240	1000	240 seconds
1200-100	1200	100	120 seconds
600-100	600	100	60 seconds
2400-100	2400	100	240 seconds

4.4.2 Algorithm (Redis) Parameters. As discussed in subsubsection 3.4.4, the parameters **NUM_BUCKETS** and **BASE** determine the **size** of buckets and influence the algorithm efficiency. Since the maximum TTL is 60 seconds, we design 6 different configurations as shown in Table 4. In total, we run 7 instances for each experiment to compare the performance of them.

4.5 Result Analysis

In this section, we will present our experimental results and analyze them comprehensively. The following subsections are composed of the following parts: **the overall performance**, **the effect of the number of operations**, **the number of client threads**, **number of TTLs**, and **the optimal choice of parameters**. In our analysis, we classify our instances into 3 categories based on their **size**, which are 60 seconds, 120 seconds, 240 seconds, which represent 1 TTL, 2 TTL, and 4 TTL respectively. One point required to be emphasized is that, in the macro scale analysis, we use the relative difference setting the Redis expiration algorithm as the baseline, where reduction of memory is given by $(mem_{baseline} - mem) / mem_{baseline}$ and the improvement of throughputs is given by $(throughputs - throughputs_{baseline}) / throughputs_{baseline}$.

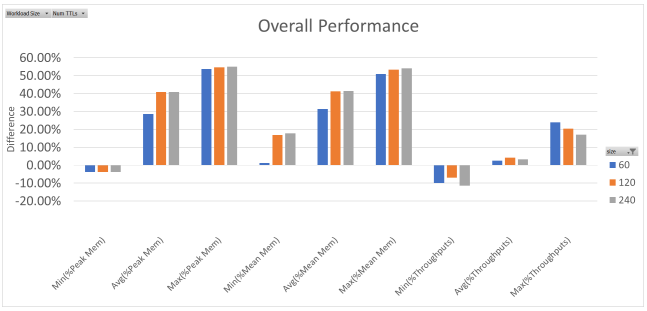

Figure 4: Overall performance for different bucket sizes: the range and average value of the percentage of memory reduction or throughput improvement between our algorithm and the baseline one.

4.5.1 Overall Performance. As shown in Figure 4, our algorithm can significantly reduce peak memory usage and mean memory usage with an acceptable impact on throughputs compared with the Redis expiration algorithm in most cases. Here, we calculate the reduction of memory and improvement of throughputs using Redis 5.0 as the benchmark. For configurations with size 60 seconds (1 TTL), it improves the poorest of memory usage. For peak memory, it reduces from **-3.74%** to **53.69%**, and for mean memory, it reduces from **1.20%** to **50.97%**. But meanwhile, it may potentially increase the throughputs the most, which is between **-9.96%** and **23.89%**. The degradation of its memory usage performance is due to its insufficient size and will be discussed detailly in subsubsection 4.5.2. For configurations with size 120 seconds (2 TTL) and 240 seconds (4 TTL), they both significantly reduce memory usage with a low sacrifice of throughputs. They on average reduce memory usage by about **40%** and at most reduce memory usage by about **54%**. Considering throughputs, the configurations with size 120 seconds (2 TTL) is slightly better than the configurations with size 240 seconds (4 TTL). The worst throughputs are decreased by **11.45%** and the best ones are improved by **20.5%**.

In Figure 5, Figure 6, and Figure 7, we show the performance on workload a, f, and i respectively. We observe that on all three types of workloads, our algorithm's performance is similar, which can significantly decrease the memory usage with low impacts on throughputs. Therefore, in the following detailed analysis, we will use **workload a** as an example.

In summary, our algorithm can significantly reduce memory usage with low influence on throughputs. Our performance is more related to the size of buckets rather than the type of workloads.

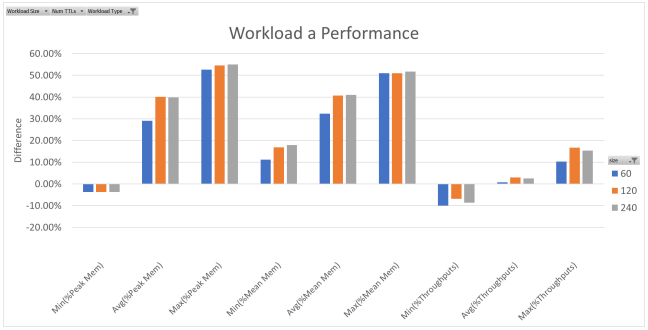

Figure 5: Performance of workload a: the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis for workload a.

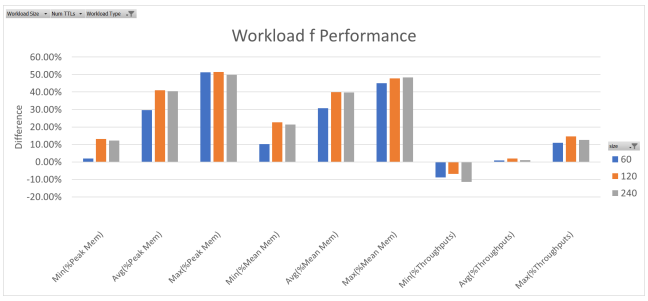

Figure 6: Performance of workload f: the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis for workload f.

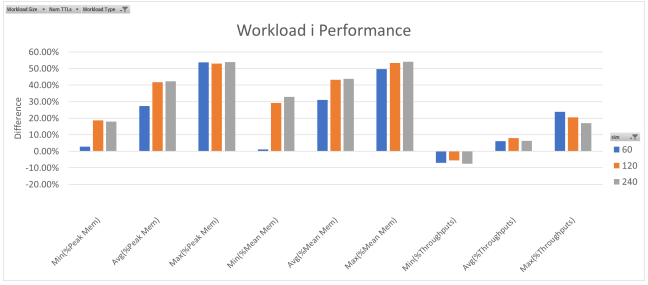

Figure 7: Performance of workload i: the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis for workload i.

4.5.2 Effect of Bucket Size. In this section, we will discuss the effect of different bucket sizes. In total, we have 3 bucket sizes as shown in Table 4. As discussed in subsubsection 3.4.4, configurations with different bucket sizes perform differently, especially for the bucket size 60 seconds. In Figure 8 and Figure 9, we illustrate the effect of size using the example of the experiment on workload a with single TTL, large workload, and 4 client threads. The blue

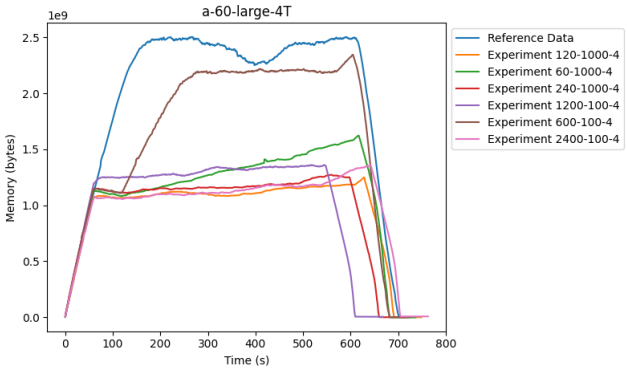

Figure 8: Memory usage versus experiment time with the setting: Workload a, large data size, 60 seconds TTL, 4 client threads.
Seven instances: Reference: the baseline Redis; Experiment-X-Y-4: our algorithm with bucket size X and number of buckets Y.

curve shows memory usage on Redis 5.0 for reference, which is significantly greater than all others. For the instance **600-100**, it degrades partially to the reference case because of its insufficient buckets. Because its size is merely 1 TTL, as long as we cannot evict expired keys in time, buckets will be unavailable to reuse and our algorithm will degrade to Redis expiration algorithm. The other instance with size 60 seconds is **60-1000**, which defers and alleviates the performance degradation. The reason behind it is that it has fewer ($\frac{1}{10}\times$) buckets and larger (10×) BASE, and as a result, it has lower overheads for bucket operations compared with **600-100**. For the other 4 configurations, their memory usages are similar because they have enough buckets.

Concerning throughputs, all configurations have a little decrease or even increase of throughputs, showing our algorithm balance the tradeoff between memory usage and bucket overheads.

4.5.3 Effect of The Number of Operations. In our experiment, the number of operations determines the running time and workload size, where the greater number of operations means the longer term of the same load. In Figure 10 and Figure 11, we show the memory usage and throughputs for the experiment of workload a, small workload, and 4 client threads. As we can see, for the small workload size, all operations will be finished before one TTL (60 seconds), so no eviction is available during the operations. In this case, the difference in memory usage and throughputs shows our algorithm's overhead. A key difference of the behavior is that after 60 seconds, our algorithm will evict keys almost linearly, while the referenced one will start eviction when the proportion of expired keys exceeds 20%. As for the throughputs, our algorithm has a minor negative effect or positive effect on the throughputs, showing our relatively low overheads of bucket maintenance.

Comparatively, the case for a large workload is shown in Figure 8 and Figure 9 describing the long-term behavior of performance. The long-term performance can better represent the performance

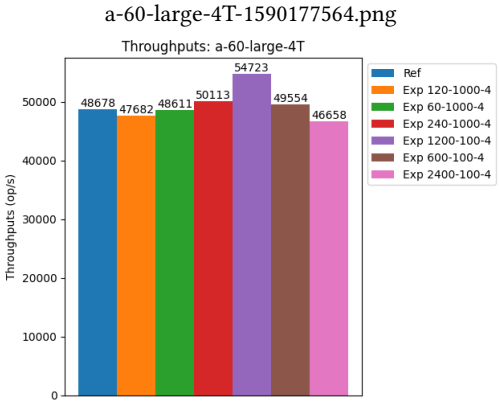

Figure 9: Throughputs with the setting: Workload a, large data size, 60 seconds TTL, 4 client threads.
Seven instances: Reference: the baseline Redis; Experiment-X-Y-4: our algorithm with bucket size X and number of buckets Y.

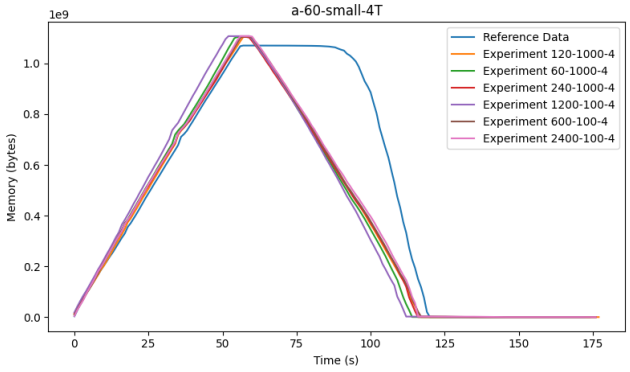

Figure 10: Memory usage versus experiment time with the setting: Workload a, small data size, 60 seconds TTL, 4 client threads.
Seven instances: Reference: the baseline Redis; Experiment-X-Y-4: our algorithm with bucket size X and number of buckets Y.

of our algorithm under the real heavy-load scenarios where eviction and operations occur simultaneously. Therefore, we focus on the performance under the large workload size in our following analysis.

As for the long-term performance, we observe that our algorithm can always reduce memory usage and have an acceptable impact on the throughputs as shown in Figure 12. The configurations with size 60 perform the worst which in the worst case may not reduce memory usage. For the greater sizes like 120 and 240, they can reduce memory usage at least about 30% and at most about 50%. As for throughputs, our algorithm will reduce no more than 10% but can also improve up to 20.5%. Therefore, our algorithm has good long-term performance in reducing memory usage.

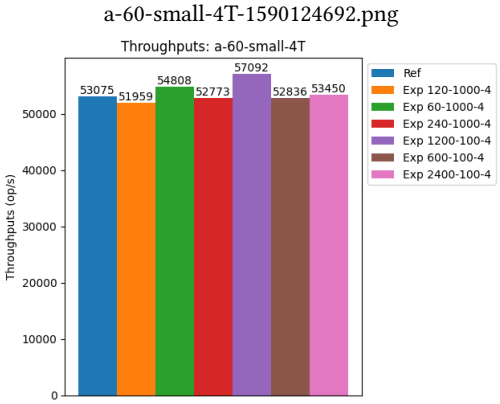

Figure 11: Throughputs with the setting: Workload a, small data size, 60 seconds TTL, 4 client threads.
Seven instances: Reference: the baseline Redis; Exp-X-Y-4: our algorithm with bucket size X and number of buckets Y.

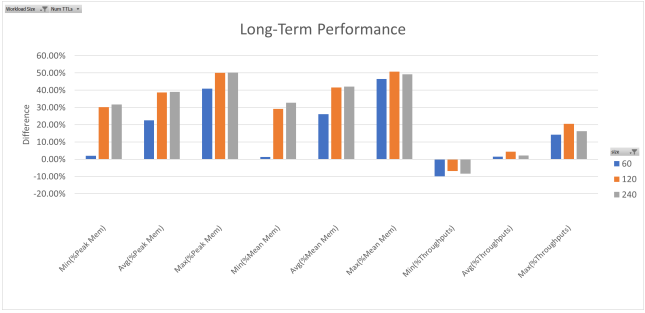

Figure 12: Long-Term Performance: the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis for large data size.

4.5.4 Effect of The Number of Client Threads. The number of client threads determines the load of clients, where more client threads will simulate heavier loads. Figure 13 and Figure 14 show the characteristics of our algorithm under 1 client thread. Compared with the case under 4 client threads in Figure 9 and Figure 8, its execution for the same number of operations last longer and its peak memory is smaller. However, in either case, our algorithm can significantly reduce the memory usage, and our throughputs are also within an acceptable range.

4.5.5 Effect of The Number of TTLs. The number of TTLs represents the randomness of TTL distributions. In our experiment, every key will have an associated expiration time. With a single TTL 60 seconds, all keys without further operations will be expired in 60 seconds. As shown in Figure 15, configurations with size 60 perform much worse than those with larger sizes because of their insufficient buckets. As for the throughputs, our algorithm at most decreases about 4% throughputs, but for most cases, it improves the throughputs. Figure 16 and Figure 17 show the performance with 2 and 4 TTLs. We observe that the increased randomness of

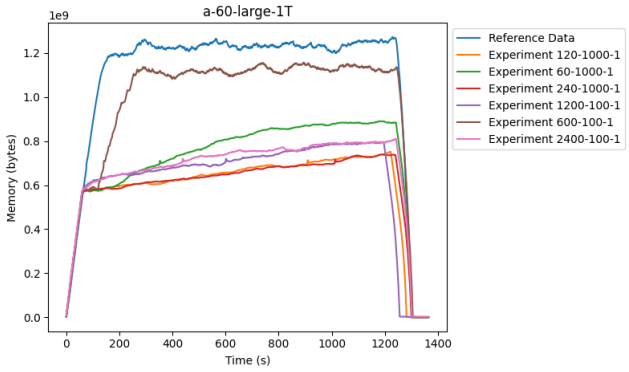

Figure 13: Memory usage versus experiment time with the setting: Workload a, large data size, 60 seconds TTL, 1 client thread.
Seven instances: Reference: the baseline Redis; Experiment-X-Y-1: our algorithm with bucket size X and number of buckets Y.

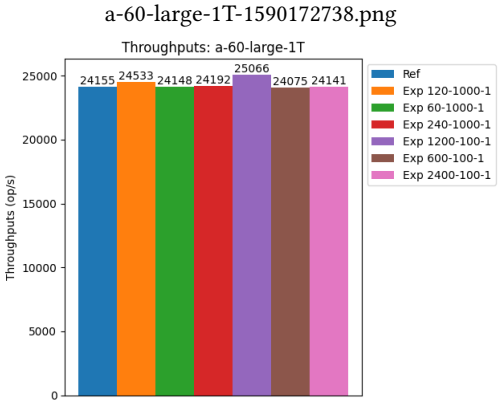

Figure 14: Throughputs with the setting: Workload a, large data size, 60 seconds TTL, 1 client thread.
Seven instances: Reference: the baseline Redis; Exp-X-Y-1: our algorithm with bucket size X and number of buckets Y.

TTLs will alleviate the performance of configurations with size 60. For configurations with size 120 and 240, they have excellent performance on memory usage in all 3 cases, showing that, with sufficient buckets, our algorithm can suit the cases with randomly distributed TTLs.

4.5.6 Effect of NUM_BUCEKTS and BASE. In this section, we will analyze the effect of NUM_BUCKETS and BASE experimentally. From subsubsection 4.5.2, we observe that with size merely equal to one TTL, our algorithm may degrade to the referenced one significantly, but configurations with size 2 TTLs (120 seconds) and 4 TTLs (240 seconds) perform similarly. To further analyze the effect of NUM_BUCKETS and BASE, we choose the size 120 and 240 seconds and the large workload. The performance is shown in Figure 18. Comparatively, we observe that the one with larger BASE

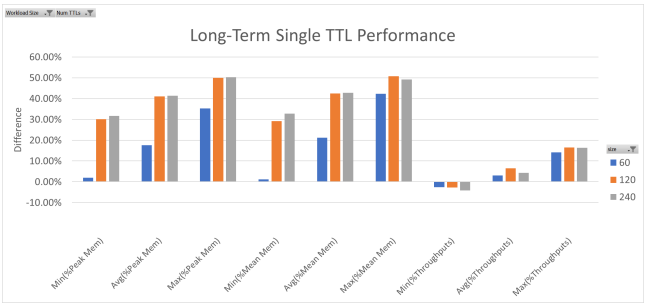

Figure 15: Long-term performance with a single TTL (60s): the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis.

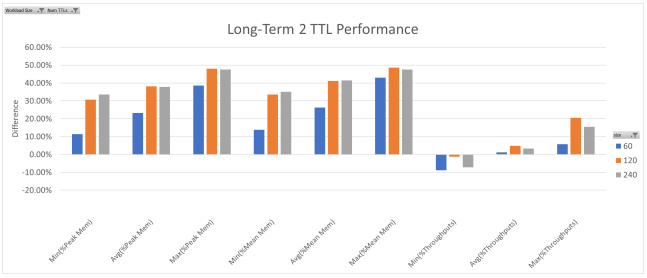

Figure 16: Long-term performance with 2 TTLs (30s and 60s): the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis.

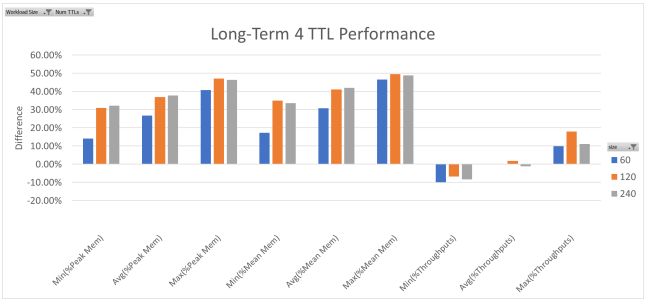

Figure 17: Long-term performance with 4 TTLs (15s, 30s, 45s and 60s): the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis.

(1000) will have lower memory usage but also smaller throughputs. It is a tradeoff between memory usage and overhead of bucket maintenance. For the ones with larger BASE, they have fewer buckets and each bucket contains more keys. As a result, fewer buckets help to save the memory used for buckets, but meanwhile, with more keys stored in a bucket, the overhead for bucket operations will increase, which reduces the throughputs. Therefore, the larger

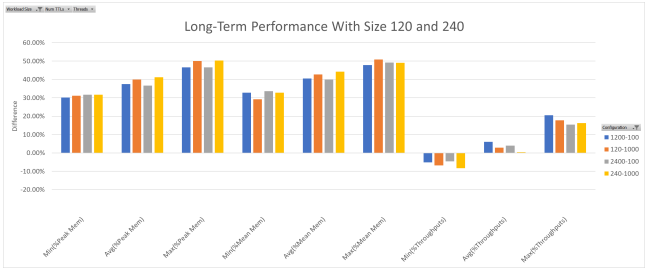

Figure 18: Long-term performance of Bucket Size 120 and 240: the range and average value of the percentage of memory reduction and throughput improvement compared with the baseline Redis.
Four instances: X-Y means an instance with X buckets and BASE Y.

BASE and fewer buckets will save a bit more memory but sacrifice some throughputs.

4.5.7 Optimal Choice of Parameters. In this section, we will conclude our experimental analysis with the optimal choice of parameters from our current settings.

From subsubsection 4.5.2, we observe that our eviction performance significantly degrades when the size of buckets is insufficient. Hence, we should not choose the size merely equal to one TTL (60 seconds).

Concerning the size 120 and 240 seconds, we balance the extra memory and computation overhead for buckets. The analysis of subsubsection 4.5.6 shows that more buckets may require some memory to achieve better throughputs. From Figure 18, we can conclude that the configuration **120-1000** balance memory usage and throughputs the best among them.

Therefore, an optimal choice is with NUM_BUCKETS 120 and BASE 1000 which achieves relatively low memory usage and good throughputs.

5 CONCLUSION AND FUTURE WORKS

In this work, we have proposed a Bucket-Based expiration algorithm to efficiently evict expired keys, which is a hybrid algorithm with a deterministic one with buckets and a randomized one inherited from Redis expiration algorithm. It adopts a simple data structure and has low maintenance overhead. Besides the theoretical analysis of time complexity, we further implement our algorithm into Redis 5.0 and have a comprehensive experimental result using the original Redis 5.0 as our benchmark. The result of low memory usage and slightly affected throughputs shows that our algorithm significantly improves eviction efficiency with an acceptable maintenance overhead.

In general, our algorithm can be further applied to other value-based eviction policies where the value of eviction priority can be discretized and grouped into buckets. Besides, extending our algorithm to a database cluster like Redis Cluster is also valuable and may have more variations depending on the base of a database cluster. Furthermore, the dynamic configuration of buckets using methods of optimization or machine learning is another promising issue. We will address these issues to extend our algorithm in the future.

ACKNOWLEDGMENTS

This work is partially supported by Key-Area Research and Development Program of Guangdong Province under contrat No. 2019B010155002.

REFERENCES

[1] [n.d.]. MongoDB: The most popular database for modern apps. https://www.mongodb.com/
[2] [n.d.]. Redis: EXPIRE key seconds. https://redis.io/commands/expire
[3] Kyle Banker. 2011. *MongoDB in action.* Manning Publications Co.
[4] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17).* 499–511.
[5] Josiah L Carlson. 2013. *Redis in action.* Manning Publications Co.
[6] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo. 2016. Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. In *2016 IEEE Trustcom/BigDataSE/ISPA.* 1660–1667.
[7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing.* 143–154.
[8] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A survey on NoSQL stores. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–43.
[9] Biplob Debnath, Srinivasan Krishnan, Weijun Xiao, David J Lilja, and David HC Du. 2011. Sampling-based garbage collection metadata management scheme for flash-based storage. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST).* IEEE, 1–6.
[10] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.
[11] Jing Han, Ee Haihong, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *2011 6th international conference on pervasive computing and applications.* IEEE, 363–366.
[12] John D Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in science & engineering* 9, 3 (2007), 90–95.
[13] T. Lee, Y. Kim, and E. Hwang. 2018. Abnormal Payment Transaction Detection Scheme Based on Scalable Architecture and Redis Cluster. In *2018 International Conference on Platform Technology and Service (PlatCon).* 1–6.
[14] S. Li, H. Jiang, and M. Shi. 2017. Redis-based web server cluster session maintaining technology. In *2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD).* 3065–3069.
[15] B. Luo, W. Zhu, P. Li, and Z. Han. 2018. Distributed Dynamic Cuckoo Filter System Based on Redis Cluster. In *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS).* 244–247.
[16] Andy McCurdy. 2020. andymccurdy/redis-py. https://github.com/andymccurdy/redis-py
[17] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.
[18] Travis E Oliphant. 2006. *A guide to NumPy.* Vol. 1. Trelgol Publishing USA.
[19] Konstantinos Psounis and Balaji Prabhakar. 2001. A randomized web-cache replacement scheme. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, Vol. 3. IEEE, 1407–1415.
[20] Konstantinos Psounis and Balaji Prabhakar. 2002. Efficient randomized web-cache replacement schemes using samples from past eviction times. *IEEE/ACM transactions on networking* 10, 4 (2002), 441–454.
[21] Enqing Tang and Yushun Fan. 2016. Performance comparison between five NoSQL databases. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD).* IEEE, 105–109.
[22] Yinqian Zhang, Fabian Monrose, and Michael K. Reiter. 2010. The Security of Modern Password Expiration: An Algorithmic Framework and Empirical Analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10).* Association for Computing Machinery, New York, NY, USA, 176–186. https://doi.org/10.1145/1866307.1866328