A Fast and High Throughput SQL Query System for Big Data

Feng Zhu, Jie Liu, and Lijie Xu

Technology Center of Software Engineering,
Institute of Software, Chinese Academy of Sciences, Beijing, China 100190
{zhufeng10,ljie,xulijie09}@otcaix.iscas.ac.cn

Abstract. Relational data query always plays an important role in data analysis. But how to scale out the traditional SQL query system is a challenging problem. In this paper, we introduce a fast, high throughput and scalable system to perform read-only SQL well with the advantage of NoSQL's distributed architecture. We adopt HBase as the storage layer and design a distributed query engine (DQE) collaborating with it to perform SQL queries. Our system also contains distinctive index and cache mechanisms to accelerate query processing. Finally, we evaluate our system with real-world big data crawled from Sina Weibo and it achieves good performance under nineteen representative SQL queries.

Keywords: Big Data, Query Processing, NoSQL, HBase, MapReduce.

1 Introduction

Many analytical tasks perform on ever growing massive data. As before, structured data storage and query still play an important role in the big data analysis. But for plenty of online Internet services, relational data queries require high performance, including scalability, low-latency and high-throughput. However, commonly used software is facing too many difficulties to store, manage and process big data, not to mention achieving these three criteria simultaneously.

We analyze the Internet service SQL logic and find that most SQL are read-only (i.e., no insert, update and delete). At the same time, NoSQL systems provide horizontally scalable storage and high-performance "***get(key)***" operations even under heavy read/write workloads. So our idea is to store the structured data in terms of ***Key-Value*** and convert SQL into a series of imperative operations written against distributed Key-Value stores. This is called **denormalization**. In this way, queries which need to scan large ranges of records or entire tables can be divided into lightweight distributed operations.

Based on the above discussion, we finally choose HBase [1] as our storage layer. There are some reasons: first, it is a distributed architecture and we only need to add servers to increase its scalability; second, the large table is automatically split into small regions. HBase itself can manage the metadata of these regions; third, its data model supports a flexible schema design. We can conveniently add some attributes to

X.S. Wang et al. (Eds.): WISE 2012, LNCS 7651, pp. 783–788, 2012.

the schema; the most important and the last one, HBase is built on top of Hadoop and provides a simple interface. We can process the data very expediently. However, HBase's simple key-value data model cannot satisfy various types of complex queries. So we add a distributed query engine integrated with HBase. The query engine is responsible for collecting and merging query results to clients.

This rest of the paper is organized as follows. Section 2 introduces our data modeling and denormalization method. Section 3 describes our detailed architecture and key technologies. The experiment is illustrated in section 4. Section 5 concludes the paper.

2 Data Modeling

Since NoSQL's data model is flexible and related to specific queries, we should first investigate the contest given SQLs. We find three significant features from them that strengthen our belief on NoSQL solution: (1) read-only queries, no write or update (2) fixed structured data without modification (3) query type is known beforehand with limited variable parameters. Based on these features, we can convert the relational data into Key-Value records and modify SQL into a series of "get(key)" operations. The main problem is that NoSQL does not support JOIN operation natively. We need to handle it at design time by applying "JOIN-free" technique. In other words, we create and partition joined table in advance, so subsequent queries can be done by just looking up the result table. This idea is simple but can dramatically reduce the query latency. Other complex and time-cost queries can be done in the same way.

Formally, this method can be defined as "Denormalization". In opposite to normalization in relational algebra, "Denormalization" encourages to store data in a query-friendly form to simplify query processing. Here, we not only tackle with JOIN, but also let structured data fit Key-Value pattern for scalability and high performance. Note that denormalization may increase our total data volume because of the different forms of duplicated data.

Now, we take the first contest query as an example to illustrate how denormalization works. The first query is to find the Top N suggested followees for user A. User A's followees are users who were followed by A. User A's r-friends are users who followed A and were followed by A. The recommendation algorithm is that: ***get all r-friends of A's r-friends, filter A's r-friends, order them by the number of people in A's r-friends list connecting to them, and then select the top N of them.***

The most time-consuming part arises from the table JOIN operation. To avoid it, we need to pre-create and partition A's r-friend table. If A's r-friend table is available, our query engine selects out all r-friends of A's r-friends, filtering A's r-friends and count the number of each user's connection with them; then the query engine returns the top-x user ids to the client.

Furthermore, the nineteen queries can be divided into two types: (1) Queries need no denormalization. These queries do no need join operation or index. (2) Queries need denormalization. These queries always contain complex operations. Several approaches such as building a secondary index, pre-joining tables and so on can be used to reduce their time cost.

3 System Architecture

Figure 1 shows the main components of our query system. The underlying file system is Hadoop Distributed File System (HDFS) [2, 3]. HDFS creates multiple replicas of data blocks and distributes them to different nodes. Based on HDFS, HBase plays the role of the database. We store all the preprocessed tables into HBase. Apart from the operations of "Query by key" and "Filter" provided by HBase, we also use coprocessor in HBase for complex queries.

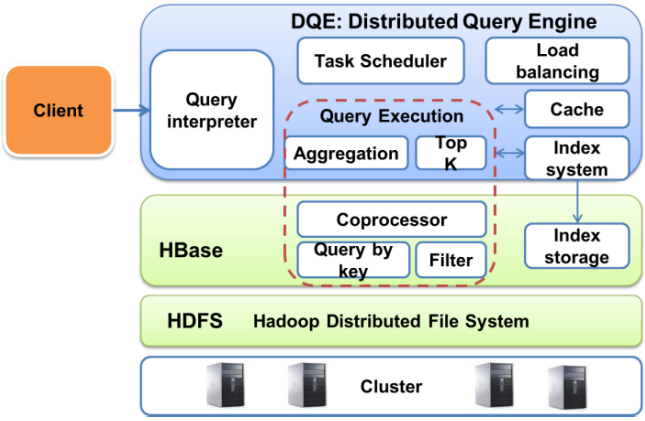

Fig. 1. System architecture

The upper layer is the distributed query engine (DQE). It is a logical processing component of this query system. To achieve high concurrency, we hold the principle of distributing the workloads to multiple nodes. Therefore, the DQE, as a middleware between the client and storage layer, adopts the master-slave structure. The master node is responsible for load balancing, aggregation, top-k selection and so on. Task Scheduler is responsible for distributing the query requests to various slave nodes.

Our system also contains distinctive index and cache mechanisms to accelerate query processing. Because the variables of nineteen queries in this contest are selected from specific collections, the results of these queries are limited. The cache may achieve 100% hit rate. **To test the real performance of our underlying system, we get the experiment data without cache system.**

Next, we list some key technologies while implementing the 19 query interfaces.

- **Data Load**

How to rapidly load the Big Data to the storage system is a basic problem before data query. With the help of Hadoop interfaces, we first upload all the data files to HDFS. Then, we use MapReduce [4] jobs to generate desirable tables. According to the logic of queries, we decide whether to store the generated tables in HDFS for other MapReduce jobs or in HBase for direct queries.

- **Row key design**

As a column-oriented database, HBase stores the data with row key in a lexicographic ascending order. Therefore, it supports not only single key query but also range query. To make the best use of this feature, designing a well-suited row key is very important. Like multi-dimension index, we sometimes combine several attributes as a composite row key. With the composite key, the value can be fetched effectively. Here we take several typical queries as examples to show this skill.

Among these nineteen queries, it is common to get the data in a time range. To avoid scanning a whole table, we put the time attribute into the row key. For example, after de-normalization in query 11, we get a table containing three items: ***(uid1, time, uid2)***, which means user ***uid1*** is re-tweeted by the user ***uid2*** at ***time***. In this case, the row key can be designed as ***uid1+time+uid2***. The corresponding range is ***userID+timestamp+uidx*** to ***userID+(timestamp+timespan)+uidy***. So we just need to scan this range to extract the total ***uid2*** set.

- **Key-list data model**

Most data relationship can be modeled as key-list model. For example, a tag may correspond to a large number of microblog and even more users.

Formally, we define key-list data model as "k-v_1($attri_1$), v_2($attri_2$), v_3($attri_3$)…".Item v_j($attri_j$) implies that the j-th value belongs to attribute $attri_j$. Then the problem can be defined as: given an attribute set SA and a key k, we need to retrieve the corresponding value set SV from k's values. SV= $\{v_j \mid attri_j \in SA \;\&\&\; v_j \in \text{value list (k)}\}$.

To make the best use of HBase's data model feature. We first consider and evaluate several table schemas below.

— Link the values together as a single value V, V= v_1 ($attri_1$) +v_2 ($attri_2$) +v_3 ($attri_3$)… and the row key is designed as k.

Row key	Column family
k	v_1 ($attri_1$) +v_2 ($attri_2$) +v_3 ($attri_3$)…

— Consider various values as different columns, then the schema can be designed as:

Row key	Column family			
k	v1 (attri1)	v2 (attri2)	v3 (attri3)	…

— Huge number of columns has negative affection on HBase. Considering putting the values into row key and the corresponding column is the attribute. Then the schema can be designed as:

Row key	Column family
k+ v_i	$attri_i$

— With the previous three designed schemas, we need to scan the whole value set for selecting SV. However, when the attribute set SA satisfies the sort condition, for example the *datetime*, row key can be designed with the attribute.

Row key	Column family
k+ $attri_i$+ v_i	…

The different schemas design fit for different cases, a comprehensive consideration is necessary.

- **Coprocessor Endpoint**

For those queries need to scan a wide range of the table or even the whole table, HBase coprocessor is a good choice. Resembling stored procedures in RDBS, coprocessor endpoint is powerful. A table stored in HBase is split into various regions. In the DQE, we invoke the coprocessor endpoint implementation for a table. It will execute in parallel on each region and returns the partial results to the DQE.

For coprocessor endpoint, there is also a tradeoff between the degree of parallelism and network traffic. More regions means less time for scanning a region, but it will cost more memory or even may block the network IO. Therefore, we should carefully set a proper region number.

4 Experimental Evaluation

This contest released a sample of structured big data from microblog. The dataset contains two parts: the first one is "Followship network" of 12.8GB and the second one is "Tweets" of 61.7GB. Through denormalization, we create about 30 tables and the total volume of data is over 300GB. Our query system is deployed on a cluster with 10 nodes. The configuration of each node is: CPU: Intel(R) Core™ i7-2600 3.4GHz; CPU cores: 4; Memory: 4 * 4GB DDR3; Hard Disk Capacity: 2 * 1TB; OS: Ubuntu-11.04 x86_64; Network: 1Gb/s.

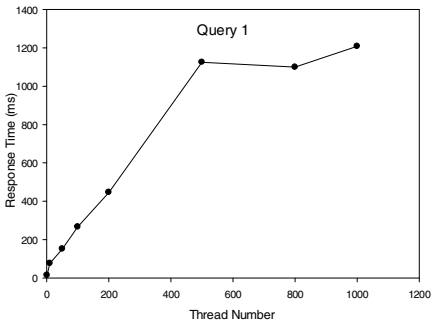

Fig. 2. Response time with different thread number

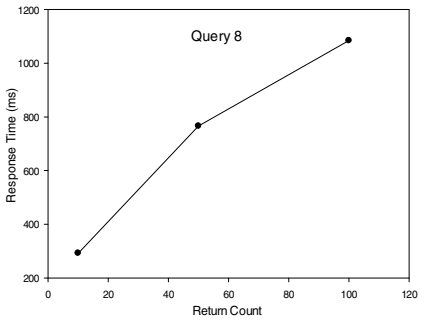

Fig. 3. Response time with different return count

Thread number, return count and time range are three main factors that affect the performance dramatically. Because there are so many experiments, here we just present a sample of the results and analyze the influence of each factor on response time. (1) Figure-2 shows the relation between response time and thread number of query 1. The return count is 50. (2) Figure-3 shows the influence of return count.

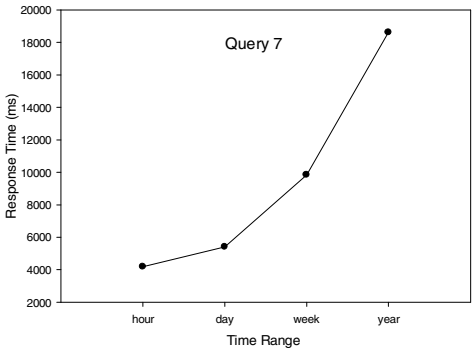

Fig. 4. Response time with different time range

The thread number of query 8 is 10. (3) Figure-4 shows the time range's influence on the scan rate. The thread number is of query 7 is 1.

Above figures just describe three main factors' general influence on performance. Our system actually performs well under different experiment configurations. The results show it is scalable. However, there may be data skew problem. For example, the number of user's followees may vary from one to thousands, which leads to the unpredictable results. Detailed results are available in separate files generated by BSMA tool.

5 Conclusion

In this paper, we first propose a denormalization method. This method is to store the structured data in terms of *Key-Value* and convert SQL into a series of operations written against distributed Key-Value stores. Then we present a query system based on HBase. Experiment shows the system can support high throughput query request with low latency. In the future, we will research how to support complex queries.

Acknowledgements. This work was partially supported by the National Natural Science Foundation of China (61170074), the National Grand Fundamental Research 973 Program of China (2009CB320704), the National Key Technology R&D Program (2012BAH05F02), National Core-High-Base Major Project of China (2010ZX01042-001-001-05).

References

1. Apache HBase, `http://hbase.apache.org/`
2. Apache Hadoop, `http://hadoop.apache.org/`
3. HDFS, `http://hadoop.apache.org/hdfs/`
4. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI 2004 (2004)