2014 IEEE International Congress on Big Data

A Memory-based Continuous Query Index for Stream Processing

Cuiwen Xiong Peng Zhang*
Institute of Information Engineering, Chinese Academy of Sciences Beijing, China
University of Chinese Academy of Sciences Beijing, China
Email:pengzhang@iie.ac.cn

Yan Li Shipeng Zhang Qingyun Liu Jianlong Tan
National Engineering Laboratory for Information Security Technologies Beijing, China
National Computer network Emergency Response technical Team Beijing, China
Email: liuqingyun@iie.ac.cn tanjianlong@iie.ac.cn

Abstract—**Most of the “Big Data” applications, such as decision support and emergency response, must provide users with fresh, low latency results, especially for aggregation results on key performance metrics. However, disk-oriented approaches to online storage are becoming increasingly problematic. They do not scale grace-fully to meet the needs of large-scale Web applications, and improvements in disk capacity have far out-stripped improvements in access latency and bandwidth. To this end, the paper proposes a memory-based continuous query index to implement scalable and efficient aggregation query.**

Keywords-*memory-based query**; data stream;query index*

I. INTRODUCTION

Big Data’s time is coming, the three defining characteristics are volume, variety, and velocity. In this context, most of the “Big Data” applications must provide users with fresh, low latency results. However, data access latency makes the traditional hardware systems and software architectures are difficult to meet these requirements.

TABLE I. THE DEVELOPMENT OF DISK ACCESS PERFORMANCE

	the mid of 1980s	2013	improvement of performance
Disk capacity	30MB	500GB	16,667x
Maximum transmission rate	2MB/sec	100MB/sec	50x
delay	20ms	10ms	2x
capacity/bandwidth (big block)	15s	5,000s	333x
capacity/bandwidth (1KB block)	600s	58days	8,333x
Jim Gray rule(1KB block)	5min	30hours	360x

In the past forty years, even though the capacity of disk was improved quickly, the improved transmission rate is still not very ideal. As seen from Table 1, the transmission rate is improved 50 times, but the access delay only is improved 2 times. If it is measured by capacity/bandwidth (Jim Gray rule[1]), the results seem worse. As a typical representative of low-latency and high throughput, the memory access could speed up the data access exponentially, and is an excellent solution to achieve efficient stream processing and query, such as Storm[2], S4[3], RAMCloud[4]. However, all of them consider little on the query index to improve the performance. To this end, this paper proposes a memory-based continuous query index for stream processing in our stream processing system MRAQ, which can implement scalable and efficient aggregation query on data stream.

II. THE OVERVIEW OF ACHITECTURE

In MRAQ, the fundamental storage unit is a segment, and each table is divided into a collection of segments. MRAQ partitions table into well-defined time intervals, typically an hour or a day, and may further partition according to values from other columns to achieve the desired segment size. In MRAQ, there are persistence segment and memory segment, the former is stored permanently in Hadoop Distributed File System(HDFS). All persistence segments have their metadata to describe their attributes such as the size, the compression format and the storage location. The persistence segment can be updated through the creation of a new persistence segment that obsoletes the older one. The segment covered very recent intervals is memory segment. The memory segment is incrementally updated after new data are injected, and can support immediately query during incremental indexing process. The memory segment can periodically be converted into persistence segment. The incremental indexing only works by calculating the aggregate value of the interesting metric. This often brings an order of magnitude compression without sacrificing the numerical accuracy. Of course, this is at the cost of not supporting queries over the non-aggregated metrics. The query involves the following types of nodes as shown in Figure 1. The memory query node is responsible for data injection, storage, and response to queries for the memory segments. Similarly, the persistence query node is responsible for loading and responses to queries for persistence segments. A query will firstly be sent to the master node, which is responsible for finding and routing the query to the query nodes containing related data, the query nodes execute their portion of the query in parallel and return the results to the master node, then the master node receives

*Author for correspondence

the results and mergers them, and finally returns the final result to the users. In addition, MRAQ also has a management node to manage the segment assignment, distribution and replication. The management node depends on the external MySQL database and the Apache Zookeeper to achieve coordination. The memory query node is highly scalable. If the injection rate exceeds the maximum capacity, additional memory query nodes will be added. All of them simultaneously consume the data from the same data stream, and each memory query node is only responsible for a part of the data source.

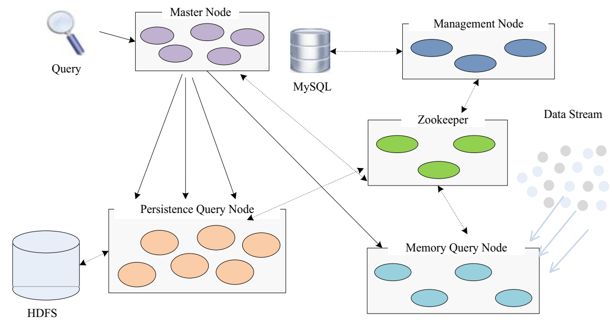

Figure 1. An overview of the MRAQ architecture

III. Memory-based Index

MRAQ adopts column-oriented storage format, which could make the CPU more efficient as a result of only needed data are loaded and scanned. MRAQ supports different column types, according to these types, MRAQ reduce the cost of storing a column on memory and disk by using different compression methods. We use dictionary encoding for String column. The dictionary encoding is a common method to compress data, for example, we map each publisher column into a unique integer identifier.

Publisher	Clicks	Revenue
sina.com	25	15.70
sina.com	42	29.18
yahoo.com	17	17.31
yahoo.com	170	34.01

sina.com -> 0

yahoo.com -> 1

The mapping transforms the publisher column as an integer array, and the array indices represent the rows of the raw data set. For the publisher column, we can transform publishers as follows: [0, 0, 1, 1]. The integer array of this result is very suitable for compression. The generic compression algorithms based on encoding are very common in column-oriented storage. Similar methods can be applied to the numeric columns. For example, the following two numeric columns can be transformed into two arrays.

Clicks -> [25, 42, 17, 170]

Revenue -> [15.70, 29.18, 17.31, 34.01]

In this case, we compress the original value instead of the encoded dictionary representations. In addition, MRAQ create additional indices for the string column to support any filters set. These indices are compressed and MRAQ operates their compressed form. Filters can be represented by the Boolean expression of multiple indices. Boolean operations on compressed indices can improve performance and save space. Consider the publisher column. For each unique publisher, we can get some information which row of the table the publisher is seen. We store the information in a binary array, which represents the row by the array indices. If the publisher is seen in a certain row, the array indices will be marked as 1, for example:

sina.com -> rows [0, 1] -> [1][1][0][0]

yahoo.com -> rows [2, 3] -> [0][0][1][1]

The sina.com appears at 0 and 1 column. The mapping of column values to the row indices forms an inverted index. In order to know which rows contain sina.com or yahoo.com, we join the two arrays with OR.

[0][1][0][1] OR [1][0][1][0] = [1][1][1][1]

IV. Experiment

We created a large test cluster with 80GB data including millions of rows. This data set includes more than a dozen dimensions, and the cardinalities ranges from double digits to tens of millions. We calculate three aggregation metrics for each row (count, sum, average) through 6 queries. Testing benchmark cluster contains 6 nodes, and each node has 16 cores, 16GB of RAM, 10GigEFA Ethernet and 1TB of disk space. Overall, the cluster contains 96 cores, 96GB of RAM, as well as enough fast Ethernet and enough disk space.

Figure 2 shows the cluster scanning rate and Figure 3 shows the core scanning rate. In Figure 2, we find the results of the expected linear scaling based on the result of the 5 cores cluster. In particular, we inspect the performance of the marginal revenue decreases with the scale of the cluster increasing, while in Figure 3 the core scanning rate of the query remains almost stable.

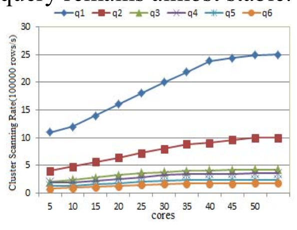

Figure 2. cluster scanning rate

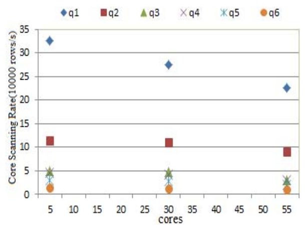

Figure 3. core scanning rate

Acknowledgements

The research work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No.XDA06030602; the China Postdoctoral Science Foundation under Grant No. 2013M541076.

References

[1] J. Gray, G Graefe. 1997. The five-minute rule ten years later, and other computer storage rules of thumb, ACM Sigmod Record, 26(4):63-68.

[2] Storm Project[EB/OL]. http://storm-project.net/.

[3] Yahoo S4[EB/OL].http://incubator.apache.org/s4.

[4] J. Ousterhout, P. Agrawal, D. Erickson, et al.. 2010. *The case for RAMClouds: scalable high-performance storage entirely in DRAM*, ACM SIGOPS Operating Systems Review, 43(4):92-105.