An Approach to Improve Load Balancing in Distributed Storage Systems for NoSQL Databases: MongoDB

Sudhakar and Shivendra Kumar Pandey

Abstract The ongoing process of heterogeneous data generation needs a better NoSQL database system to accommodate it. NoSQL database stores data in the distributed manner in their globally deployed shards. The data stored in these databases should have high availability, and the system should not compromise with the scalability and partition tolerance. The distributed storage systems have the main challenge to address the skewness in the data. The process of distribution of data items over the nodes in the system causes skewness of data. To address this problem, we propose a different approach to balance load in the distributed environment is the partitioning of data into small chunks that can be relocated independently.

Keywords NoSQL · Data load balancing · MongoDB · Chunk migration Big data

1 Introduction

Recent developments in size of data have heightened the need for storing world digital data exceeds the limit of a zettabyte (i.e., 10^{21} bytes); it is a challenge as well as necessity to develop a powerful and efficient system that has the capacity to accommodate data. For example, a system using a very dense storage medium like deoxyribonucleic acid (DNA). DNA can encode two bits per nucleotide (NT) or 455 exabytes per gram of single-stranded DNA [1]. Taken into account the fact that

Sudhakar (✉)
Indian Computer Emergency Response Team, Ministry of Electronics
& Information Technology, New Delhi, India
e-mail: sudhak82_scs@jnu.ac.in

Sudhakar · S. K. Pandey
School of Computer & Systems Sciences, Jawaharlal Nehru University,
New Delhi, India
e-mail: shivaiert@gmail.com

© Springer Nature Singapore Pte Ltd. 2018

P. K. Pattnaik et al. (eds.), *Progress in Computing, Analytics and Networking*,
Advances in Intelligent Systems and Computing 710,
https://doi.org/10.1007/978-981-10-7871-2_51

ton of genetic material is prerequisite by DNA as to store zettabyte of information. The argument can simply be made that data storage needs to be distributed for quantitative reasons alone.

For this, the distributed storage systems need to be available and scalable when needed. This amount of storage will only be possible if it is distributed geographically and should exhibit the property of being accessible by millions of users besides its raw storage capacity.

Web giants like Google, Amazon, Facebook, and LinkedIn are the industries that use distributed storage systems. To fulfill their requirements, they have deployed thousands of data centers globally so that they can make data available all the time with scalability at any level. Furthermore, in the case of failure that occurs in scaling process is either the failure of the software or the hardware components. Therefore, these failures need to be handled during the planning and implementation phases.

In addition to this, the traditional distributed large-scale storage systems are inefficient to store the enormous heterogeneous data (structured, semi-structured, or unstructured) [2, 3] as they do not satisfy the 7 Vs (volume, variety, velocity, veracity, validity, volatility, and value) [4]. The way it is accessed and stored in traditional databases may pose many limitations. These new horizons of the data bring *big data* [5] into reality. Therefore, we require more powerful and efficient solutions to process big data.

Big data requires a new kind of database system to handle heterogeneous data sets. One of the most popular and well-known databases for big data is NoSQL (Not Only SQL) that has the capability to process big data for mining valued information. As a result, many industries have developed various NoSQL databases depending on their requirements such as Facebook's Cassandra [3], Amazon's Dynamo [6], Yahoo! Pnuts [7], Google's BigTable [8] or Riak, and MongoDB [9]. These systems scale very well over the trade-offs of consistency, availability, and partition tolerance as mentioned in the CAP theorem [10]. Each one of them is using the shards to store their data. Since deploying shards globally is the main challenge for big data to handle as it suffers from load balancing problem.

The load balancing techniques used in NoSQL systems do not consider chunk migration as performance indicator rather they prefer high availability and partition tolerance as their key indicator. In this work, we aim to bridge this gap by proposing an improvement over existing load balancing techniques by taking into account shard utilization and the migration of chunks to increase the efficiency of NoSQL databases, considering the particular NoSQL, e.g., MongoDB. Here, in this work, we have proposed a new approach to improve load balancing for MongoDB and compared our results with automatic load balancing algorithm of MongoDB, which clearly shows the better performance of our approach without affecting memory utilization of the shards. The main contribution of this article is:

- A naive and holistic approach is proposed for load balancing for MongoDB based NoSQL systems.

- Our proposed method is computationally cost-effective as the number of chunk migrations among shards are less compared to the traditional load balancing algorithms for MongoDB.
- Consistent performance is achieved by the proposed method irrespective of number and size of the shards.

The rest of the article is organized as follows: Sect. 2 comprises literature survey of NoSQL. Section 3 explains the maintenance and structure of the data being stored in the shards in MongoDB. In Sect. 4, we present our proposed approach for load balancing in MongoDB. Section 4.3 presents experiments and evaluation. Finally, Sect. 5 concludes the paper with some of the future research directions.

2 Related Work

Facebook's Cassandra [3, 11] is a distributed database designed to store structured data in a key-value pair and indexed by a *key*. Cassandra is highly scalable from both perspectives; one is that of storage and the second is that of request throughput while preventing from single point failure. Additionally, Cassandra's store's data in the form of *tables* that is very similar to the distributed multi-dimensional maps is also indexed by a *key*. It belongs to the column family like BigTable [8]. In a single row key, it provides atomic per-replica operation. In Cassandra, *consistent hashing* [12] is used for the notion of data partitioning to fulfill the purpose of mapping keys to nodes in a similar manner like Chord distributed hash table (DHT). Partitioned data is stored in a Cassandra cluster that would contain the moving nodes on the ring. To facilitate the load balancing, it uses DHT on its *keys*.

Amazon's Dynamo [6] is distributed key-value store database. It mainly focuses on scalability and availability rather than consistency. To address the problem of non-uniformity of node distribution in the ring, it uses the concept of virtual nodes (vnodes). Also, it follows a different strategy for partition-to-vnode assignment which results into the better distribution of load across vnodes and therefore over the physical nodes.

Scatter [13] unlike Amazon's Dynamo, it is a distributed *consistent* key-value store database, which is highly decentralized. For data storing, it uses uniform key distribution through consistent hashing typical for DHTs. Scatter uses two policies for load balancing. In the first policy, the newly joined node in the system is directed to randomly sample *k* groups, and then it joins the one handling a large number of operations. In the second policy, based on load distribution, neighboring groups can trade-off their responsibility ranges.

MongoDB [9, 14] is schema-free document-oriented database written in C++. MongoDB uses replication to provide data availability and sharding to provide partition tolerance to manage data across the distributed environment. It stores data in the form of chunks. To manage even distribution of chunks across all the servers in the cluster, a balancer is used in this system. Whenever balancer detects an

uneven chunk count event (i.e., chunk difference between minimal loaded and maximal loaded shards is greater than or equal to 8), it redistributes the chunks among shards until the load difference between any two shards is less than or equal to two [15].

3 Maintenance of Load in MongoDB

The basic concept of automatic load balancing of MongoDB is breaking up the larger collections into smaller chunks and distribute evenly over all available shards so that each subset of the data set belongs to one shard. The criteria of partitioning the collection in the database of MongoDB are that specify a shard key pattern for the chunk in two more parameters, *minkey* (minimum size of the chunk) and *maxkey* (maximum size of the chunk) [14]. We can say that chunks can have three attributes of the collection (i.e., *minkey*, *maxkey,* and *shardkey*). When chunk size reaches to a maximum size (i.e., 200 MB as configured for automatic balancing) then the *splitter* splits the chunk into two new equal chunks.

As already mentioned, the basic idea of automatic load balancing of MongoDB is that if the difference between the number of chunks of any two shards is greater than or equal to eight as detected by balancer of MongoDB, then balancer consider it as imbalanced shards and starts migrating chunks to other shards until the difference will decrease to two [15].

As discussed CAP theorem in Sect. 1, MongoDB embraces the properties of availability and partition tolerance. Hence, to ensure the availability, it uses replicated servers with automatic failover. In case of any failure in the system, partition tolerance ensures smooth functioning by allowing it to continue work as a whole. In the imbalance condition, the chunk migrations among the shards are managed according to the automatic load balancing algorithm of the MongoDB. To implement availability and partition tolerance in distributed systems, replication and sharding are commonly used which are briefly discussed below:

3.1 Replication

Replication is the process of synchronizing data across multiple servers connected in a distributed manner. It provides redundancy and increases data availability by making multiple copies of data on different database servers. In this way, data will be protected from the single point failure and loss of one server will not affect the availability of the data because the data can be recovered from the other copy of the replica set. The replica is $p_j^r \in \mathcal{S}_i$ where, r represents the replica of a particular node $\mathcal{S}_i = \{p_1^r, p_2^r, p_3^r\}$ where, $\mathcal{S}_i \epsilon S, S \in \mathbb{Z}^+$ and $1 \leq j \leq 3$. It should be noted that one

node has only three replicas in which one must be a *primary* replica and others are *secondary* replicas.

Every node has replication group G of size n, so the quorum is a subset of nodes in a replication group G. Let view v is a tuple over G that defines important information for G and view id is denoted as $i_v \in \mathbb{N}$ [16].

In some cases, replication can be used to service more read operations on the database. To increase the availability of data for distributed applications, we can also use different data centers to store the database geographically. Replica sets have the same data in the replica group. In this group, one replica is *primary,* and rest are *secondary* [14]. The *primary* accepts all the read/write operations from the clients. In the case, if *primary* is unavailable, one of the *secondary* replicas is elected as *primary*. For the election purpose, Paxos algorithm is used [17].

3.2 Sharding

MongoDB scales the system, when it needs to store data more than the capacity of a single server (or shard) with the help of horizontal scaling. The principle of horizontal scaling is to partition data by *rows* rather than splitting data into *columns* (e.g., normalization and vertical partitioning do in the relational database) [14]. In MongoDB, horizontal scaling is done by automatic sharding architecture to distribute data across thousands of nodes. Moreover, sharding occurs on a per collection basis; it did not take into consideration of the whole database. MongoDB is configured in such a way that it automatically detects which collection is growing monotonically than the other. That collection has become the subject of sharding while others may still reside on the single server. Some components that need an explanation to understand the architecture of the MongoDB sharding is given in Fig. 1.

- ***Shards*** are the servers that store data (each run *mongod* process) and ensure availability and automatic failover, each shard comprising a replica set.
- ***Config Servers*** "store the cluster's metadata," which include the basic information of chunks and shards. These chunks are contiguous ranges of data from collections that are ordered by the shard key.
- ***Routing Services*** are run *mongos* processes performing read and write request on behalf of client applications.

Auto-sharding in MongoDB provides some necessary functionality without requiring large or powerful machines [15].

1. Automatic balancing if changes occur in load and data distribution.
2. Ease of adding new machines without downtime.
3. No solitary point of failure.
4. Ability to recover from failure automatically.

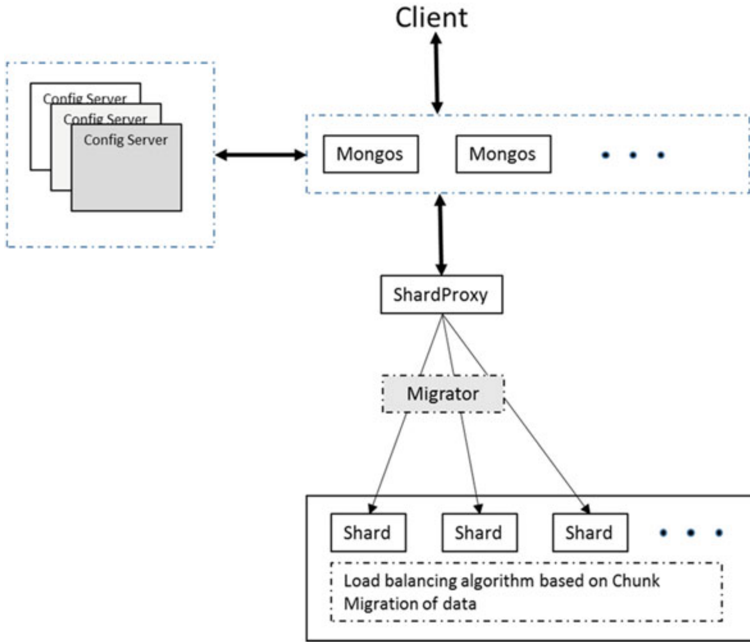


Fig. 1 Modified MongoDB distributed architecture [15]

In the system, S is the set of shards and each shard consists of $\mathcal{S}_i$ replica sets then we can state it as $S = \left\{ \mathcal{S}_i : \mathcal{S}_i \in S \, where \, p_j^{r_i} \in \mathcal{S}_i, i \geq 2, 1 \leq j \leq 3 \right\}$, here $p_j^{r_i}$ is represent j replicas available in ith shards.

4 Proposed Method for Load Balancing

4.1 Basic Idea About Load Balancing and Preliminaries

In this section, we are going to introduce all the terms that we will use in this document. For every data item $d \in D$, where D is the set of all data items, we define all types of load here, that represented by Γ. The *load* $l_t : D \rightarrow \mathbb{R}$, $t \in \Gamma$ is a function that use for assigning the associated load value of load type t from set D. There exists an associated load value for every unit of replication $U \in D$, i.e., $l_t^U = \sum_{v \epsilon \mathrm{U}} l_t(v)$. And any node H in the distributed system, at a particular node U_H contain all units of replication, and an associated value $l_t^H = \sum_{U \in U_H} l_t^U$. Every node has capacity $c_t^H \in \mathbb{R}$ for each load type t. Thus, the inequality $l_t^H < c_t^H$ must be maintained as invariant, as the violation would result in failure of H. We also calculate the *utilization* [18] of a node $u_t^H = l_t^H / c_t^H$ at $t \in \Gamma$. If a system has

utilization μ_t^S where S is a set of all nodes in the system and average utilization $\hat{\mu}_t^S$ of $t \in \Gamma$ [18], given as

$$\mu_t^S = \frac{\sum_{H \in S} l_t^H}{\sum_{H \in S} c_t^H}, \quad \hat{\mu}_t^S = \frac{1}{|S|} \sum_{H \epsilon S} u_t^H \tag{i}$$

In addition to the above, we need to consider the third parameter that represents the *cost* of moving replication unit or data item U from one host to another host in S, i.e., $\rho U: S \times S \rightarrow \mathbb{R}$, this parameter is linearly depends on l_{size}^U. If the system is considered uniform, then $H, H', H'' \in S$ and $U \in H$ such that $\rho U(H, H') = \rho U(H, H'')$ is seems to be a constant where $\rho U \in \mathbb{R}$.

Let

$$L_S^H = \{l_t^H | t \in \Gamma, U \in H\} \tag{ii}$$

$$C_H = \{c_t^H | t \in \Gamma\} \tag{iii}$$

$$L_H = \{l_t^H | t \in \Gamma\} \tag{iv}$$

$$L_S = \{(C_H, L_H, L_S^H) | H \in S\} \tag{v}$$

where L_S is referred as the system statistics for S. The migration is done by the balancer, i.e., $MIGRATOR(U, H)$ to $MIGRATOR(U, H')$ where U for a unit of replication and H, H' are the hosts.

4.2 Modified Load Balancing Algorithm

In the algorithm, there are two methods one is *IsBalance*() that will return a Boolean value depending on whether the particular *shard* or *host H* is balanced or not. However, the condition of balance is as follows: if a shard has more data than its threshold value, then it will become imbalanced. The threshold value is $Const * c_t^H$ (e.g., $Const = 0.7$ or 0.8 or 0.9) where c_t^H the capacity of the *shard H*. If a shard shows that it is imbalanced, then it needs to migrate. The second method is *MIGRATOR*() which migrates data from imbalanced shard to a balanced shard until $\hat{l}_t^S$ data remains in the shard.

We are calculating the total data occupied in all shards that are l_t^S and then calculate average data occupied by all shards that is $\hat{l}_t^S$.

Furthermore, we check if the shard is balanced or not, and for each value of imbalance shard, we migrate chunks from imbalance shard to balance shard until

the condition $\left(l_t^{H_t^i} \geq \hat{l}_t^S \ \textbf{OR}\ l_t^{H_t^j} \geq l_{t_{max}}^{H_t}\right)$ met and this process repeated until all shards become balance.

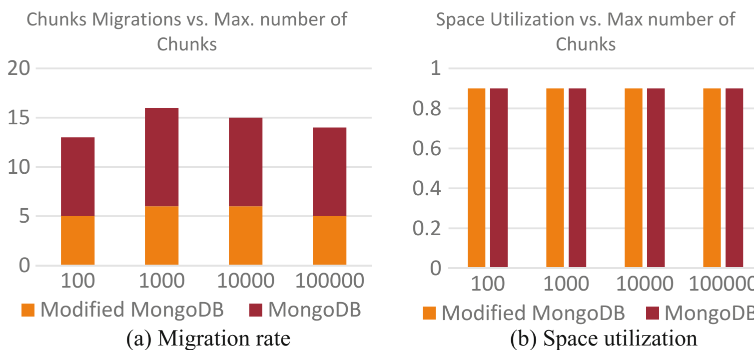
4.3 *Experimental Results*

To show the effectiveness of modified MongoDB, we have compared our approach with traditional MongoDB and compared the two on the basis of chunk migration rate and space utilization metrics.

To prove our claim, we have performed the experiments four times with a different number of chunks and noted the results. In the first experiment, we have considered shards with 100 chunk capacity. Whenever imbalance event occurs, balancer algorithm executes automatically and redistributes chunks among shards in order to maintain balance in the system. Furthermore, our approach performs less chunk migration which in turn reduces the overhead cost of migration over MongoDB load balancing algorithm, resulting into the efficient utilization of the shards.

For analytical purpose, the same experiment is performed with 1000, 10000, and 100000 chunk count capacity shards which is clear from the Fig. 2a.

It should be noted that the memory utilization of the shards in the process of load balancing, then automatic load balancing algorithm and modified MongoDB performs exactly the same as mentioned in Fig. 2b. Hence, our algorithm performs better by minimizing overhead cost due to chunk migrations which leads to improved load balancing computational complexity without compromising with space utilization. Thus, we are able to improve two factors of the balancing algorithm—computation cost of chunk migrations and memory utilization of the shards.



(a) Migration rate (b) Space utilization

Fig. 2 Experimental evaluation of modified MongoDB and MongoDB based on migration rate and space utilization

5 Conclusion and Future Work

The large-scale applications and data processing require handling of issues like scalability, reliability, and performance of distributed storage systems. One of the prominent issues among them is to handle the skewness in the data distribution and accessing of data items. We have presented an improved algorithm of load balancing for NoSQL database MongoDB, which handles aforementioned issues by providing automatic load balancing. We have analyzed our algorithm and shown that our approach is better than many similar systems employed specifically in MongoDB database [14], in terms of chunk migration and memory utilization for the individual shard.

Our proposed method is for MongoDB based NoSQL database systems. We will try to incorporate this approach into existing different flavors of NoSQL.

References

1. Church, George M., Yuan Gao, and Sriram Kosuri, 2012, "Next-generation digital information storage in DNA." *Science* 337.6102: 1628–1628.
2. Dean, Jeffrey, and Sanjay Ghemawat, 2008, "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1: 107–113.
3. Lakshman, Avinash, and Prashant Malik, 2010, "Cassandra: a decentralized structured storage system." *ACM SIGOPS Operating Systems Review* 44.2: 35–40.
4. M. Ali-ud-din, et al., 2014, "Seven V's of Big Data understanding Big Data to extract value," *American Society for Engineering Education (ASEE Zone 1), 2014 Zone 1 Conference of the, Bridgeport, CT, USA*.
5. E. Dumbill, 2012, "What is big data?," O'Reilly Media, Inc., Available: https://beta.oreilly.com/ideas/what-is-big-data.
6. DeCandia, Giuseppe, et al., 2007, "Dynamo: amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41.6: 205–220.
7. Cooper, Brian F., et al., 2008, "PNUTS: Yahoo!'s hosted data serving platform." *Proc. of the VLDB Endowment* 1: 1277–1288.
8. Chang, Fay, et al., 2008, "Bigtable: A distributed storage system for structured data." *ACM Trans. on Computer Systems (TOCS)* 26.2: 4.
9. "MongoDB," MongoDB Inc., 2015, Available: https://en.wikipedia.org/wiki/MongoDB.
10. E. A. Brewer, *Towards robust distributed systems. (Invited Talk),* Oregon, 2000.
11. Featherston, Dietrich, 2010, "cassandra: Principles and Application." *Department of Computer Science University of Illinois at Urbana-Champaign*.
12. Thusoo, Ashish, et al., 2010, "Data warehousing and analytics infrastructure at facebook." *Proc. of the 2010 ACM SIGMOD Inter. Conf. on Management of data*.
13. Glendenning, Lisa, et al. "Scalable consistency in Scatter, 2011," *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*.
14. MongoDB Documentation," 25 June 2015. [Online].
15. Liu, Yimeng, Yizhi Wang, and Yi Jin., 2012, "Research on the improvement of MongoDB Auto-Sharding in cloud environment." *Computer Science & Education (ICCSE), 2012 7th Inter. Conf. on*. IEEE.
16. Gifford, David K, 1979, "Weighted voting for replicated data." *Proc. of the seventh ACM symposium on Operating systems principles*.

17. Lamport, Leslie, 1998, “The part-time parliament.” *ACM Transactions on Computer Systems (TOCS)* 16.2: 133–169.
18. Godfrey, Brighten, et al., 2004, “Load balancing in dynamic structured P2P systems.” *INFOCOM 2004. Twenty-third Annual Joint Conf. of the IEEE Computer and Communications Societies*. Vol. 4.