HASS : Hotness-Aware Sharding Split for MongoDB

Jin Yang, Heejin Yoon

Abstract— **With increasing size of data, distributed database system is more necessary system for most companies. However, because data size is too massive, cost of storing and managing them by company's own server is too expensive. Therefore, cloud database is emerged. By using cloud database, companies manage its massive data on cloud and cost is reduced. Simultaneously, limitation of relational database is clear in aspect to scalability and flexiblity of data format. For overcoming these problem, NoSQL is more remarkable scheme for distributed database. MongoDB[1] is representative example of NoSQL distributed cloud database. It supports distributed database system using sharding technique. And it has 2 types of sharding, range sharding and hashed sharding. They have their own advantages and disadvantages, so user can select type followed by their preference. However, both does not consider "hot spot", which is spot contains frequently accessed data. This ignorance leads imbalanced I/O overhead between multiple server. Therefore, we designed hotness-aware sharding split and implemented it on MongoDB[1]. For evaluation, we used YCSB[2] and showed fairness of disk utilization and performance. With HASS dynamic tuning, our design can get performance improvement, 2.43% of insert, 15.72% of read and 10.8% update operation in zipfian workload with small size of dataset compared to default MongoDB performance.**

I. INTRODUCTION

As time goes on, size of data is increasing. Traditionally, many companies stores and manages their data with their own storage device. However, size of data is too massive and they needs to another method to store their large data. Therefore, "cloud database" emerged and is getting more remarkable. By definition, cloud database is database which is operated on cloud and it provides access to database as a service. Due to fast develop of cloud database, companies can lessen the burden of managing massive data and reduce cost.

With development of database, there are many scheme used for database such as MySQL. Traditionally, relational database is commonly used. Because data can be well-organized and structured by using relational database. However, it has very 2 severe limitations for massive data, scalability and data format. Main reason why scalability of relational database is too poor is supporting transaction with ACID. Moreover, relational database targets for single-server node. Therefore, for scaling it, user needs to pay for more expensive server. Handling massive data requires scalable database, but relational database is not proper solution. Secondly, relational database support structured data format which requires many attributes. It can helps efficient query, but with increasing of unstructured data, user including many companies needs another solution.

Therefore, NoSQL scheme is getting more commonly used for cloud database. Advantage of NoSQL is 2 things, scalability and flexibility which are main limitation of traditional database. In contrast of relational database, NoSQL supports record-level atomicity. Therefore, it can be scalable horizontally. It means that simply adding server can perform larger transactions. Second reason is flexibility. In other words, it can support unstructured data. In case of relational database, it requires many attributes for data. However, NoSQL doesn't require any constraint for data format, just key-value pair is sufficient. Therefore, many companies prefer NoSQL scheme due to these reason and NoSQL is used in many platforms including key-value store[3],[4]. MongoDB[1] is representative NoSQL cloud database system. It is document-oriented database and supports distributed database by using its own efficient technique, "sharding". In MongoDB[1], chunk is used for unit of data item group. Sharding splits chunks and migrate them to proper server node for balance. There are 2 sharding techniques, range sharding and hashed sharding. They has their own advantages and disadvantage. Considering feature of their own workload, user can select type of sharding.

However, 2 types of sharding could be more optimized for common real-world workload. In many real-production workload, data access is skewed. In other words, specific data is frequently accessed and they are called as "hot data". Conventional sharding techniques of MongoDB[1] doesn't consider the hot data, only focused on chunk size. By this reason, specific server node which contains many hot data suffers from high I/O. It causes unfairness of disk utilization between multiple servers. In our report, "fairness" is how similar amount of disk utilization between multiple servers. If specific server has many hot chunk which contains hot data and access is concentrated to the server, disk I/O of the server will be highly increasing compared to other servers as shown in figure 2. In figure 2, x-axis is id of server and y-axis is disk utilization of each server. However, if sharding technique distribute hot data to many server node, its I/O overhead could be relieved.

Therefore, we designed and implemented hotness-aware sharding split for MongoDB[1], HASS. We used "tuning interval" which will used for hotspot, which is the spot contains hot data. Tuning interval supports 2 ways, static and dynamic. Static tuning interval moves split points with static interval. It is very light-weight method of hotness-aware sharding split. However, we found it has very serious problem during design step. If hot spot is too skewed in chunk, static tuning interval could cause extreme chunk size imbalance. The imbalance can influence

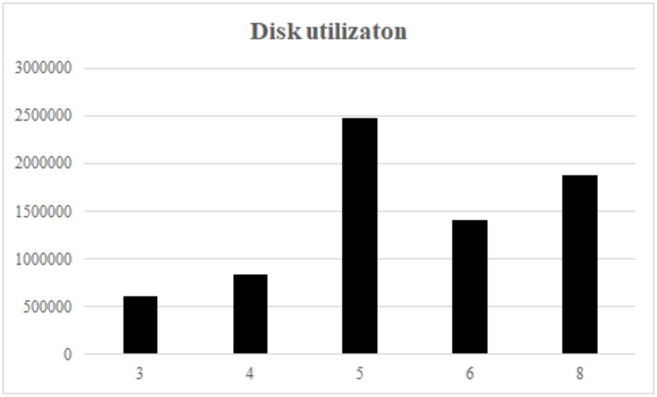


Fig. 1. Unfairness of Disk utilization in MongoDB

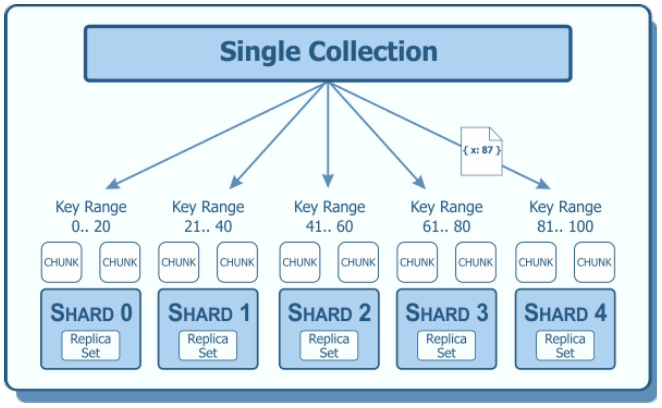


Fig. 2. Sharding technique in MongoDB[5]

performance of operations because specific chunk has too many items. Therefore, we designed dynamic tuning interval, which relieves size imbalance problem and implement the algorithm on MongoDB[1]. Its performance is better than conventional MongoDB, especially for zipfian workload which is our target workload. Moreover, HASS tries to achieve fairness between multiple servers.

To summarize, there are 2 main contributions.

- We design fair hotness estimation algorithm for chunk. This algorithm is cheaper than prediction and recording access count for every item.
- We implement HASS, hotness-aware sharding split by dynamically adjusting shard key considering hot spot. In result, it can reduce I/O unfairness between multiple server nodes.

II. Background

A. MongoDB

MongoDB[1] is open source document-oriented NoSQL database system. Like key-value pair, document is also efficient data format and flexible. Therefore, it is called "scheme-free database". By using the advantage of NoSQL, MongoDB[1] also supports highly scalable database system. This horizontal scalability is more appropriate with handling massive data. Therefore, it is commonly used for distributed cloud database system. Also, it allows replication for each database server. It leads high availability of stored data.

B. Sharding technique

Storing data into single server can lead high overhead and single server may not store every data item due to capacity limitation. Therefore, distributed database system is necessary. Sharding is core technique for distributed database system because it can make data item stored into multiple server and managed efficiently. Using this technique leads partitioned data group, which is called "chunk". Chunk will be partitioned by types of sharding. There are 2 types of sharding, range sharding and hashed sharding. Range sharding partitioned by key range. For example, key 12 is inserted to shard which chunk range contains 12. This type allows for efficient queries where reads target documents within a contiguous range. However, if shard key selection is too poor, both read and write performance may decrease. In contrast, hashed sharding calculates hash value of key and then insert it followed by hash value. Using hashed sharding provides more even data distribution across the sharded cluster. However, when performing a range query, it may request a query from most of the shards. Therefore, it can cause higher read overhead than ranged sharding due to bad locality. Each type of sharding has their own advantages and disadvantages. User can select appropriate type reflecting feature of their workload.

Still, these 2 types of sharding does not consider hot spot. If hot spot is concentrated to specific shard server, then the server suffers from high I/O overhead. Therefore, it is important that the hot spot has to be splitted to multiple shard server to resolve this unfairness between multiple servers. By this reason, we design and implement hotness aware algorithm and adjusting shard key for splitting hot spot.

III. Design

To tuning split keys, we go through two main processes.

1) Calculate the average using the shard key of the items in chunk.
2) When the split key is selected by the default setting of MongoDB, tune the split key using the average.

A. Calculate Average

There have been numerous studies for hotness-aware in database research. For example, ElasticBF[6] uses a fine-grained heterogeneous Bloom filter management scheme with dynamic adjustment according to data hotness in order to leverage access skewness feature,also called hotspot, which is very common among SSTables or even small-sized segments within each SSTable in LSM-tree based key-value (KV) stores. In another research, TRIAD[7] leverages hotspot in data popularity to avoid frequent I/O operations

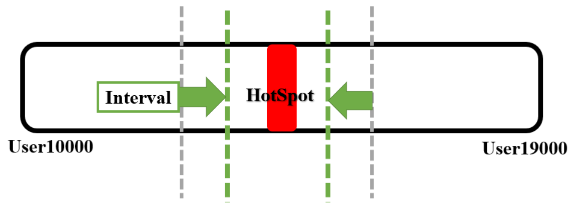
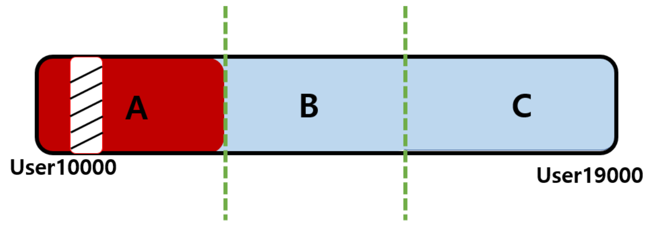
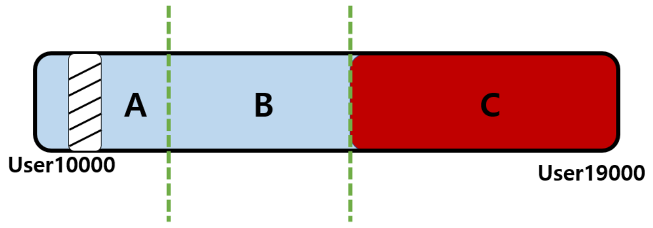


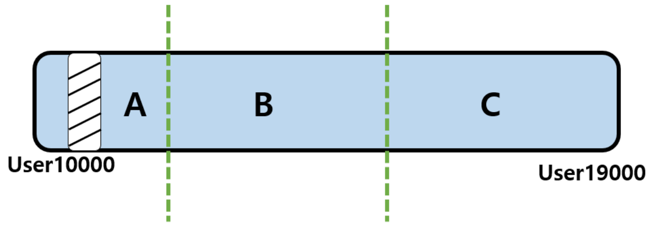
Fig. 3. How HASS tunes split points



(a) Original MongoDB Split



(b) HASS Static Split



(c) HASS Dynamic Split

Fig. 4. Comparison between original mongoDB split, HASS static tuning split and dynamic tuning split

on the most popular keys at the LSM memory component level. However, in these case, they do not consider "split" and distributed environment. Moreover, they have to manage another data structure for recording hotness. Recording and additionally storing is very expensive operations. Similarly, to predict hotspot is not only expensive complicated algorithm but also difficult to learn small data set which is constantly inserted. [8] For light-weight hotness awareness, we calculated average value to distinguish hotspot in the chunk. When an operation like update and read is requested for a particular chunk, the value of the item touched by that operation is updated to the average value. This can happen in duplicate, so if an item is accessed several times, it will have a significant impact on the average.

B. Tune Split Points

When the chunk size gets bigger and a split is attempted to occur on the chunk, MongoDB[1] selects split points so that the chunk size can be uniform. For example, suppose that there was a split attempt when an object with a shard key of 100000 entered the chunk after inserting objects with a shard key from 1 to 99999, sequentially. (By the way, 1 to 100000 do not exceed the minimum and maximum values of chunk.) In this case, if four split points are selected, the MongoDB[1] adopts 20000, 40000, 60000, and 80000 as the split keys by using the average object size and number of object to find approximately how many keys each chunk should have. We add steps to calibrate these selected split points considering the size of the chunks to be divided by MongoDB[1]. Tuning method is simple. Based on the average shard key value of objects calculated by each chunk, each split point is moved close to the average. At this time, we call the degree to the distance, which split point has changed due to tuning interval. Specifically, if the tuning interval of each split point depends on the relative distance of the split point and the average of objects, then it is defined as dynamic tuning, or static tuning if the tuning interval of each split is the same regardless of the distance.

Algorithm 1: Static Tuning of Split Points

```
1  function Static Tuning $(C, t_s, avg, SplitPoints)$;
   Input : target chunk $C$, static tuning ratio $t_s$, chunk
           average $avg$ and default split points
           SplitPoints[$p_1 ... p_n$])
   Output: TunedSplitPoints[$p'_1 ... p'_n$]
2  $ChunkRange = C.max() - C.min()$
3  $tuning_s = ChunkRange \times t_s$
4  for $i = 1$ upto $n$ do
5      if $p_i < avg$ then
6          $p'_i += p_i + tuning_s$;
7      else
8          if $p_i > avg$ then
9              $p'_i -= p_i + tuning_s$;
10         end
11     end
12 end
```

1) Static Tuning: In static tuning, all split points are moved (as much as tuning interval) by the same size. The tuning interval is calculated as follows.

$$\textbf{Static Tuning Interval} = \textbf{chunk range} \times \textbf{tuning ratio}_s \quad (1)$$

where **chunk range** is $Chunk.max() - Chunk.min()$, obtained by subtracting the minimum shard key of the object from maximum shard key inserted the corresponding chunk, and **tuning ratio**$_s$ is a tuning parameter between 0 and 1. In the experiment, the default value of it is 0.1.

Using the average and tuning interval, if the split key is less than the average value, HASS moves the split key to the right by adding a tuning interval to the split point. Conversely, if the split key is greater than the average value, move the split key to the left by subtracting the tuning interval on the split key. The above process is detailed in Algorithm 1. Through this process, each split point is adjusted to reduce the size of the chunk containing the hotspot.

Algorithm 2: Dynamic Tuning of Split Points

```
1  function Dynamic Tuning
     ($C, t_s, avg, SplitPoints, t_d$);
   Input  : target chunk $C$, static tuning ratio $t_s$, chunk
            average $avg$, default split points
            SplitPoints[$p_1...p_n$]) and dynamic tuning
            ratio $t_d$
   Output: TunedSplitPoints[$p'_1...p'_n$]
2  $ChunkRange = C.max() - C.min()$
3  $tuning_r = ChunkRange \times t$
4  $tuning_l = ChunkRange \times t$
5  Var $a = 1$
6  for $i = 1$ upto $n$ do
7      if $p_i <= avg$ then
8          $a = a + 1$;
9      else
10         break;
11     end
12 end
13 for $i = a$ upto $n$ do
14     $p'_i - = p_i + tuning_r$;
15     $tuning_r = tuning_r \times t_d$
16 end
17 for $i = a - 1$ downto $1$ do
18     if $p_i == avg$ then
19         continue;
20     end
21     $p'_i - = p_i - tuning_l$;
22     $tuning_l = tuning_l \times t_d$;
23 end
```

2) Dynamic Tuning: Static tuning, which moves all split keys by the same tuning interval, does not take into account the size of a chunk that does not include hotspot at all. Thus, if the hotspot is located at skewed side of the chunk as chunk A in 4, the opposite side of the chunk may cover inefficiently large chunk range, resulting in an imbalance in the size of the chunk C. For size imbalance, We add dynamic tuning interval.

$$\textbf{Dynamic Tuning Interval} = \textbf{TI}_S \times (\textbf{tuning ratio}_d)^n \quad (2)$$

where $\textbf{TI}_S$ is Static Tuning Interval,

tuning ratio$_d$ is a dynamic tuning parameter between 0 and 1 whose default value is 0.5,

and **n** is relative distance away from average value of the chunk. n is obtained by the number of other split points between the corresponding split point and the average value. If the split point is closest to the average having no other split point between them, n of the point is 0.

By doing so, HASS can reduce the size of the chunk, including the hotspot, while at the same time adjusting the size of the other chunk to prevent it from growing abnormally.

IV. Evaluation

In evaluation section, we will check fairness of disk utilization between conventional MongoDB[1] and our design, HASS. And we will compare performance of HASS using static tuning interval and dynamic tuning interval with MongoDB[1] in uniform workload and zipfian workload.

A. Experimental Setup

We implement HASS on MongoDB[1] which version is 3.6.18. For distributed database, we use 5 servers. Each use 2 Intel(R) Xeon(R) CPU E5-2640 v3 processor operating at 2.60GHz with 8 cores and 32G DRAM. And 1 router server, 1 config server and 3 shard servers are main components of distributed database which we operated. Each config server and shard servers are replicated, each replica is composed of 1 primary server and 2 secondary servers. Because we define "frequently access" as "frequently updated", we use YCSB-A workload[2]. Each item size is 1KB which is default of YCSB[2]. Number of operation is 1M for dynamic tuning interval evaluation and 0.1M for static tuning interval evaluation.

Workload	Insert	Read	Update
Uniform	4.6896%	1.0033%	6.9008%
Zipfian	-2.0187%	0.3350%	3.3505%

TABLE I
Relative Performance Improvement Ratio of Static Tuning Interval based on MongoDB

B. Static Tuning interval Behavior Analysis

For this section, we use 0.3G dataset and 0.1M operations with YCSB-A workload[2]. Fig 7 show standard deviation of HASS with static tuning interval and MongoDB. Gap of disk utilization of HASS is lower than MongoDB. In other words, HASS use disk utilization more fairly than MongoDB.

Figure 8 and figure 9 shows latency of HASS with static tuning interval. In perspective of performance, HASS is better than MongoDB in most case. Fine-grained split leads performance improvement. However, in zipfian workload, performance of HASS's insert operation is worse than one of MongoDB's insert operations. We guess that the reason of performance degradation is caused by size imbalance which is essential problem of static tuning interval. In zipfian workload, hot spot could be skewed in specific chunk. However, static tuning interval cannot find more appropriate split point and ignore size imbalance. Therefore, it could be main limitation of static tuning interval.

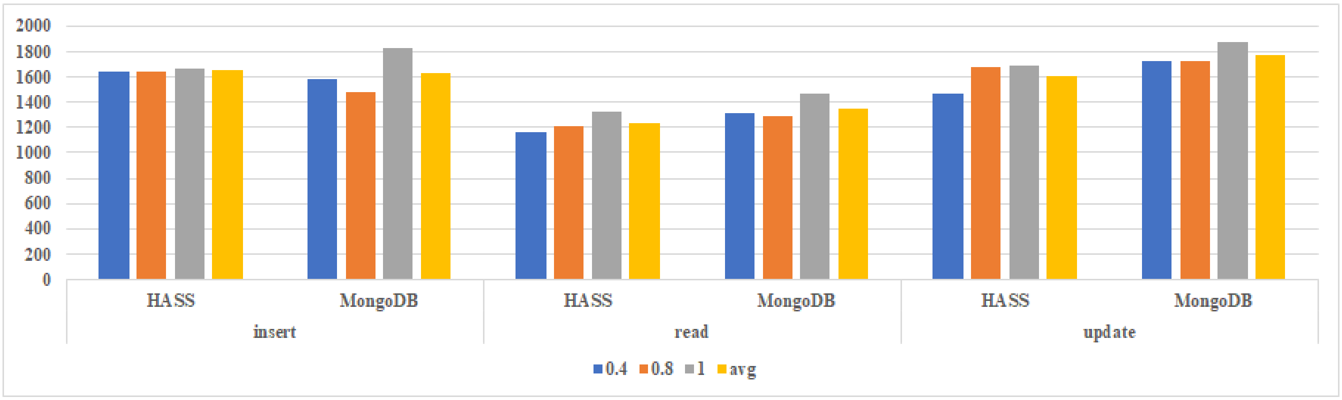


Fig. 5. Latency with HASS dynamic tuning in uniform workload

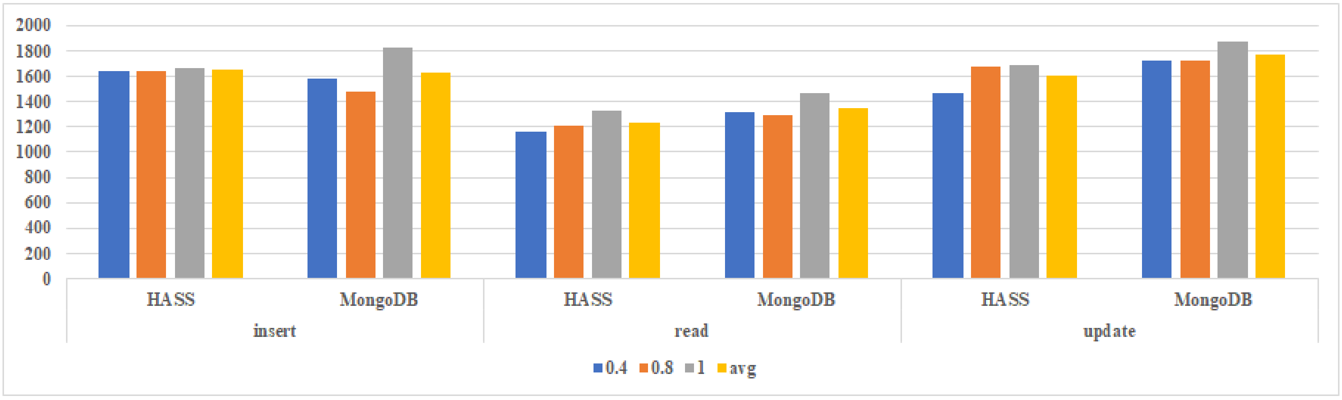


Fig. 6. Latency with HASS dynamic tuning in zipfian workload

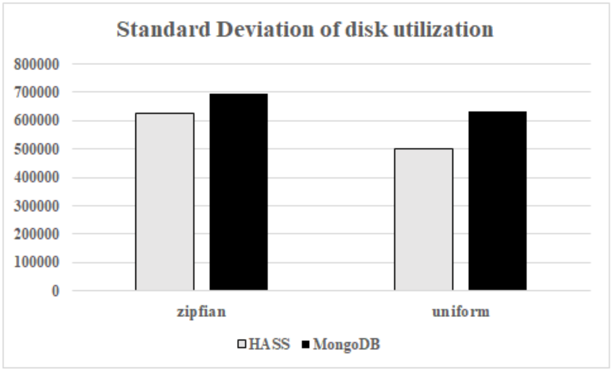


Fig. 7. HASS is fairer than MongoDB

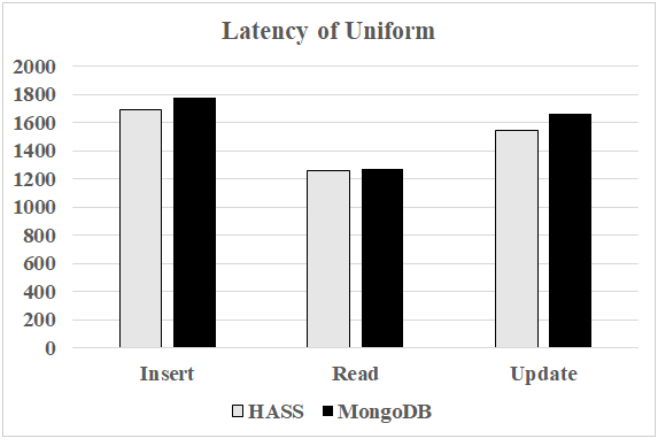


Fig. 8. Latency with static tuning interval in uniform workload

C. Dynamic Tuning interval Performance Evaluation

As shown in figure 5 and figure 6, Performance of HASS with dynamic tuning interval is better than on of conventional MongoDB. It proves limitation of static tuning interval and more fine-grained hotness awareness is effective. Especially for small dataset, HASS is highly well-performed compared to MongoDB. The reason why HASS with large dataset is degraded is more scattered hot data. It means that if dataset is largest, hot data is more distributed and limitation of static tuning interval could influence performance even if we use dynamic tuning interval. For scaling it, finding appropriate dynamic interval is necessary and we remains it as future works.

As shown in table II, HASS with dynamic tuning interval still tries to achieve more fairness compared to MongoDB.

V. RELATED WORKS

As described in previous section, there are some researches which considers "hot data"[6],[7] and give

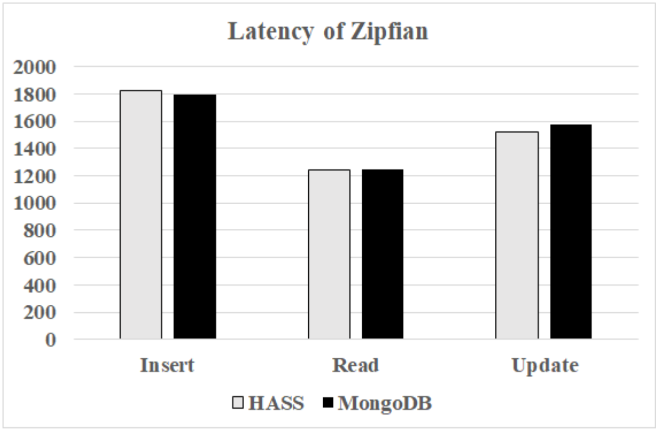


Fig. 9. Latency with static tuning interval in zipfian workload

	HASS	MongoDB
Standard Deviation	1538318	1641742

TABLE II
STANDARD DEVIATION OF DISK UTILIZATION BETWEEN SHARD CLUSTERS

solution for specific problems which they point out. Among them, Boonyalit[9] implement "tag" and it helps to improve performance in skewed workload. The "tag" will estimate hot spot and for balancing, tag will be adjusted followed by hot spot. If tag is set, position of data is also set. If item A is inserted and there is a tag which range includes A, A will be inserted to shard server where tag points. By using hotness aware tag, it can improve performance and utilization is more efficient. Difference with our works is approaches. Motivation is almost same. However, they use tag, not split. Therefore, this work can cause additional overhead to manage and store data related to tag. If split aware hotness of item, it doesn't cause additional overhead which can caused by tag.

VI. CONCLUSIONS

In this report, we introduced hotness-aware sharding split, HASS. Moreover, we designed 2 algorithm for effective and light-weight hotness awareness for chunk, static tuning interval and dynamic tuning interval. Static tuning interval still achieve fairness. However, it has its own limitation, size imbalance. For solving this problem, we designed and implemented dynamic tuning interval. By using it, we can not only get higher performance compared to MongoDB, but also make multiple shard fair. Even if dynamic tuning interval has to be more flexible, it is efficient algorithm in perspective of performance and fairness.

REFERENCES

[1] "Mongodb." [Online]. Available: https://www.mongodb.com/

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[3] "Rocksdb, a persistent key-value store for fast storage," 2016. [Online]. Available: http://rocksdb.org

[4] "Leveldb, a fast and lightweight key/value database library by google," 2005. [Online]. Available: https://github.com/google/leveldb

[5] "Choosing a good shard key in mongodb." [Online]. Available: https://www.kenwalger.com/blog/nosql/mongodb/choosing-good-shard-key-mongodb/

[6] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 739–752. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/li-yongkun

[7] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 363–375. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[8] T. Peng, M. Jiang, and M. Hu, "A dynamic clustering algorithm based on small data set," in *2009 Sixth International Conference on Computer Graphics, Imaging and Visualization*, 2009, pp. 410–413.

[9] B. Oonhawat and N. Nupairoj, "Hotspot management strategy for real-time log data in mongodb," in *2017 19th International Conference on Advanced Communication Technology (ICACT)*, 2017, pp. 221–227.