

Hotspot Management Strategy for Real-Time Log Data in MongoDB

Boonyalit Oonhawat, Natawut Nupairoj

Department of Computer Engineering, Chulalongkorn University, Bangkok, Thailand, 10330

boonyalit.o@student.chula.ac.th, natawut.n@chula.ac.th

Abstract— MongoDB is a Document based NoSQL that developed to answer the increasing demands for scalable data store in big data era. To achieve good performance, the system must be designed properly from the start. The performance issues like hotspot can be fatal to overall performance and various researches were focusing on applying load balancing techniques to resolve this problem. However, these techniques may not be effective for time series data, such as real-time system logs. In this paper, we propose a new data distribution algorithm based on tag aware sharding to minimize the effect of hotspot problem, especially the system with heavy writing requirements. This improves the overall performance and allow us to handle time series data effectively.

Keywords—Big Data; Database; NoSQL; MongoDB; Load Balance; Data Distribution; Sharding; Log Data; Hot Spot;

I. INTRODUCTION

In the age of information, data size is massive, fast growth rate and have a high variety. Various concepts such as big data and cloud computing are introduced to stored or process large amount of data. NoSQL or “Not only SQL” is a schema less database developed to answers the increasing in demand for system that scales. NoSQL featured Scalability, Availability and Fault Tolerance [1] which is important factors in distributed system environment. MongoDB [2] is one the most popular document-based NoSQL [3] because of its flexible data type, high functionality and widely support by many programming languages.

One type of data is log data, which is very significant for various computer fields such as security, performance management, and data mining. Due to its fast-growing nature, massive size, and usually come in real-time, this type of data can be quite difficult to handle especially the hotspot issues, which can cause performance bottleneck in distributed system environment. There are various researches in improving the performance of distributed storage system using load balancing or hotspot prevention. For example, Cruz [4] stated that each type of data workload [5] affect system performance differently and it is beneficial for system to match their configuration with data workload pattern in order to utilize system resources. To support this concept, MongoDB provides two data distribution method which are Range sharding and Hash sharding. Range method stores data according to their range number while Hash using hash function to compute hash key before randomly distribute data among server. Range sharding is better at read operation such as time range query since the data key are visible

and can be directly selected from the database. However, system using range method for real-time data can experience performance degradation dramatically since the range method concentrates new data insertion on only one server at a time. This could easily lead to hotspot problem during write operation. Hash sharding, on the other hand, can distributed workload evenly among servers, which results in high write performance. However, range query of time-series data is terrible due to the data key are not visible to query due to hash function and end up scanning entire database for each result. However, this problem can be solved by do indexing on the document field used in query. Despite this solution, which indeed can help improving read performance, it ends up suffer from high additional expense on disk space and memory because each write operation need to update index which result to additional operation cost and heavily affect performance especially under heavy write environment [10].

In system like log or time series data in particular, the amount of data and its growth rate can be increase dramatically overtime and high write performance like hash sharding is required while still need to retain range query capability. Both data distribution method still isn't exactly ideal due to mentioned drawbacks and how hotspot problem still affects overall performance. We discovered that system can gain solid improvement in performance by properly planning the distribution of data to match with log workload using available MongoDB feature. Our system managed to gain two positive result from experiments. First, system processes that affect performance such as data migration will be less likely to occur thus directly reduce the operation procedures and stress on server. Second, hotspot or heavy write environment effects on read performance is lessen or avoidable due to the carefully query plan and data placement based on log data access prediction can separate read operations away from hotspot area.

This research paper is organized as followed. In section II, we describe the general background of MongoDB and its mechanism. Our hotspot management strategy is discussed in section III. We present our experimental results in section IV. Section V concludes our paper.

II. BACKGROUND

A. Related Research

There are various researches focused on solving load balance problems in NoSQL. Z, Chen et al. [6] proposed load site aware strategy for data distribution, this method avoids site that has

heavy load by distributed data to lower load site. This group also proposed data re-allocation strategy based on hypergraph [7] to handle specific applications. Cruz et al. [4] proposed that different workloads require different system resources and introduced workload aware system to increase system elasticity by automatically reconfigure data site configuration with different workload patterns. Yahoo! [8] developed Load balancer to prevent their server from overload and become hotspot by calculate the heat or load of data unit called tablets and migrate hot tablets out from overload server. Dbalancer [9] is another load balancer that base on the detection of dynamic and skewed workloads.

Similar to our research, several researches also focused on the improvement of MongoDB. Yimeng [12] proposed the improvement of MongoDB auto-sharding mechanism based on frequently of data operations. Similarly, Xiaolin [13] also proposed an algorithm to improve auto balancing mechanism by using replica set to help reducing the rebalance costs. Han-Sheng et al. [11] developed Altila the load monitoring tool build on top of several NoSQL including MongoDB. Huang [14] developed auto-scaling system based on workload and concurrent user. Lin [15] introduced dynamic data balancing method by detected workload on chunk and shard.

B. MongoDB Introduction

MongoDB [2] is a high-performance document based NoSQL that stores data as document type similar to JSON. Document data are flexible and process dynamic schema suitable for most system. MongoDB also has various functions, which are necessary for big data applications such as parallel processing, map reduce, and data replication.

MongoDB uses Horizontal scaling or sharding to scale out the system. Data storage servers in MongoDB are called Shard and each Shard contain a database called collection which process a set of document call chunk. MongoDB also has its load balance mechanism called balancer to automatically distributed chunk evenly among shard by detect the different number of chunk that each server hold before migrate chunk from the highest populated server to lowest.

C. MongoDB Sharding Mechanisms

Sharding needs to be enabled in order to allow MongoDB to start distributing data. This process has to be initiating before system deployment and receiving first data. The first step requires shard key to be selected from the document field which will be used as a key and splitting new chunks. The higher cardinality of shard key will ensure the number of chunk to be split. The range of shard key will act as an identification for each chunk. Sharding will be fully enabled once issue shard collection command in which strategies will depend on the assigned value for the selected shard key in which "1" for range and "hashed" for hash. Sharding procedure will start to become different after we issued shard key command in which will be explain further.

1) Range Sharding: Shard server will start splitting first chunk once it received first data, this first chunk will act as a receiver for recent new data. The chunk will be splitting into

new chunk once it reaches chunk maximum setting size. The process will keep on in insert into first shard until its detect uneven number of chunk between all shards before begin auto balancing process and migrate chunks into another shard until the numbers become equal. The chunk auto balancer threshold will increase once it reaches certain number. This results in the migration become less frequent after certain amount of time.

The insert behaviour for range sharding can be explained in Figure 1. There will always be a chunk with max id as a shard key. This chunk acts as a receiver chunk for newest incoming documents. The document will continue to be inserted into this chunk unless there is a chunk that possess range that cover incoming document's key for example unless the chunk which range is 1000 as lower key and 4000 as upper key, new document will continue to insert into chunk with maxid until reach maximum chunk capacity and start splitting into two new chunk.

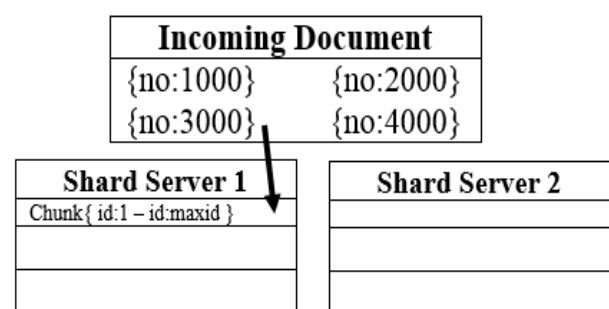


Figure 1. Range sharding data distribution

2) Hash Sharding: Shard server immediately creates set of chunks on every shards with key range based on hash value. Selected key field in incoming document will get computed into hash value and assigned into the chunk that stores chunk with that hash key range.

As can be seen in Figure 2, each shard server holds equal number of chunks with hash key as id instead of range number. Every new document will pass through hash function and got assigned into chunk base on computed hash value.

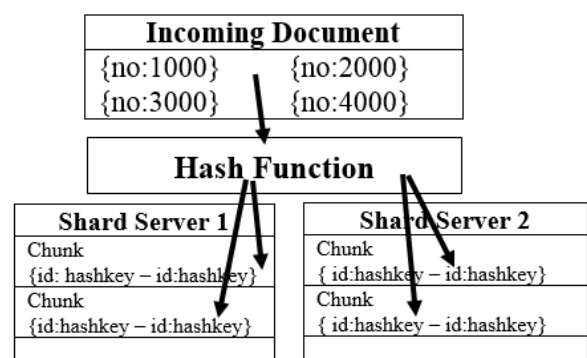


Figure 2. Hash sharding data distribution

D. Log Operations and Sharding

Log operation traffic can be varied by time depending on said time incoming clients' requests. If the load is not

distributed properly, it might end up become hotspot during the high write period, especially if data is concentrated on single shard server. Hotspot can be fatal for systems since it consumes high amount of system resource which might end up cause dramatically drop in performance once system reach its limit capability [15]. In order to get better understanding on effect of incoming workload and how it can lead to hotspot, we will need to find the differences in performance between each method in handle large amount of data.

After storing log document into server, range query is required in order to get information out of storing data. The amount of return document per one query will be larger the wider the range are which will end up took longer time for shard that hold a wider area of result. Since MongoDB can parallel processing the information, it is more ideal to distribute data equally among servers. On the other hand, the query processing speed does not guarantee to become equal among shards due to the different in CPU consumption especially if under heavy writing where insert performance take a large portion of CPU resource. Both log operations in each scenario will be further explain using experimental result in part 4 performance evaluation.

III. HOTSPOT MANAGEMENT USING TAG AWARE SHARDING

A. Tag Aware Sharding

Selecting sharding mechanism for storing log operation traffic is not straight-forward. Range sharding matches quite naturally to log query patterns of which time range is the main query criteria. However, range sharding can lead to the hotspot problem, especially when chunk size is large. Hash sharding, on the other hand, does not suffer from the hotspot problem, but it may not perform well for range query operations. To overcome these problem, we propose hotspot management using Tag Aware Sharding. This sharding mechanism normally uses for improving data locality by grouping chunk with specific shard according to their information. We can tag a specific range of shard key or existing chunk range and assign similar tag to a shard. The chunk range that received tag will automatically migrate to correspond shard and if specific chunk range didn't exist before tag, the system will automatically create new empty chunk on shard that hold recent chunk before migrating it to assigned shard. Since the chunk has been created and migrated from the start just like hash sharding, this makes the insert performance to be similar to hash sharding. It should be noted that we will have to manage the number of newly-created chunks since this migration takes a long time to complete if the number of newly-created chunks are too high.

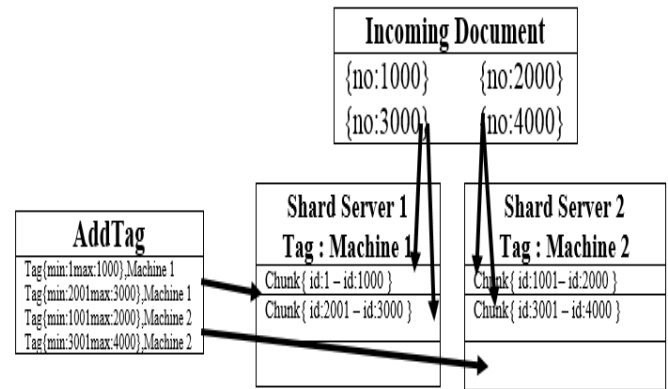


Figure 3. Sharding data distribution with Tag

To apply tag aware sharding with log data, we use timestamp as a shard key in order to match the nature of log query patterns and tag names that are corresponding to the sharding servers. As shown in Figure 3, the range of timestamps is carefully calculated such that the size of data in each chunk is not too big or too small. This can prevent the hotspot problem while still ensure good range query performance. By assigning with a proper tag, each chunk will be placed on a specific sharding server. To make full use of parallel processing and lessen the hotspot problem, our proposed algorithm distributes chunks evenly among the sharding servers. In addition, we reduce the overheads of chunk creation by pre-creating new chunks before the data in that time ranges arrived. The number of newly-created chunks is calculated to avoid the overheads of migration and presplitting process.

B. Shard Optimization Based on Log Operation

According to what we learn from both range and hash sharding behaviours on log data operations, our purposed algorithm determines the range of each tag first in order to ensure that workload distribution will be equally distributed among multiple shard. The goals of tag range determination include matching data distribution with query pattern to make full use of MongoDB parallel capability and lessen the chances for query to hit on shard that become hotspot. The range of timestamps assigned to each chunk will affect the size of that chunk. To ensure the proper chunk size, we will use document insertion rate in determine the range of timestamps. Note that, each chunk can only hold up to 25,000 document per chunk [6] regardless of its size. Otherwise, chunk will become unmovable. Thus, we will use 25,000 as the upper limit of number of logs to be placed in each chunk. The algorithm to determine the range of timestamps is presented in Algorithm 1.

Algorithm 1 Tag Range Determination Algorithm

1. **let** q be the expected range of query window (in seconds)
2. **let** D be the expected log arrival rate (in log document per second)
3. **Formula** Tag range $T_r = T + t$
4.
$$t = \begin{cases} q & \text{if } (q * D \leq 250000) \\ \frac{250000}{D} & \text{otherwise} \end{cases}$$
5. **return** t

Tag Range Determination calculates the range of the timestamps to be assigned to a chunk (t). The main idea is try to fit documents within the query window into a single chunk. To calculate the range, two parameters, the expected range of query window (q) and the expected log arrival rate (D), are considered. We determine time range t in line 4 by multiplying expected range query window with expected log arrival rate. This allows us to define the chunk size to match the typical query range and the expected incoming rate. If the incoming rate is too high, we limit our chunk size to not exceed 250000 documents. For example, suppose the expected range of query window is 15 minutes (or 900 seconds). If the arrival rate is 100 log document per second, number of log within the query window of 15 minutes are 90,000 documents, which is below 25,000 documents. In this case, we will assign 15 minutes as the range of this chunk. However, if the arrival rate is 1,000 log document per second, there will be 900,000 documents within 15 minutes. This will exceed 250,000 documents. In this case, we will assign 250,000 divide by 1,000 result in the range of 250 seconds or approximately 4 minutes.

To improve the performance, we include a mechanism called pre-assigned chunk creation in our proposed solution. In this mechanism, the determination of tag ranges for new chunks will be calculated in bulk. Let $N_{precalc}$ be the number of new chunks to be pre-calculated. Once calculated, these new chunks will be assigned and scattered among shard servers to fully utilize parallel capability. When the number of remaining chunks has reached the pre-defined remaining threshold or $N_{threshold}$, new tag ranges will be calculated, assigned, and distributed. This allows MongoDB to have enough time to generate new chunks and migrate them to correspond shards.

There are still couple issues with the respect to the arrival rates. In reality, the arrival rates of log data are not fixed and constantly changes. For example, the system activities during daytime usually generate more log data than the nighttime. Since the Tag Range Determination Algorithm bases on the expected arrival rates, changes in arrival rates will affect data distribution plan and overall performance. The sensitivity of our proposed solution to react to the changes could be adjusted by using smaller $N_{precalc}$ and smaller $N_{threshold}$. However, these numbers should not be too small as the system will react too

frequent which leads to performance degradation due to frequent data migration. Others issue is the impacts of using inaccurate arrival rates which can lead to load imbalance. We will leave these issues as our future works.

IV. PERFORMANCE EXPERIMENT AND EVALUATION

We setup our experimental environment based on MongoDB recommending setup which consisted of 4 shard servers. The CPUs in all shard servers are Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz. The experimental log data is generated by using python code and the number of test documents are approximately 17 million records which come from the IT log of Chulalongkorn University. The log are regenerated at the arrival rate of 200 documents per second. Unless stated otherwise, all experiments will be based on data in the span of 1 day period.

A. Insert Performance

We evaluate the insert performance by measuring the insert limitation for each sharding server by continuously inserting 1,000,000 record of document into the system and computing average processing time to see how much time needed for each sharding mechanism under the same amount of documents. The results show the elapsed time until each sharding mechanism can finished inserting all document into the system.

Figure 4 presented the comparison of insertion performance between Range sharding, Hash sharding, and our proposed tag-aware sharding. As expected, Range sharding performed the worst. This is due to the hotspot problem, which caused the hotspot sharding server to split and migrate chunks continuously. This incurred high overhead and made the hotspot shard to become the bottleneck. Hash sharding performs better as the insertion loads are distributed among sharding server. The results also clearly showed that our proposed solution outperformed other sharding mechanisms. Although our tag-aware sharding behaved like mini Range sharding, pre-assigned chunk creation with proper chunk size management can reduce the impacts of hotspot problem. Incoming data were not delivered to only single sharding server. This helped distribute the workloads. In addition, pre-assigned chunk creation minimized the overheads of chunk splitting and data migration mechanism which usually incurred lots of overheads.

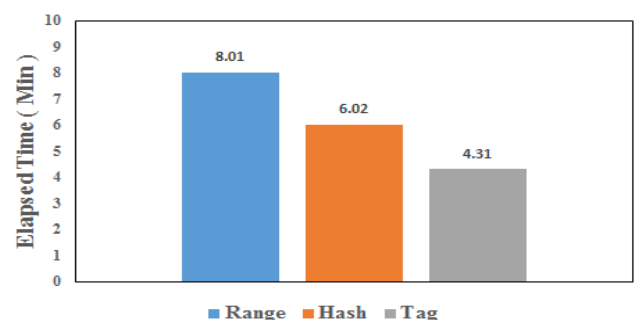


Figure 4. Sharding Insert Performance

B. Range and Hash Sharding Query Performance

To demonstrate the query performance in real-world situation, we performed our experiments based on three different query scenarios, including 'no write', 'normal write', and 'heavy write'. In 'no write' scenario, the experiment will have only query operations. No new data was inserted during the experiment. CPU of each sharding server provided services for the query operations only. The 'normal write' scenario included both query and insert operations. New data were inserted at the moderate rate, which in our case is 1000 document per second. In 'heavy write' scenario, new data were inserted at the very high rate, which was 10000 document per second. This would cause hotspot and CPU capacities were very limit for query operation. To make our experiment reflect reality, we created index for Hash sharding to allow the best query performance for this mechanism. Furthermore, we randomly selected 10 of 5-minute and 2-hour query ranges from within 1-day period which will return 60000 and 1440000 documents respectively and calculated average query time.

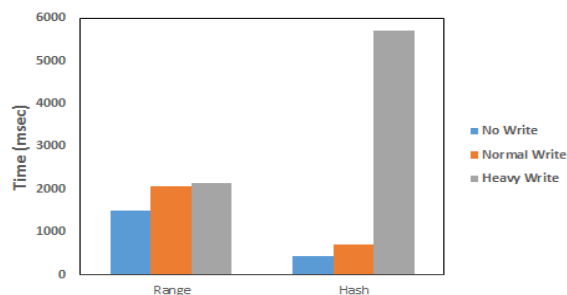


Figure 5. Range and hash 5 minute query performance

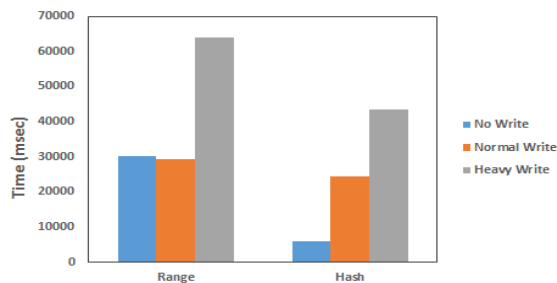


Figure 6. Range and hash 2 hour query performance

Figure 5 and 6 showed the results of query performance for the query ranges of 15-minute and 2-hour, respectively. Based on the results, Hash sharding query performance clearly performs better than Range sharding in 'no write' and 'normal write' scenarios. This is due to the nature of Hash sharding of which query workloads were shared equally among sharding servers while Range sharding was much more imbalance. However, both strategies suffered heavy performance degradation in 'heavy write' scenario. The hotspot effects become more visible in hash sharding strategy of which query operations were performed on every shards and make hotspot area become unavoidable. This greatly affected query performance. This issue also greatly affected performance

when the number of return documents of target shard are high as can be seen in range performance under heavy write in Figure 6. To further study this problem, we captured the overall CPU Utilization pattern during the experiments and depicted in Figure 7 and Figure 8 for Range sharding and Hash sharding, respectively.

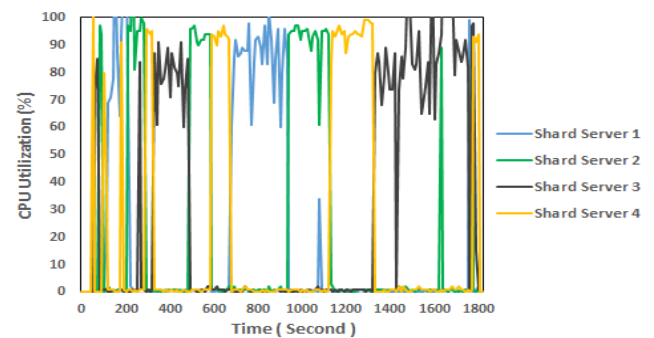


Figure 7. Range Sharding CPU Utilization

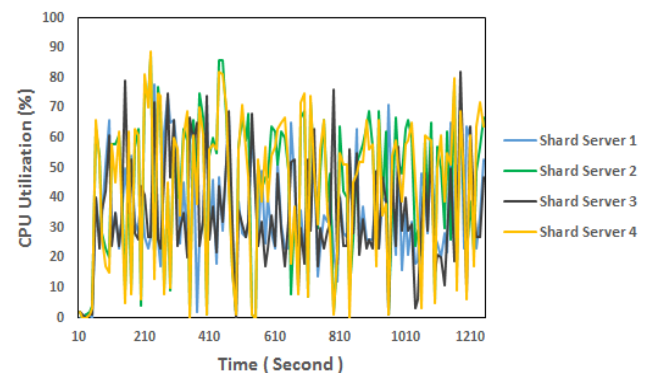


Figure 8. Hash Sharding CPU Utilization

The CPU utilization clearly explained the hotspot problem. For Range sharding, the CPU utilization of all sharding servers were quite swing and unpredictable. CPU utilization were usually reached 80-90%. Hash sharding had moderately CPU utilization at 40-60% but happened on every shards. There are several factors that can clearly explain the above result. First, MongoDB chunk splitting and migration has several procedures which directly affect each insert operations which end up demand a lot of CPU especially on Range sharding where it required a lot of splitting and migration process. Second, the hash sharding required less CPU is due to how it can identify which shard to store the document. Since the pre-created hash chunk are already cover all hash key range and distributed evenly among shard.

C. Tag Sharding Performance

In this experiment, we performed experiments on our proposed tag sharding strategy using the same experiment with range and hash strategies. The result for tag sharding performance can be seen compare with both default strategy in Figure 9 and 10

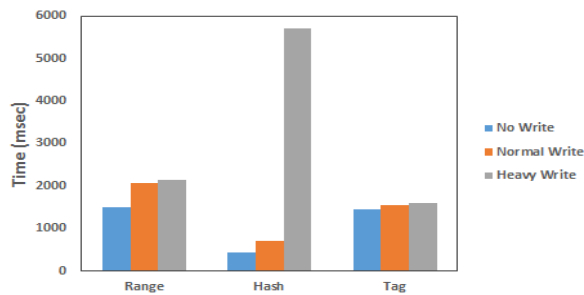


Figure 9. Tag 5 minute query performance

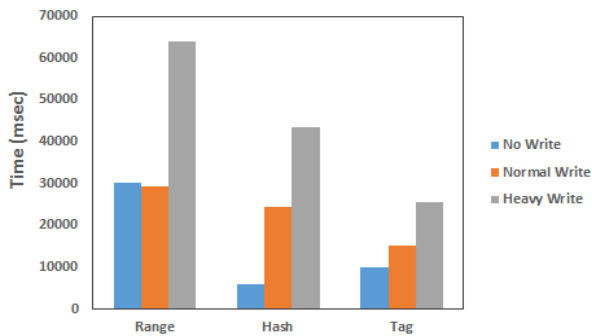


Figure 10. Tag 2 hour query performance

The average query performance for tag strategy were slower than hash during 'no write' scenario. This is due to the fact that hash sharding could distribute the workload more equally. Our proposed solution performed similarly to Range sharding in 5-minute runs. The query time for our proposed solution slightly increased comparing to other two strategies in 'heavy write' scenario. This proved that the impacts of hotspot sharding were lessen by a good data organized and distribution thus gaining better performance than other strategies especially the large query range under 'heavy write' scenario.

The CPU utilization of our proposed solution can be seen in Figure 11. Unlike range sharding where hotspot period is hard to predict, our proposed solution allowed all sharding servers took turn to receive data. The overheads of chunk creation and migration were also minimized with our pre-assigned chunk creation.

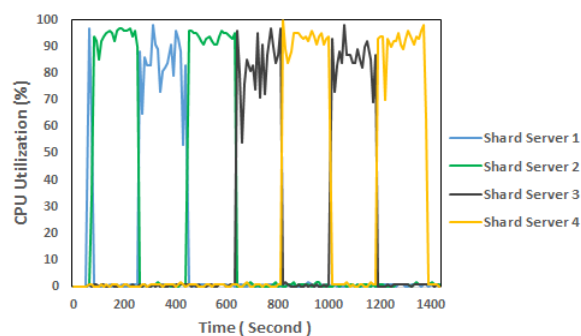


Figure 11. Tag Sharding CPU Utilization

V. CONCLUSIONS

In this paper, we introduced an efficient way to utilize MongoDB tag aware sharding feature for log system. This feature allows the system to become flexible even under the situation where hotspot occur frequently by plan and organized data distribution within multiple shard server. The performance is slightly better due to hotspot shard are less likely to be hit during query compare to other sharding strategy where query easily affect by hotspot.

There are several ideas that can be studied in our future works. First, we will need to study the impacts of miss-predicting expected throughputs, especially in the system with the workload with frequently changes in patterns which will lead to an uneven distribution of data and several performance issues. Second, each shard server capability in handle workload can be detected and utilize based on its specification if shard server specification in system are not homogenous.

REFERENCES

- [1] J. Han, E. Haihong, G. Le and J. Du "Survey on NoSQL database." Pervasive computing and applications (ICPCA), 2011 6th international conference on. IEEE, 2011. p. 363-366.
- [2] MongoDB <http://www.mongodb.org/>
- [3] DB-Engines <http://db-engines.com/en/ranking>
- [4] F. Cruz et al. "Met: workload aware elasticity for NoSQL" Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013. p. 183-196.
- [5] B. F. Cooper et al. "Benchmarking cloud serving systems with YCSB" Proc. of the 1st ACM symposium on Cloud computing, ACM New York, NY, USA 2010, pp. 143-154
- [6] Z. Chen et al. "Load-Aware Fragment Allocation Strategy for NoSQL Database" Proceedings of International Conference on Internet Multimedia Computing and Service. ACM, 2014. p. 416.
- [7] Z. Chen et al. "A new fragment re-allocation strategy for NoSQL database systems" Frontiers of Computer Science, 2015, 9.1: 111-127.
- [8] M. Klems et al. "The Yahoo!: cloud datastore load balancer." Proceedings of the fourth international workshop on Cloud data management. ACM, 2012. p. 33-40.
- [9] I. Konstantinou, D. Tsoumakos and I. Mytilinis "DBalancer: distributed load balancing for NoSQL data-stores" SIGMOD'13 Proc. of the 2013 ACM SIGMOD International Conference on Management of Data, New York, USA, 2013, pp. 1037-1040
- [10] MongoDB limitation and thresholds <https://docs.mongodb.com/manual/reference/limits/>
- [11] H. Huang, S. Hung and C. Yeh, "Load balancing for hybrid NoSQL database management systems" Proc. of the 2015 Conference on research in adaptive and convergent systems New York, USA, 2015, pp. 80-85
- [12] Y. Liu, Y. Wang and Y. Jin "Research on the improvement of MongoDB Auto-Sharding in cloud environment" Computer Science & Education (ICCSE), 2012 7th International Conference on. IEEE, 2012. p. 851-854.
- [13] X. Wang, H. Chen and Z. Wang "Research on improvement of dynamic load balancing in MongoDB." Dependable, Autonomic and Secure Computing (DASC), 2013 IEEE 11th International Conference on. IEEE, 2013. p. 124-130.
- [14] CW. Huang et al. "The improvement of auto-scaling mechanism for distributed database-A case study for MongoDB." Network Operations and Management Symposium (APNOMS), 2013 (pp. 1-3).
- [15] S. Lin, H. Chen and F. Hu "A Workload-Driven Approach to Dynamic Data Balancing in MongoDB." In: 2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity). IEEE, 2015. p. 786-791.



Boonyalit Oonhawatt graduated from Assumption University, Thailand in 2011, Bachelor degree in Business Administration and Major in Business Information System. Currently study at Chulalongkorn University Computer Engineering department. He also experienced in several computer field such as web development but currently focus on researching cloud computing, distributed system and database.



Natawut Nupairoj received the B.E. in Computer Engineering from Chulalongkorn University in 1990, M.S. and Ph.D. in Computer Science from Michigan State University in 1993 and 1998, respectively. Since 1998, he has been a Lecturer at Department of Computer Engineering Chulalongkorn University. His research interests include grid computing, distributed system, and service oriented architecture.