Journal of Computers Vol. 29 No. 5, 2018, pp. 232-243
doi:10.3966/199115992018102905018

A Distributed Caching Scheme for Improving Read-write Performance of HBase

Bo Shen[1,2*], Zi-Bo Yu[1], Wei Lu[1], Yi-Chih Kao[3]

[1] School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing, China
{bshen, 14120167, 16120103}@bjtu.edu.cn

[2] Key Laboratory of Communication and Information Systems, Beijing Municipal Commission of Education, Beijing, China

[3] 3Information Technology Service Center, National Chiao Tung University, Taiwan
ykao@mail.nctu.edu.tw

Received 7 January 2018; Revised 7 July 2018; Accepted 7 August 2018

Abstract. As an important infrastructure of big data, HBase has been wildly used in lots of businesses and provides high performance in massive data reading and writing. However, HBase is still undergoing continuous improvement for meeting the special demands in lots of individual application fields. The issues involve cache replacement algorithm, region split and compact strategy, load balance and so on, and a relatively uniform scheme is still needed. In this paper, we propose a RowKey generation method based on consistent hashing and introduce a pre-partition scheme to distribute data to RegionServers uniformly. We also give the consideration to reading performance while design the improved writing mechanism for several data types. Experiments show that the performance of reading and writing has been significantly improved comparing with original HBase, which benefits by mapping data to the virtual nodes in the designed hashing ring.

Keywords: consistent hashing, HBase, load balance, partition

1 Introduction

Nowadays, the amount of Internet data reaches level PB. Traditional storage methods will no longer meet the requirements of efficient IO performance and expansion capability. People begin to reconsider the technologies of storing and indexing massive data, and a kind of non-relational database named NoSQL begins to play an important role [1]. Compared with relational database, NoSQL has several advantages [2]. For example, fixed table pattern is not necessary for data storage. The native distributed structure of NoSQL can provide high concurrent and efficient reading and writing abilities to massive data.

The common NoSQL databases belong to four major categories [3]: key-value store database, column storage database, document database and graphics database. Generally, key-value database, such as Redis, is a kind of memory databases and usually used as content caching. So, it rarely considers the performance of data persistence. Column storage database, which is usually used to store massive data, has high abilities of distributed storage and scalability. It provides large storage capacity but relatively limited function of processing data relationship. The representatives are HBase and Cassandra. Document database is considered as semi structured in general and documents are stored in a specific format, for instance JSON. The table structure is variable and the data structure is flexible, for example MongoDB. Graphics database has flexible graphical model and can be extended to multiple servers, such as Neo4J.

HBase is a multi-row and multi-dimensional mapping table based on HDFS, which is inspired by the Google file system [4] and MapReduce [5] and provides high reliability, high performance and high availability. Currently HBase has become the most popular massive data storage, management and

[*] Corresponding Author

computing technology, with more than 60 related components in programming and data access framework [6]. HBase has been widely used by Internet companies such as Facebook, Alibaba and Amazon.

HBase continues to improve. In some cases, it still cannot meet the demand of business. For example, during the data writing stage of HBase, blocking mechanism is extremely easy to cause the split and compact operations are performed circularly, which greatly affects the write performance of HBase. Also, in data storage period, storage mechanism of HBase does not achieve load balancing purpose well [7]. It indicates that there are still some room for improvement in HBase storage. The current researches usually take specific scenario as the targets of improvement and optimize the performance on specific business function. But many times, a universal optimization strategy is needed. Combination of current development of massive data processing, the design of the storage layer, as the bottom of the whole big data system, is particularly necessary. In view of this, we focus on the distributed caching strategy of HBase and propose an applicable scheme for enhancing the storage performance of HBase.

2 Two-stage Partition Storage Schemes

2.1 Problem Analysis

In HBase, region is the minimum unit of storage which fells in each RegionServer [8-10]. When data need to be updated, the region where the data is stored should be located first by RowKey [11]. And then the writing request is sent to the corresponding RegionServer. So, RegionServer plays a very important role in the whole process of storage [12]. Because each region stores data in ascending order of RowKey strictly, the seemingly distributed method tends to cause the problem of hot spot writing and result in unbalanced load.

The creation process of regions, which involves several components such as Region, MemStore and StoreFile [13-16], indicates that all data will be written to one region at the beginning when there is no partition. So only one RegionServer is working at this time. In addition, data are usually inserted on the basis of a certain order or a range of RowKey regardless of partitioning regions. Thus, data will be still inserted into individual regions at the same time. Normally, a region rarely performs write operations after data inserted unless some data need to be updated. Obviously, each RegionServer does a lot of storage operations at some time and the rest stays in idle waiting period, which do not make the distributed storage systems work to their advantage.

There are three key mechanisms about data storage in HBase, named flush, compact and split [17], which are all carried out in region unit and have the significant influence over storage performance. When writing in HBase, data are first written into memStore of distributed writing cache. Once the threshold of memStore is reached, all memStore in the same Region will perform flush operation and write data to StoreFile. If a table has multiple column families, many StoreFiles will generate and their number will quickly reach the threshold of compact. And then the compact probability will be greatly improved, which easily produces compaction storm and reduces the overall throughput of the system. Further, split mechanism is employed to ensure that each region does not store too much data, which is achieved by smaller split threshold value. Small split threshold can increase the efficiency of data inserting but lead to more frequent split operation. When split, regions enter offline mode and all requests to these regions will be blocked. It is thus obvious that small split threshold could result in many regions splitting at the same time and the whole access service of the system would be greatly affected. Specifically, the case that the average throughput of the system is larger but the throughput is extremely unstable will arise. On the other hand, the larger Split threshold can reduce the probability of Split and overcome the issue of unstable throughput but give rise to more StoreFile existing, which leads to more Compact operations in a Region. Compact operation first reads all data in original files and then writes them into a new file. After that the original files are deleted. Compacted file will become bigger and bigger and take up lots of IO. So, the average throughput decreases. In summary, it can be seen that the flush, Split, Compact mechanism of HBase are interrelated and constrain each other.

2.2 Pre-partition

The above discussion indicates that well designed RowKey [18] and partition method would be the way of solving the problems of unbalanced data distribution and hot-writing point. There are three patterns of partition in HBase, called automatic partition, manual partition and pre-partition. Automatic partition divided a region into two regions once the specified threshold is reached. Manual partition means a region can be divided into two by shell command manually. Pre-partition method builds several Regions according to the scope of RowKey in the initialization stage of a table. After pre-partition, each region is assigned a startKey and a endKey except the first Region which has no StartKey and the last Region no EndKey, as shown in table 1. Where StartKey and EndKey have the same means as that in range query of HBase [19-20]. When writing data, the request will be sent to the corresponding Region according to the RowKey and the requests field of the Region will plus one. Obviously, pre-partition pattern is beneficial to share the load of RegionServer.

Table 1. Partition information

Name	Region Server	Start Key	End Key	Requests
t1,,1479956792917.ebbebfabdb130955225177630abbfb03.	localhost:60030		201605000000	0
t1,201605000000,1479957153101.ba9a2b4da601d84866d19e065c73eca0.	localhost:60030	201605000000	201610000000	0
t1,201610000000,1479957270016.8248f7556d2e45cf78e93d16a8d7aa04.	localhost:60030	201610000000	201611000000	0
t1,201611000000,1479957270016.6f47062bdb2cadc6301570366672a2a8.	localhost:60030	201611000000		0

However, it is difficult to estimate the range of every Region in accordance with the RowKey of each table, which means no universal pre-partition scheme exists. Further, the current simple pre-partition method does not take the situation of cases that node is added or deleted. It also does not consider the dynamic adjustment of partition for achieving load balance.

To distribute data onto more than one Regions uniformly, here we first calculate the hash value of RowKey when pre-partition is executing. Hashing algorithm has the ability of changing the distribution of data and achieve uniform distribution. Once data are distributed equally to each Region, the problem of writing hot point is avoided naturally. Further, the uniqueness of hash result ensures that multiple calculation results of a data have the same value, which enables the data to be located in the correct position when query.

There are many hash algorithms in literature. We need the RowKey hash to ensure the uniform distribution of data. Meanwhile, the calculation performance and conflict probability are also important. Fig. 1 shows the distribution of hash value of several common hash algorithms, which indicates that MD5 [21] and SHA-1 have stable and relatively uniform distribution. Table 2 gives the number of conflict and running time on 100 million randomly generated string. The results show that MD5 and SHA-1 do not bring any conflict although they take more time. Comparing with SHA-1, MD5 has better calculational performance. So, we use MD5 as the hashing method of RowKey.

MD5 produces 128-bit hash value. In order to take advantage of its distribution character, we map this 128-bit value into a string with 32 letters from A to P. This means every 4 bits are converted to a letter of A to P. And then, the new RowKey is formed by MD5 hash value of the original RowKey and the original RowKey.

The new RowKey ensures all data to be stored in HBase begin with a letter from A to P, which can be used as the dividing point of Regions. That is, the data having the RowKey starting with A will be stored in Region 1, and so on. Because of having explicit StartKey and EndKey, all pre-regions can be built in the initialization stage of a table.

2.3 Adaptive Partition

Pre-partition mentioned above could be helpful at initial phase of storing data into HBase, but some RegionServer might still be congested due to their overload and performance differences. So, we introduce adaptive partition to bring load balance to HBase during run time.

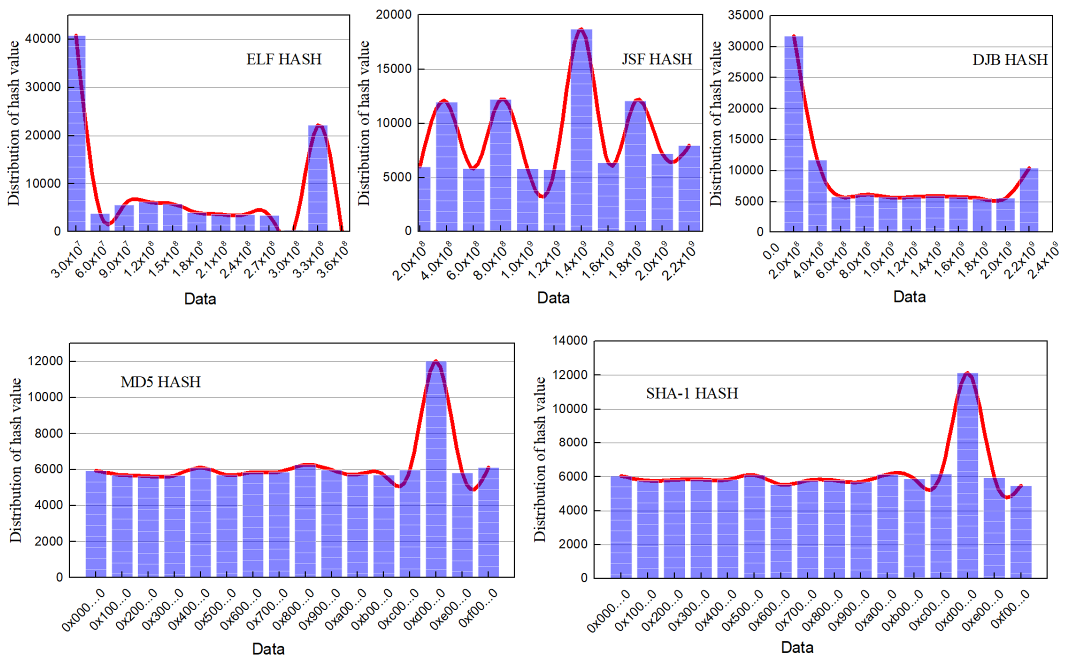


Fig. 1. Distribution of hash value of several algorithms

Table 2. Hash conflict and running time of several hash algorithm

Hashing algorithm	The number of conflicts	The running time(ms)
ELFHash	59168	78
JSHash	29306	61
DJBHash	30574	55
MD5	0	852
SHA-1	0	1423

After pre-partition, 16 Regions will be built. If every region is treated as a virtual node, all nodes divide the circle of consistent hashing space into 16 fragments as shown in Fig. 2, and consistent hashing algorithm just can be used to distribute each region to which of RegionServer. The specific condition of RegionServer decides which of several regions are distributed to every RegionServer. For example, suppose a system has 8 RegionServers, the virtual nodes corresponding to each RegionServer might be as shown in Table 3. Thus, the storage location corresponding to each data is in the first RegionServer arrived closewise.

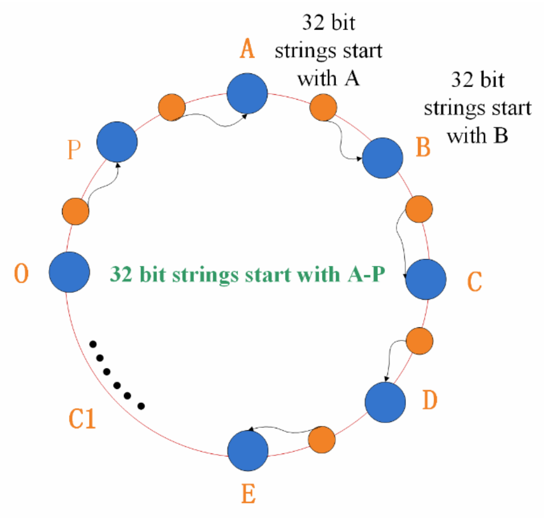

32 bit strings start with A
32 bit strings start with B
A
B
C
D
E
C1
O
P
32 bit strings start with A-P

Fig. 2. Distribute regions in consistent hashing space

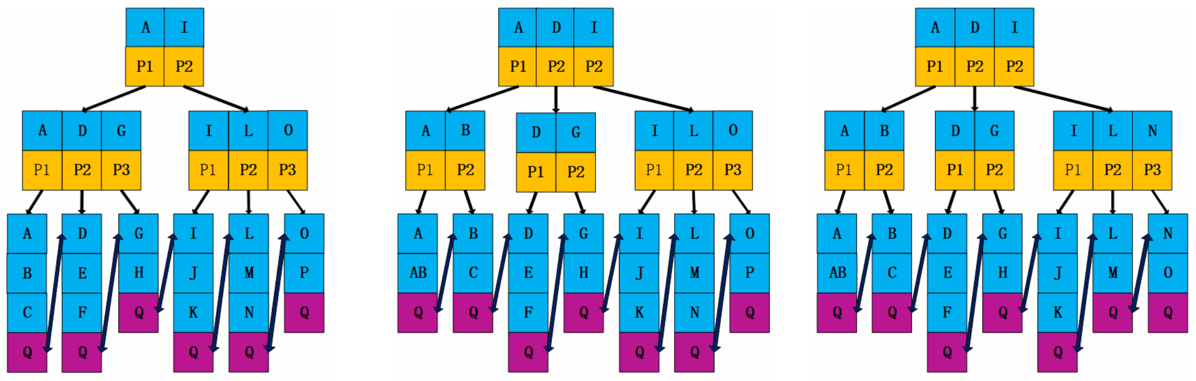
Table 3. Allocation of virtual nodes in RegionServer

RegionServer	virtual nodes	RegionServer	virtual nodes
RegionServer01	A，I	RegionServer05	E，M
RegionServer02	B，J	RegionServer06	F，N
RegionServer03	C，K	RegionServer07	G，O
RegionServer04	D，L	RegionServer08	H，P

Some researches indicate that massive data mapping always meet mapping failure caused by the addition and deletion of storage nodes identified by conventional Hash algorithm [22-25]. We find combining the pre-partition method with consistent hashing algorithm can solve the problem. The data structures of common implementation of consistent hashing algorithm include list and AVL tree. Comparing with list, AVL tree has better performance. But nodes in AVL tree cannot known the situation of their neighbors. The consistent hashing algorithm needs to know not only the information of storage nodes, but also their neighbors'. So, we employ the idea of B+ tree [26] to design the required data structure.

To search node on consistent hash ring, the initial machine nodes on the hash ring are treated as the key of nodes on B+ tree, and other machine nodes are stored as the information of leaf nodes on the tree. Because the data of adjacent nodes of target node are needed, the uni-directional chain list structure of B+ tree should be translated into bi-directional chain list, named extended B+ tree, in which the time complexity of finding target storage node is O(log n) and finding adjacent node is O(1). Fig. 3(a) gives the example of a third order extended B+ tree that uses A-P as its key.

The consistent hash ring based on extended B+ tree has flexibility to add nodes to the ring or remove them. If the target leaf node in the extended B+ tree is not full, new node can be inserted into the bi-directional chain list directly. Otherwise, the target leaf node will be divided into two parts and a new key will be added to its parent node. At the same, terminal node Q should be appended at the end of sub-chain, as shown in Fig. 3(b). Removing node from the consistent hash ring has reverse process, which means nodes combination will occur when a node has less child-nodes due to the removal of nodes. Fig. 3(c) gives an example of node removal.



(a) Extended hash algorithm implemented by extended B+ tree (b) Extension of the B+ tree after inserting AB (c) Extension of the B+ tree after deleting AB

Fig. 3.

2.4 RegionServer performance evaluation strategy

To achieve load balance, it is necessary to evaluate the performance of each RegionServer periodically for adjusting data distribution in running process of the system. Here we employ TOPSIS [27-28] and AHP [29] to design the evaluation strategy. On the basis of the storage mechanism of HBase, there are several factors that affect the performance of RegionServer node and the load balance of the whole system, as shown in Table 3.

Table 3. Factors affecting load balance

Factors affecting the performance of system (A)	Hardware (B1)	Region Number (C11), CPU(C12), Memory (C13), Disk (C14)
	Network performance (B2)	Bandwidth (C21), Throughput (C22), Number of RegionServer read requests (C23), Number of RegionServer write requests (C24)
	Mobile cost (B3)	Localized data (C31)

The factors of each node in a system form a matrix A named the decision matrix, in which the column represents RegionServer node and the row is corresponding factor set. Due to the difference of value range, these factors are normalized first by the following method.

$$\bar{a}_{ij} = \begin{cases} \dfrac{a_{ij} - \min(a_j)}{\max(a_j) - \min(a_j)}, & \max(a_j) \neq \min(a_j) \\ 0, & otherwise \end{cases} \tag{1}$$

where a_j is the column j of A and a_{ij} is the entry of A and $\bar{a}_{ij}$ is the normalized value.

Next, AHP is employed to calculate the weight of each factor, as following,

$$w_i = \sqrt[N]{\prod_{j=1}^{N} u_{ij}} \Big/ \sum_{j=1}^{N} \left(\sqrt[N]{\prod_{j=1}^{N} u_{ij}} \right) \tag{2}$$

where u_{ij} is the element of judgment matrix obtained through AHP.

And then, the element of normalized decision matrix is weighted as follow.

$$v_{ij} = w_i \bar{a}_{ij}$$

Let v_i^+ and v_i^- are the positive ideal solution and the negative ideal solution respectively, the distance between column J of the decision matrix and the ideal solution has the following form,

$$S_j^+ = \sqrt{\sum_{i=1}^{n} (v_{ij} - v_i^+)^2}\ ,\ S_j^- = \sqrt{\sum_{i=1}^{n} (v_{ij} - v_i^-)^2} \tag{3}$$

Then, the evaluation index of all RegionServer nodes and load equilibrium points can be calculated by,

$$S_j = \frac{S_j^+}{S_j^+ + S_j^-} \tag{4}$$

It means nodes are running healthily when the evaluation index of nodes is less than the evaluation index of the load balance point. When the evaluation index of half of nodes are higher than that of the load balance point, the load balance point index becomes the average value of current evaluation index of all nodes, and the system is waiting for the next round of evaluating. In other cases, the evaluation index that goes over load balance point index will be re-calculated.

2.5 Optimization of HBase Query Performance

In HBase, the read cache is controlled by BlockCache [30]. RegionServer contains a Block priority queue with three levels: Single, Multi and InMemory. Such that, the core content such as .META. Table is loaded into InMemory, and multiple use data is in Multi. This design is very flexible to avoid the mutual influence between Cache. However, it is not difficult to find that the BlockCache cache replacement strategy is a typical LRU algorithm, which decides the cache elimination only according to the write memory time. The design doesn't take care of data hotspots. The data in Single queue will upgrade to Multi queue when they are hit again and become old generation. After old generation are replaced, the memories they occupied will be recycled by CMS, which uses marking clean algorithm and generates lots of discontinuous memory fragments. When large objects need to be created, GC should be executed first. GC will stop the whole progress and affects the normal read and write requests seriously.

The current data replacing rule of Single queue is based on the timestamp of inserting data, which would result in the replacement of hot data by new data. An intuitionistic judgement is that the more times a data appears in unit time, the hotter it is. So, the frequency can be used to represent the heat of data, as follows,

$$F = \frac{c}{CurTime - LastTime} \tag{5}$$

where $CurTime$ is the time of current cache replacement, $LastTime$ indicates the last time of cache replacement and c is the times that a data has been queried after the last time of cache replacement occurred.

Consider that data have diverse frequency in different periods, recent query frequency of data has great influence on the frequency next time, and the influence is weakening with time, the caching evaluation index is defined as,

$$S(n) = \alpha \times F + (1-\alpha) \times S(n-1) \tag{6}$$

where α is the decay factor, which determines the proportion of the current and the previous heat. Obviously, through continuous iteration, the heat proportion of previously cached data is getting smaller and smaller, which avoids the problem that some data remain in cache for a long time once they appear.

To reduce full GC further, recycling the Blocks with large size should be a priority in Multi-layer. On the other hand, the data that are queried frequently are usually some data in Block. Evidently, the larger the Block is, the lower the value of caching it. Therefore, we assess a Block with the way,

$$V = \frac{S}{F} \times (LastVisit - BuildTime) \times (CurTime - LastVisit) \tag{7}$$

where S represents the size of a Block. Because the data entering Multi-layer in the earlier time will become useless data with larger probability and should be eliminated, but some data stay in cache for a long time due to constantly queried, the time factors should be considered.

LastVisit is the time of last access to the data, and *BuilderTime* indicates when the data entered Multi-layer. Once the storage capacity of Multi-layer reaches its threshold, all cached data should be sorted according to the evaluation indexes, and then the data with maximum index value should be eliminated to Single level until the storage capacity returns to normal.

3 Experiments and Results

To verify the effectiveness of proposed scheme, we use three types of data to test the writing and reading performance on modified HBase: random, centralized and continuous data set. Random data set includes 5 million lines with random RowKey, which is retrieved from Weibo system. The data in centralized data set have the RowKeys which appear in a particular range. For example, the order data of taxi in the rush hour have the timestamp that are concentrated. We generate 5-million-line data with this characteristic by codes. Continuous data mean their RowKeys are incremental or decremental, which are easily generated by codes.

Two kinds of clusters are built for the experiment, homogeneous cluster composed by the same computers and heterogeneous cluster including computers with various hardware configuration. All clusters run original HBase first, and then run the proposed HBase distributed cache strategy. The performance of data query and write under the two strategies are compared.

First, we test the write operation in homogeneous cluster by inserting 100 thousand, 1 million, and 5 million data of three different types to HBase cluster. We compare the writing time of native HBase and the HBase applied caching strategy. The results are shown in Fig. 4.

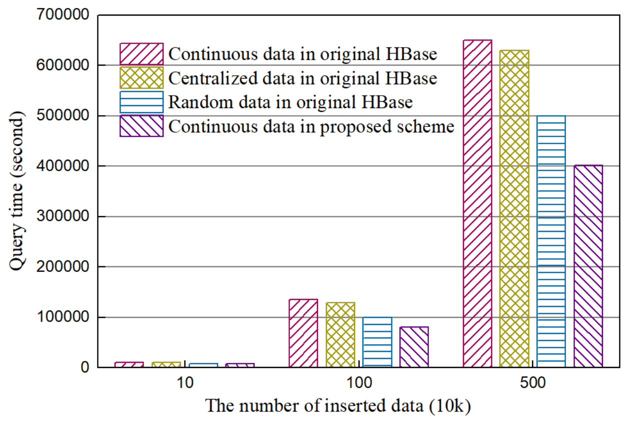


Fig. 4. Comparison of inserting data time

Obviously, after improving the storage mechanism, the writing time has a significant reduction comparing with the original HBase, no matter for which type of data. The reason is that there is a great difference in quantity of writing requests handled by each RegionServer although the number of Region allocated in each of them is basically the same, as shown in Fig. 4.

The plots in Fig. 5 imply that the number of processed requests vary greatly for continuous data in each time unit by original HBase system, and there will be a node in rarely working state for each time unit. It proofs the original HBase will write into a Region preferentially without pre-partition and current writing requests will arrived at the latest Region until the Region needs to be split. Without pre-partition, about 4 Regions will generated in each statistics time interval, which causes a waste of distributed system. The results indicate that the proposed caching strategy can guarantee the balance of requests of each RegionServer due to MD5 hashing of RowKey and pre-partition of Region. And for different type of data, the proposed method can help for realizing the performance of distributed storage system of HBase.

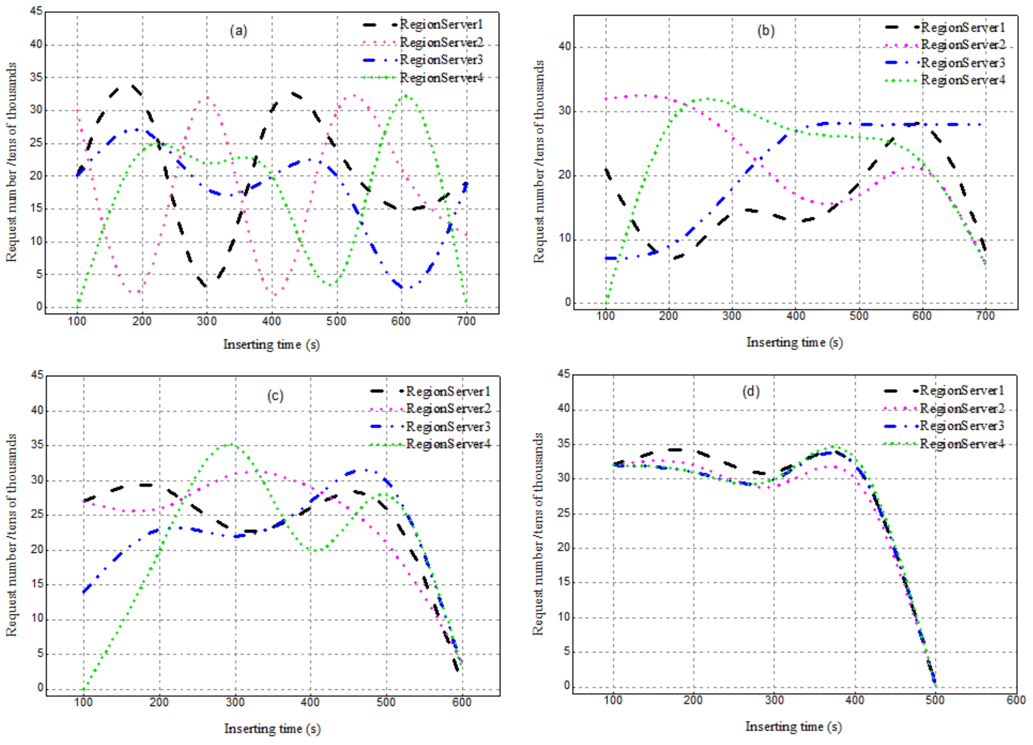


Fig. 5. The processing of various data types by HBase

Further, Fig. 6 gives the results of writing data into heterogeneous clusters which is to verify the correctness of RegionServer performance evaluating strategy. So, the storage scheme of native HBase is compared with that of MD5 pre-partition design by using random data. It can be observed that the scheme employed performance evaluation has the shortest writing data time in case of various data size. In the original HBase strategy, the Region distribution is determined by the HMaster, and Master has great randomness to decide the unallocated Region. In addition, the original HBase judge the load balance by the number of Regions, so the Regions distributed on each RegionServer can stay in equilibrium.

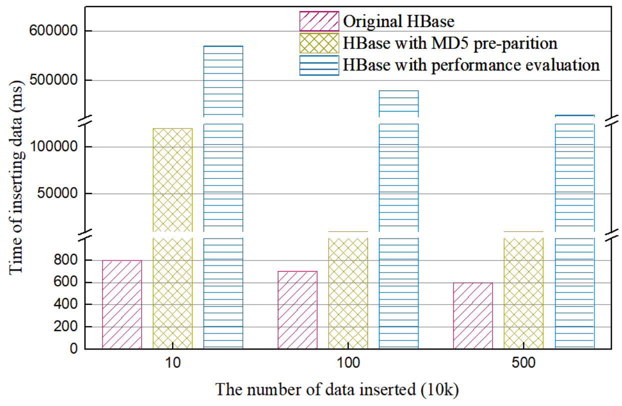
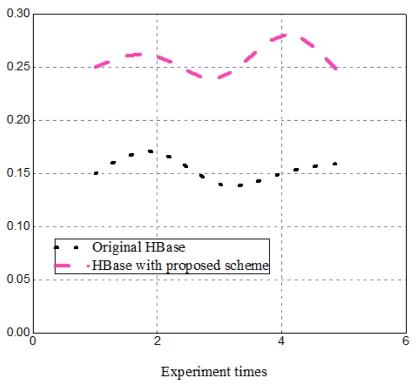


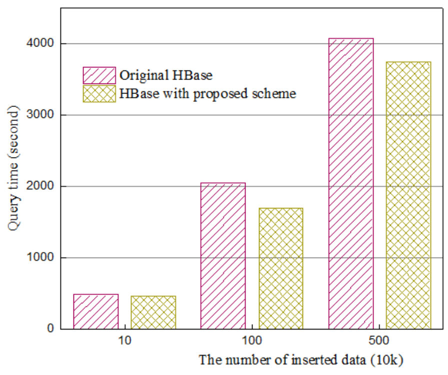
Fig. 6. Comparison of inserting data time

The MD5 pre-partitioning method initially divides the entire storage area into 16 areas (A-P), each of which contains a Region and is allocated to different RegionServer. That is, initially each RegionServer contains 4 Regions. For 5 million micro-blog user data with 2G bytes and the default HBase partition threshold of 64M, each region split into two after data writing, which proves that the MD5 uniform hashing has better performance of load balance.

Next, we test the random reading performance of HBase and proposed scheme. The query conditions are the 2 million items extracted from 5 million data randomly, and the hit rate on cache of each query is counted. The results are shown in Fig. 7.



(a) Comparison of cache hit rate



(b) Results of random reading

Fig. 7.

After optimizing, the reading cache the hit rate is obviously improved. This is because the separated design of Single-layer and Multi-layer give the opportunity of adding data heat comparison into Single-layer and adding data block size comparison into Multi-layer, which are both helpful for enhancing the hit rate of query.

We also test the continuously reading performance. In each query we choose a starting point and select up or down consecutive records of the point as query conditions. When the total number of queries achieve 10 million, the number of data processed in per second is recorded. The results are shown in Fig. 8.

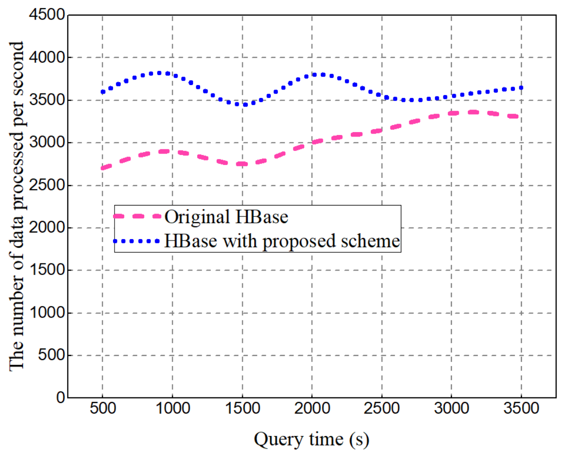


Fig. 8. Processing speed of continuous data

We find that, the original HBase sorts data by dictionary order and cache data by blocks, which provides high accessing performance. But the proposed method uses hashed RowKey and distributes continuous data to various nodes, which leads to low performance at the beginning of test. After that the second level cache stores frequently-used data, which makes the performance of proposed scheme close to the original HBase. This avoids the drawback brought by optimization of writing.

4 Conclusion

In summary, we propose a two-stage partition strategy for improving the performance of Hbase caused by load balance. In pre-partition stage, we use MD5 realize the uniform distribution of RowKey, and design a RegionSever performance evaluation method which employs the improved consistent hashing algorithm to map the partitions into a hash ring. We also introduce a design of read cache which considers the data heat and continues data distribution.

The experiment results about continuous, centralized and random data indicate that the proposed scheme can improve the performance of reading and writing data in both cases of homogeneous and heterogeneous clusters and has stable reading and writing speed no matter which type of data.

Acknowledgements

This research is supported by the National Key Research and Development Program of China (No. 2017YFC0840200) and the Fundamental Research Funds for the Central Universities (No. 2017JBZ107).

References

[1] M. Stonebraker, SQL databases v. NoSQL databases, Communications of the ACM 53(4)(2010) 10-11.

[2] T. Nasser, R.S. Tariq, Big rata fhallenges, J Comput Eng Inf Technol 4(3)(2015) 10.4172/2324-9307.1000161.

[3] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Transactions on Computer Systems 26(2)(2008) 1-26.

[4] S. Ghemawat, H. Gobioff, S. Leung, File and storage systems: the Google file system, ACM Symposium on Operating Systems Principles Bolton Landing 37(2003) 29-43.

[5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, ACM 51(1)(2008) 107-113.

[6] T. White, D. Cutting, Hadoop: the definitive guide, O'reilly Media Inc Gravenstein Highway North 215(11)(2012) 1-4.

[7] L. George, HBase: the definitive guide, Andre 12(1)(2011) 1-4.

[8] S. Nishimura, S. Das, D. Agrawal, D. El Abbadi, MD-HBase: a scalable multi-dimensional data infrastructure for location aware services, in: Proc. IEEE International Conference on Mobile Data Management, 2011.

[9] J. Huang, X. Ouyang, J. Jose, Md. Wasi-ur-Rahman1, H. Wang, M. Luo, H. Subramoni, C. Murthy, D.K. Panda, High-performance design of HBase with RDMA over InfiniBand, in: Proc. IEEE 26th International Parallel and Distributed Processing Symposium, 2012.

[10] C. Zhang, H.D. Sterck, Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase, in: Proc. IEEE/ACM International Conference on Grid Computing, 2011.

[11] S. Hong, M. Cho, S. Shin, J.-h. Um, C.-N. Seon, S.-K. Song, Optimizing HBase table scheme for marketing strategy suggestion, in: Proc. International Conference on Knowledge and Smart Technology, 2016.

[12] M.N. Vora, Hadoop-HBase for large-scale data, in: Proc. International Conference on Computer Science and Network Technology, 2012.

[13] R.C. Taylor, An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics, BMC Bioinformatics 11(S12)(2010) S1.

[14] D.J. Kim, J.H. Shin, K.S. Hong, Scalable RDF store based on HBase and MapReduce, in: Proc. International Conference on Advanced Computer Theory and Engineering, 2010.

[15] H. T. Vo, S. Wang, D. Agrawal, G. Chen, B.C. Ooi, LogBase: a scalable log-structured database system in the cloud, in: Proc. the 38th International Conference on Very Large Data Bases, 2012.

[16] G. Saloustros, K. Magoutis, Rethinking HBase: design and implementation of an elastic key-value store over log-structured local volumes, in: Proc. International Symposium on Parallel and Distributed Computing, 2015.

[17] H. Dutta, A. Kamil, M. Pooleery, S Sethumadhavan, J. Demme, Distributed storage of large-scale multidimensional electroencephalogram data using Hadoop and HBase, in: S. Fiore, G. Aloisio (Eds.), Grid and Cloud Database Management, Springer, Berlin, 2011, pp. 331-347.

[18] H. Ding, Y. Jin, Y. Cui, T. Yang, Distributed storage of network measurement data on HBase, in: Proc. International Conference on Cloud Computing and Intelligent Systems, 2012.

[19] Z.J. Yan, P. Sun, X.M. Liu, An HBase-based platform for massive power data storage in power system, Advanced Materials Research 1070-1072(2015) 739-744.

[20] L. Cai, S. Huang, L. Chen, Y. Zhang, Performance analysis and testing of HBase based on its architecture, in: Proc. International Conference on Computer and Information Science, 2013.

[21] R. Rivest, The MD5 Message-Digest Algorithm, RFC Editor, Fremont, CA, 1992.

[22] Q. Li, K. Wang, S. Wei, X. Han, L. Xu, M. Gao, A data placement strategy based on clustering and consistent hashing algorithm in cloud computing, in: Proc. 2014 9th International Conference on Communications and Networking in China (CHINACOM), 2014. doi:10.1109/CHINACOM.2014.7054342

[23] J. Petrovic, Using Memcached for data distribution in industrial environment, in: Proc. International Conference on Systems, 2008.

[24] B. Fitzpatrick, Distributed caching with Memcached, Linux Journal 124(2004) 72-76.

[25] Q. Liu, W. Cai, J. Shen, B. Wang, Z. Fu, N. Linge, VPCH: A consistent hashing algorithm for better load balancing in a Hadoop rnvironment, in: Proc. Third International Conference on Advanced Cloud and Big Data, 2015.

[26] C.S. Jensen, D. Lin, B.C. Ooi, Query and update efficient B+-tree based indexing of moving objects, VLDB (2004) 768-779.

[27] T.C. Wang, T.H. Chang, Application of TOPSIS in evaluating initial training aircraft under a fuzzy environment, Expert Systems with Applications 33(4)(2007) 870-880.

[28] I. Mahdavi, N. Mahdavi-Amiri, A. Heidarzade, R. Nourifar, Designing a model of fuzzy TOPSIS in multiple criteria decision making, Applied Mathematics & Computation 206(2)(2008) 607-617.

[29] M.C.Y. Tam, V.M.R. Tummala, An application of the AHP in vendor selection of a telecommunications system, Omega 29(2)(2001) 171-182.

[30] S. Lee, S. Shakya, R. Sunderraman, S. Belkasim, Real time micro-blog summarization based on Hadoop/HBase, in: Proc. IEEE/WIC/A CM International Joint Conferences on Web Intelligence, 2013.