International Journal of Database Theory and Application
Vol.7, No.4 (2014), pp. 261-272
http://dx.doi.org/10.14257/ijdta.2014.7.4.21

A Query Processing Framework based on Hadoop

Gang Zhao

Teachers' skill-training Center, Mianyang Normal University, Mianyang, Sichuan, China
2577477@qq.com

Abstract

With the development of cloud computing and big data, the massive volume of dataset proposes a big challenge for cloud data management systems. Unlike traditional database management method, cloud data queries are typically parallel and distributed. Intuitively, the query processing framework should embrace these characteristics. In this paper, by leveraging the inherent data structure of Hadoop HDFS, we design a query process framework. Specifically, facilitated by the key-value structure of HDFS, we construct a two-level index which first locates the target nodes for desired data, and then search within each node for further combination. As for joint operation, our query process engine optimizes by judging if the join key is equal to index key. If not, a MapReduce based join algorithm is then called. In this way, our method can reduce the cost of query processing. Besides, we conduct experiments for empirical evaluation.

Keywords: *Query processing, Hadoop, Cloud computing*

1. Introduction

Recent years have witnessed the increasing development of cloud computing [1, 2]. In cloud computing environment, enterprises or personal users can buy devices and computing according to their own requirements, without huge financial investment on high performance computers. Besides, the cost on software and hardware maintenance is dramatically reduced, since the infrastructure is provided by cloud vendors. For example, Amazon, Google, IBM and Microsoft have massively invested in research and development in cloud computing.

However, in the information age, the data generated by various enterprises and businesses has grown rapidly, which presents data management challenge to the cloud computing evolution. IDC estimates that by 2020, business transactions on the internet-business-to-business and business-to-consumer will reach 450 billion per day [3]. How to make use of this massive dataset to facilitate the management and analytics of data and mine the potential value behind the data, is a big challenge in many fields such as internet, communication and biomedicine. Moreover, traditional data management methods no longer fit in the cloud computing environment, due to the revolution of storage, computing and application patterns.

Indeed, some cloud vendors such as Google and Yahoo! have released some cloud data management systems, and some of them are used in production [4, 5]. However, the existing cloud data management systems provide limited functions with regards to query processing and optimization, especially compared to the traditional relational databases. Therefore, how to improve query processing and optimization in existing cloud platforms is the focus of this study.

As one of the most popular cloud platform, Hadoop [6] has two core components: cloud storage HDFS and a cloud computing engine Hadoop MapReduce. Hadoop enables distributed parallel processing of big data across inexpensive, industry-standard servers by either storing or processing the data. However, Hadoop is not a database management system,

ISSN: 2005-4270 IJDTA

but just a data processing system. That is, the query processing and optimization is hardly supported by the native Hadoop system. In this paper, we propose to enhance the query optimization based on Hadoop.

The remainder of this paper is organized as follows. Section 2 provides some related work. The overall framework is discussed in Section 3. Then the indexing component is introduced in Section 4, while the query processing component is presented in Section 5. Empirical experiments are conducted in Section 6. Finally, the paper is concluded in Section 7.

2. Related Work

Efforts on data management in cloud environment have always been done. Different from traditional relation based data management systems, cloud data management should make full use of the characteristics of cloud computing, so that the data management method is suitable to the scalability, flexibility and availability in heterogeneous environments. Related work of cloud data management can be categorized into three groups: index management, query processing and query optimization.

2.1. Index Management

Most existing cloud storage is based on key-value systems, which provides fast key based search. Although full table scan can be accelerated by parallel scan using MapReduce, it still remains a challenge when the size of data is extremely huge. Current techniques on indexing can be grouped into three classes.

2.1.1. Two-level Index. Wu *et al.*, [7] proposed a two-level index, where the local index is maintained on each node, and a global index is built upon local index tree. However, the global index is based on R-tree [8], which leads to many overlaps between nodes and therefore might produce false positive [9] results. To solve this issue, Andreas *et al.*, [10] introduced bloom filter [11] into R-tree.

2.1.2. Secondary Index. As a common practice in key-value store such as Bigtable and HBase, secondary index is built upon non-key columns. Zou *et al.*, [12] designed a Complemental Clustering Index (CCIndex). The basic idea is to store complemental information into the index table as well. However, the overhead is typically big for this type of index.

2.1.3. Multi-dimensional Index. Nishimura *et al.*, [13] presented a multi-dimensional index structure layered over a Key-value store such as HBase. The underlying key-value store handles high insert throughput and large data volumes, and also ensures fault-tolerance and high availability, while the index layer provides efficient multi-dimensional query processing. However, data consistency is a problem when data partition. Although Das *et al.*, [14] proposed a possible solution; this type of index is still too complicated for implementation. Besides, it gets worse if the data distribution is extremely skew.

Unlike above works, in this paper, we design a simple two-level index based on the distributed feature of cloud data systems. Our indexing method first locates correct nodes that hold desired data, and then search within specific nodes.

2.2. Query Processing

At the beginning, cloud data management systems such as BigTable, HBase and Cassandra only support some basic data insertion and acquisition operation [15]. Afterwords, efforts on developing SQL-like languages have been done by many companies and research institutes. For example, PigLatin [16] from Yahoo!, HQL [17] from Facebook, SCOPE [18] and DryadLINQ [19] from Microsoft.

Query processing in cloud data management systems is mainly about handing queries

using MapReduce framework. For example, Hive [17] employs a standard re-partition method, just similar to the sort and merge algorithm in DBMS. Similar method is also used in Pig [16] and Jaql [20]. Blanas *et al.*, [21] proposed a modified re-partition algorithm to solve the memory cache issue by using a semi-join algorithm. Okcan *et al.*, [22] presented a redundant redirect method to implement join operation, which reduces the impact of skew data distribution and the communication when shuffling. Besides, Afrati *et al.*, [23] proposed to perform join operation using MapReduce through transferring redundant data. Yang *et al.*, [24] modified typical MapReduce model by adding a Merge stage so that heterogeneous data sources can be easily processed. Similarly, Jiang *et al.*, [25] proposed a Map-Join-Reduce framework as an extension of MapReduce. However, the extended algorithm increases the problem complexity. Instead of modifying MapReduce for queries, in this study, we focus on the processing flow of handing queries.

2.3. Query Optimization

Lots of work has been done in this category. For example, Chi *et al.*, [26] proposed an algorithm to optimize the query scheduling based on Service Level Agreement (SLA). Zaharia *et al.*, [27] developed FAIR algorithm to optimize the execution of MapReduce jobs to ensure average allocation of resources. Phan *et al.*, [28] focused on the task scheduling optimization in heterogeneous environments, by transforming the problem as a Constraint Satisfaction Problem (CSP). Zaharia *et al.*, [29] designed LATE algorithm which can estimate the remaining time to complete for each task, and choose the longest one to execute. Nova [30] created a stream data management component based on Pig and Hadoop for streaming data management and query.

Intuitively, we want to make full use of the distributed and parallel characteristic of cloud data management systems. In this paper, we propose to leverage the underlying structure of cloud storage, and distribute data together by query logic. Then, the query parallelism is maximized and the communication between nodes can be minimized, and therefore the query processing can be optimized.

3. Overall Framework

In this paper, we focus on the query processing and optimization based on Hadoop.

First of all, the dataset is distributed on a HDFS cluster, as shown in Figure 1. Data is stored on DataNode in the form of blocks, and the whole system is coordinated by NameNode. Internally, a data file is split into one or more blocks which are stored in a set of DataNodes. Each block is replicated on multiple DataNodes for reliability.

The data distribution strategy, *i.e.*, data blocks splitting and assignment is crucial to parallel query processing. Indeed, data distribution is one of the most significant features of cloud computing environment. In a typical cloud data management system, data is split over multiple nodes and the position is usually determined by the row key of each record. In this study, we utilized a hash based distribution method. That is, apply a hash function over some attribute to map a data split to a DataNode. The intuition is to put logically relevant data physically together: the logic relevance is built upon the attribute, and the hash function assigns data records with the same row key to the same DataNode.

As mentioned above, we distribute our data splits according to their logical key values, notated as. Therefore, the DataNode on which data split is stored is decided by the result of a hash function over. On the other hand, the data inside the DataNode is organized by the row key of each record, notated as. Suppose the queries requested by the client are composed of a set of keywords, and the superscript denotes the identifier of each subquery. For example, if the client sends a query using two keywords, the query can be parsed into two subqueries that process independently and the final results are returned as the combination of above subqueries.

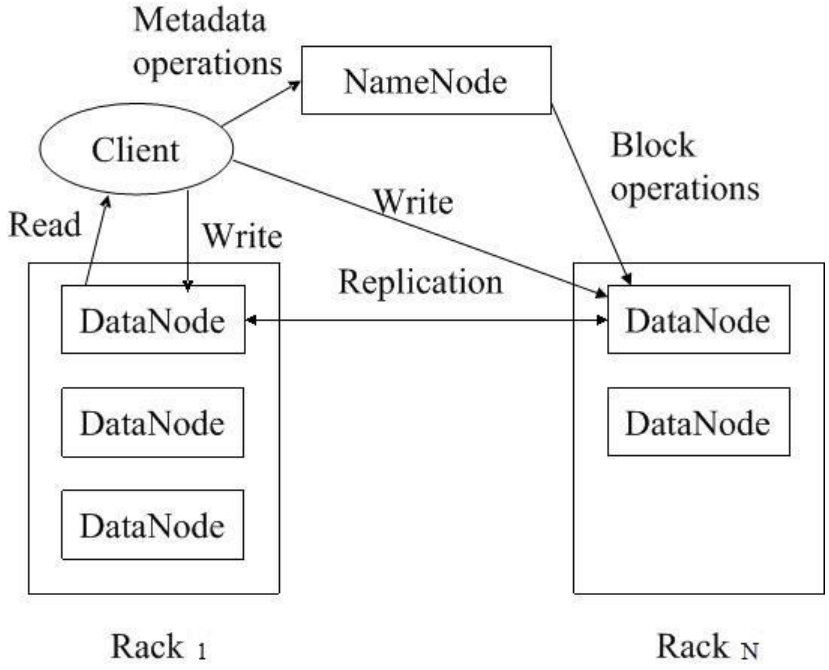

Figure 1. HDFS Framework

The overall processing flow is illustrated in Figure 2. Suppose the client sends a query request two keywords, that is, the request can be represented as $(K1^1, K2^1)$ & $(K1^2, K2^2)$. First, the Query Processing Engine (QPE) processes the query as the join of two subqueries, i.e., $(K1^1, K2^1)$ and $(K1^2, K2^2)$. Second, for each subquery, DataNode that holds specific data is located. That is, QPE passes $K1^1, K1^2$ to the Indexing Engine (IE) to find out correct DataNode. Third, within a DataNode, local searching is performed. That is, returning a subset of dataset using a local index table with keywords $K2^1, K2^2$. Last, QPE combines the results of subqueries and returns to the client.

The storage mechanism is naturally supported by HDFS, so we would not cover that in this paper. In the following sections, we will present two major components of above framework, *i.e.*, QPE and IE.

4. Indexing

Indexing is a fundamental strategy for query processing, which forms the IE component. Therefore, in this section, we present the indexing mechanism in this study.

Generally, we apply a two-level index. A typical example of two-level index is illustrated in Figure 3. Each level reduces the number of entries at the previous level, and thus narrows down the search space. As we know that in a cloud data management system, the whole dataset is distributed over a cluster of computers. First of all, the query processing component should be able to locate correct node which holds the target data. Therefore, the Level 1 index is built upon the data distribution for locating DataNodes, named as *Node Index (NI)*. Note that, we assume that the data distribution strategy ensures that relevant data is co-located.

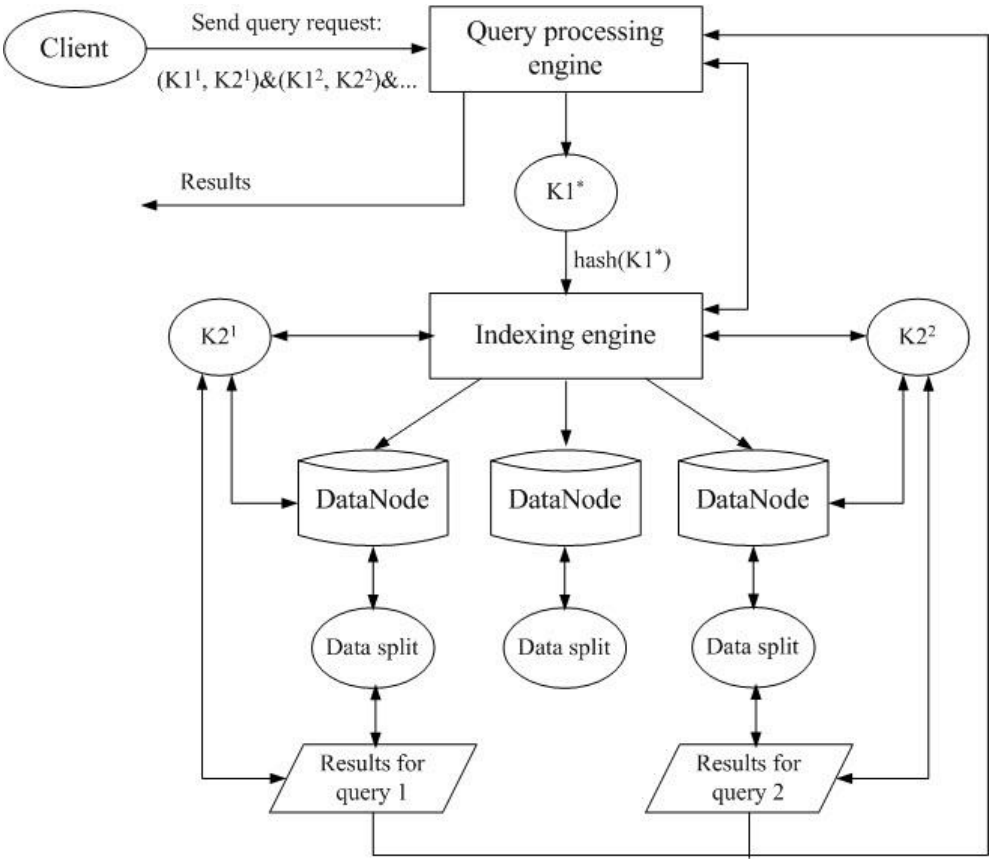

Figure 2. Processing Flow of our Framework

After specific DataNode is fixed, the search process is just similar to the single node query. That is, an index based search algorithm is performed, using the row key of the HDFS data structure. This is the Level 2 index, named as *Data Index (DI)*. Last, one or more records are located from the data split.

There are several points worthing noting. First, the mapping from Level 1 to 2 might be 1:N, that is, hash collisions. Second, even if the hash function over $K1$ produces a single value, the data split can be replicated over multiple DataNodes. Therefore, the combination of multiple sub-results from multiple DataNodes is unavoidable, no matter how the client sends the request. That is to say, the subqueries can be generated either by the actual keywords given by the client, or the multiple physical locations of target data.

To sum up, the IE component is responsible for searching target data given keywords. Algorithm 1 summarizes the process of search the two-level index using.

5. Join Processing

As mentioned above, the combination of multiple sub-results is necessary for query processing in a cloud environment, because the required data might be distributed on multiple physical nodes. Indeed, the combination is achieved by join processing.

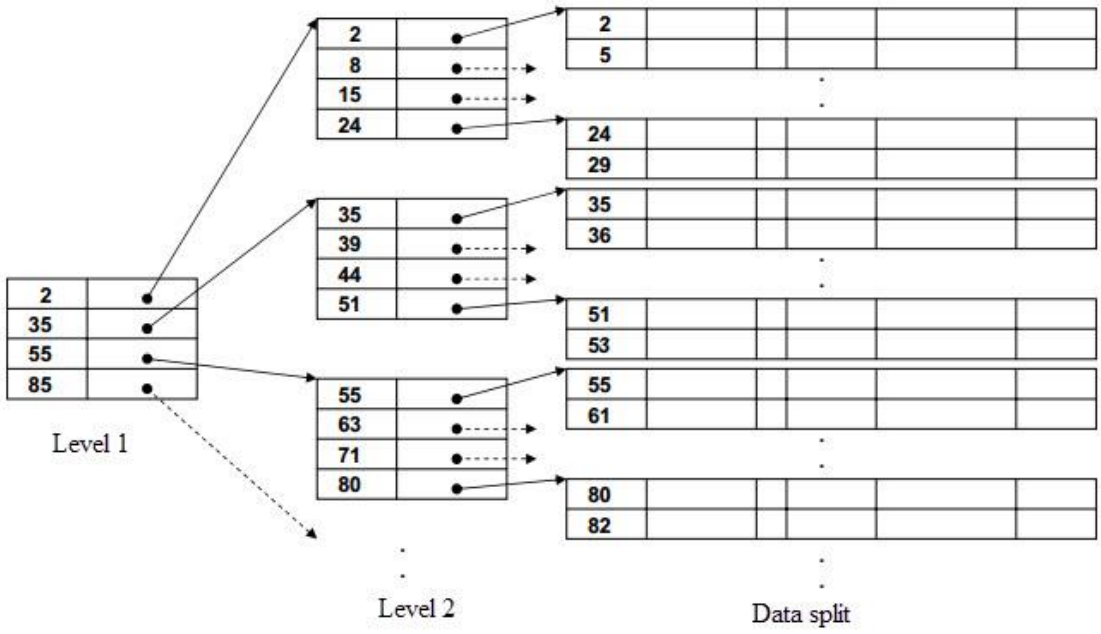

Figure 3. Illustration of Two-level Index

Algorithm 1 Searching a two-level index

1: set p as the address of Level 1 block of index;
2: read the index block whose address is p;
3: search block p for entry I such that $P(i) \leq hash(K1) < P(i+1)$;
4: set $p = P(i)$.
5: read the data split block whose address is p;
6: search p for record whose key is $K2$.

Figure 4 Searching the Two-level Index

If the subqueries are of identical structure, the combination is simply UNION, INTERSECTION or DIFFERENCE, which can be easily implemented through the row key $K2$.

If the request is to join two different tables a, b, the query is processed as Algorithm 2. (1) If tables a, b are on the same DataNode, the combination is performed as the typical database join. (2) If a, b are distributed on different DataNodes $a.node$, $b.node$, and the join key is $K2$, the join is processed in a two-step way. That is, first pull the Data Index (DI) from $a.node$ to $b.node$, find the INTERSECTION between $a.DI$, $b.DI$; then, pull the complete records from $a.node$ for the eligible row keys. (3) If a, b are distributed on different DataNodes , but the join key is other attribute instead of $K2$, Algorithm *MR-Join* is called for join processing.

Algorithm 2 Join processing

Input: tables $a, b, join\_key$;
Output: result t.

```
initialize t;
if a.node == b.node then
    perform database join;
else
    if join_key == a.K2 and join_key = b.K2 then
        find intersection between a.DI and b.DI, notated as t.DI;
        set t as the records whose key is within t.DI.
    else
        call MR-Join(a, b, join_key).
    end if
end if
return t
```

Figure 5. Join Processing Algorithm

Algorithm 3 MR-Join$(a, b, join_key)$

```
function MAP(K:null, V:a record from data split of either a or b)
    add a tag to V, notated as tagged_record;
    set composite_key as (join_key, tag);
    emit (composite_key, tagged_record).
end function
function REDUCE(K':a composite key with join key and the tag, V_LIST:
records for K' from a (first) and b (second))
    create a buffer Br for a
    for each record r in V_LIST of a do
        store r in Br
    end for
    for each record s in V_LIST of b do
        for each r in Br do
            emit null, new_record(r, l)
        end for
    end for
end function
```

Figure 6. MR-Join Algorithm

6. Experiments

In our experiments, we use five PCs to build a Hadoop cluster. One is used as master node, and also serves as the HDFS NameNode. The other four are slave nodes, and also serve as the HDFS DataNodes. The configuration of each PC is as follows: Intel Pentium 4 CPU 3.0 GHz, 2 GB ROM, 300 G hard disks, 100 Mbps NIC. The operating system is Windows 7, installed with JDK 1.6.0_27 and Hadoop 0.20.203.

The dataset we used is generated by the data generation tool dsdgen using TPC-DS. In order to evaluate the cost with various sizes of data, we generate four sample datasets: 10M, 20M, 40M, 80M.

6.1. Key Based Search Performance

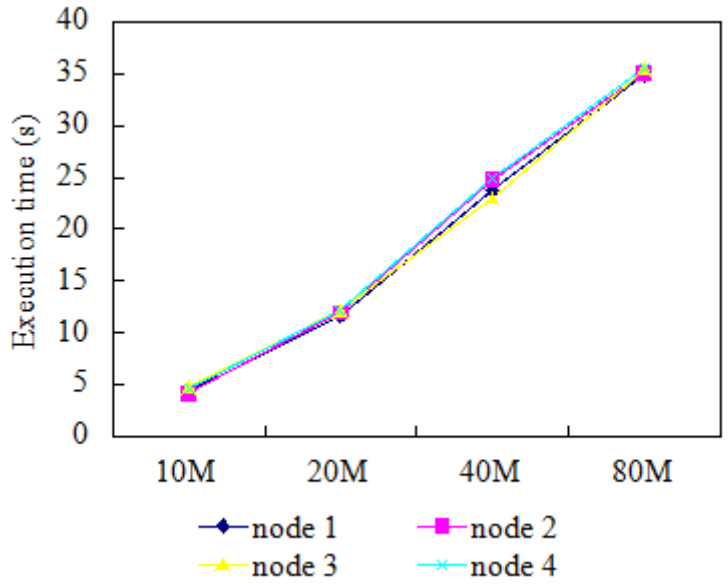

Figure 7. Performance of Individual Node for Local Key Based Search

First, we evaluate the performance of simple key based search. Figure 7 shows the performance of individual node for local search, and Figure 8 gives the join performance of key based search across various numbers of nodes.

From Figure 7, we can see that the processing capacity is generally identical for all nodes, and the larger the dataset is, the more cost for execution. Note that, the execution time would be longer than single node processing due to the distributed feature and the communication cost with the master node, even though only one node is actually used.

In fact, Figure 8 presents the combination performance of identical one level only query. The axis denotes the number of nodes involved in the query. For example, if, it means the target data for this key based search is spread over three nodes, and thus the results should be combined from all three nodes. Therefore, the execution time is the summation of processing time on each node and the combination cost. The result of 1 node is calculated the average from Figure 7. Of course, the more nodes involved, the longer it cost. The results also indicate that by storing relevant data together, execution cost could be efficiently reduced.

6.2. Join Performance

In this section, we evaluate the performance of join for tables on different DataNodes.

The first case is join key is same with the DI $K2$. The execution cost comes from three parts: locate DataNode by $K1$, INTERSECTION operation and copying data for combination. We compare our method with the intuitive method without using two-level index.

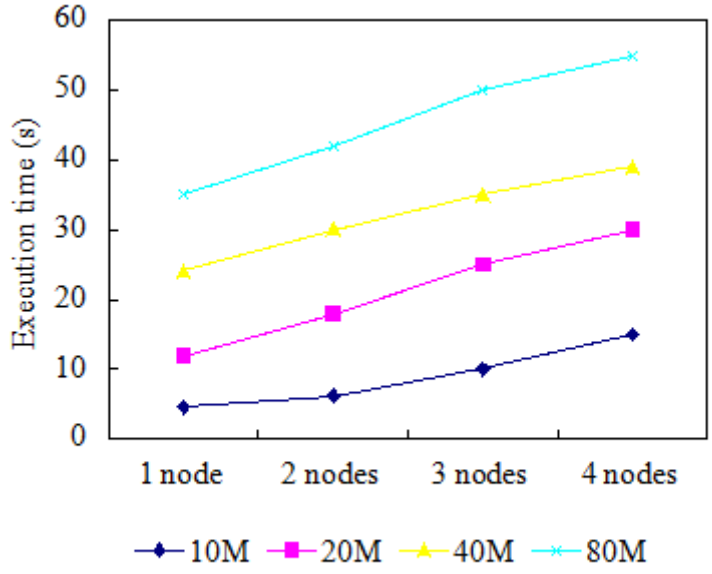

Figure 8. Performance of Key Based Search Across Nodes

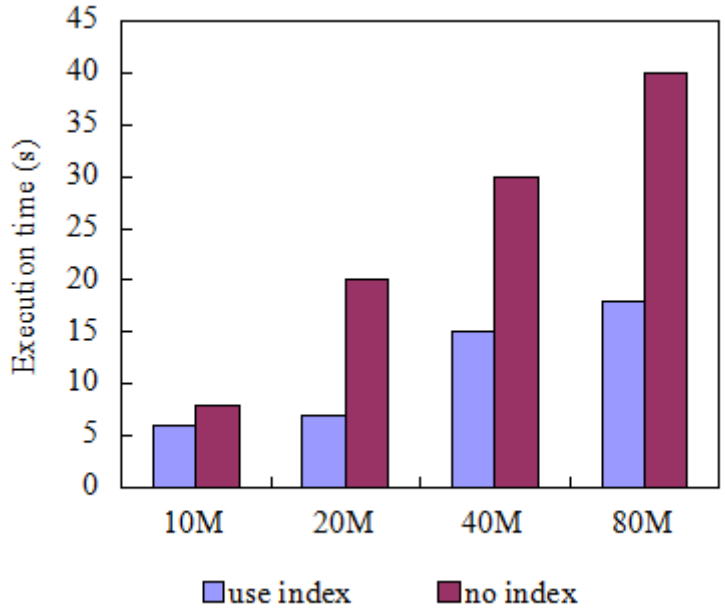

Figure 9. Join Performance when Join Key is K 2

From Figure 9 we have several observations. (1) Method that uses index outperforms the no index one no matter how large the data size is. (2) When using index, the performance of 10M and 20M is similar, but for 40M, execution time suddenly increases. The reason behind might be that the intersection set remains almost the same for 10M and 20M datasets, so the operation is same too. However, as the growth of the sample data, more target data might be involved, which enlarges the intersection set between two tables, and therefore the copying cost between nodes is greatly increased. The same thing applies for the 40M and 80M situations. (3) Unlike using index solution, the no index method always spends more time for larger dataset. The reason might be that without any index, the whole database has to be

scanned to make sure if a record matches the query or not.

The other case is join key is not $K2$. The query used in this case the identical to that in the first case except that the join key is different. In this case, MR-Join algorithm is called. The results are shown in Figure 10. Overall speaking, the performance is not that good if the join key is different with DI. However, using index is slightly better than no index. The possible reason is that in MR-Join, the reduce function iterates records in Tables a, b. This operation would be accelerated if the whole table scan is performed by the index key.

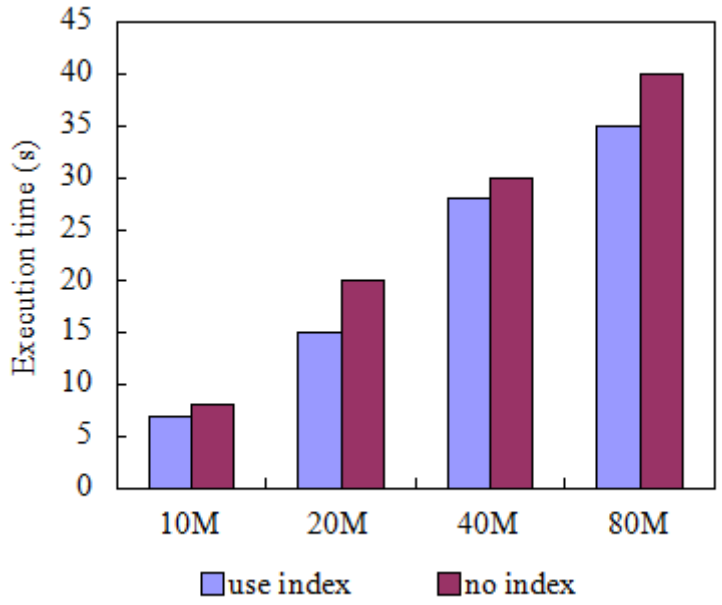

Figure 10. Join Performance when Join key is not $K2$

Therefore, we can conclude that the join performance is better if the join key and the index key are perfectly consistent. That is, the data structure should be designed the way they might be required.

7. Conclusion

In this study, we propose a query processing framework by leveraging the data structure of key-value store such as Hadoop. However, our method relies on the understanding of business logic, that is, the index key should be properly chosen. In future work, we will try to extend this framework to support various keys.

Acknowledgements

The authors would like to recognize the others who helped them and thank all our reviewers.

References

[1] M. Armbrust, "A view of cloud computing", Communications of the ACM, vol. 53, no. 4, **(2010)**, pp. 50-58.
[2] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)", NIST, special publication, vol. 800, no. 145, **(2011)**, pp. 7.
[3] A Comprehensive List of Big Data Statistics. http://wikibon.org/blog/big-data-statistics/.
[4] F. Chang, "Bigtable: A distributed storage system for structured data", ACM, Transactions on Computer Systems (TOCS), vol. 26, no. 2, **(2008)**, pp. 4.
[5] B. F. Cooper, "PNUTS: Yahoo!'s hosted data serving platform", Proceedings of the VLDB Endowment, vol. 1, no. 2, **(2008)**, pp. 1277-1288.

[6] T. White, "Hadoop: the definitive guide", O'Reilly, **(2012)**.
[7] S. Wu, "Efficient b-tree based indexing for cloud data processing", Proceedings of the VLDB Endowment, vol. 3, no. 1-2, **(2010)**, pp. 1207-1218.
[8] A. Guttman, "R-trees: A dynamic index structure for spatial searching", vol. 14, no. 2, ACM, **(1984)**.
[9] R. V. Hogg and A. Craig, "Introduction to mathematical statistics", **(1994)**.
[10] A. Papadopoulos and D. Katsaros, "A-Tree: Distributed indexing of multidimensional data for cloud computing environments", Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on. IEEE, **(2011)**.
[11] J. K. Mullin, "A second look at Bloom filters", Communications of the ACM, vol. 26, no. 8, **(1983)**, pp. 570-571.
[12] Y. Zou, Yongqiang, "CCIndex: a complemental clustering index on distributed ordered tables for multi-dimensional range queries", Network and Parallel Computing. Springer Berlin Heidelberg, **(2010)**, pp. 247-261.
[13] S. Nishimura, "MD-HBase: a scalable multi-dimensional data infrastructure for location aware services", Mobile Data Management (MDM), 2011 12th IEEE International Conference, IEEE, vol. 1, **(2011)**.
[14] S. Das and A. Divyakant, "G-store: a scalable data store for transactional multi key access in the cloud", Proceedings of the 1st ACM symposium on Cloud computing. ACM, **(2010)**.
[15] R. L. Grossman and Y. Gu, "On the Varieties of Clouds for Data Intensive Computing", IEEE Data Eng. Bull, vol. 32, no. 1, **(2009)**, pp. 44-50.
[16] C. Olston, "Pig latin: a not-so-foreign language for data processing", Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, **(2008)**.
[17] A. Thusoo, "Hive: a warehousing solution over a map-reduce framework", Proceedings of the VLDB Endowment, vol. 2, no. 2, **(2009)**, pp. 1626-1629.
[18] R. Chaiken, "SCOPE: easy and efficient parallel processing of massive data sets", Proceedings of the VLDB Endowment, vol. 1, no. 2, **(2008)**, pp. 1265-1276.
[19] Y. Yu, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language", OSDI, vol. 8, **(2008)**.
[20] K. S. Beyer, "Jaql: A scripting language for large scale semistructured data analysis", Proceedings of VLDB Conference, **(2011)**.
[21] S. Blanas, "A comparison of join algorithms for log processing in mapreduce", Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, **(2010)**.
[22] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce", Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, **(2011)**.
[23] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment", Proceedings of the 13th International Conference on Extending Database Technology. ACM, **(2010)**.
[24] H.-C. Yang, "Map-reduce-merge: simplified relational data processing on large clusters", Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, **(2007)**.
[25] D. Jiang, A. K. H. Tung and G. Chen, "Map-join-reduce: Toward scalable and efficient data analysis on large clusters", Knowledge and Data Engineering, IEEE Transactions, vol. 23, no. 9, **(2011)**, pp. 1299-1311.
[26] Y. Chi, H. Jin Moon and H. Hacigümüş, "iCBS: incremental cost-based scheduling under piecewise linear SLAs", Proceedings of the VLDB Endowment, vol. 4, no. 9, **(2011)**, pp. 563-574.
[27] M. Zaharia, "Job scheduling for multi-user mapreduce clusters", EECS Department, University of California, Berkeley, Tech. Rep. USB/EECS-2009-vol. 55, **(2009)**.
[28] L. T. X. Phan, "Real-time MapReduce scheduling", **(2010)**.
[29] M. Zaharia, "Improving MapReduce Performance in Heterogeneous Environments", OSDI, vol. 8, no. 4. **(2008)**.
[30] C. Olston, "Nova: continuous pig/hadoop workflows", Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, **(2011)**.

Authors

Gang Zhao, he Graduated from Southwest University in China in the year 2000, majoring in Science of Computer. Now, he is a lecturer in Mianyang Normal University and focuses on the teaching of computer basic as well as the application technology of database.