MRAQ: A Memory-Based Real-time Aggregation Query System

Peng Zhang, Junpeng Liu, Shu Li, Rong Yang, Pengxiao Li, Zhao Li, Qingyun Liu
National Computer Network Emergency Response and Coordination Center
Institute of Information Engineering Chinese Academy of Sciences
National Engineering Laboratory for Information Security Technologies
Email: lishu@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 index, however, disk-oriented approaches to online storage are becoming increasingly problematic. They do not scale gracefully to meet the needs of large-scale Web applications, and improvements in disk capacity have far outstripped improvements in access latency and bandwidth. To this end, the paper proposes a memory-based real-time aggregation query system named MRAQ, which adopts the shared-nothing architecture to support the scalability. MRAQ implements the efficient aggregation query through the bitmap index. The experiments show the effective on the performance improvements.**

I. Introduction

Big Data concern large-volume, complex, growing data sets with multiple, autonomous sources [1]. With the fast development of networking, data storage, and the data collection capacity, Big Data are now compelling most of the applications must provide users with fresh, low latency results, especially for aggregation results on key performance index [2]. However, the traditional hardware systems and software architectures are difficult to meet the demand for high performance. In the past forty years, the main storage medium is disk. Even though the capacity of disk is improved quickly, the performance is still not very ideal. As one can seen from Table I, the transmission rate is improved by 50 times, and the delay is only improved by 2 times. If it is measured by capacity/ bandwidth [3], the access delay of disk seems worse. As a typical representative of low-latency and high throughput, the memory storage can speed up the query exponentially, and is an excellent solution to achieve efficient query for Big Data applications.

However, when large-scale datasets have to be loaded in memory, it will be costly. But with the decrease of the price, memory storage gradually becomes available. Nowadays, many researchers have already proposed some excellent solutions. For example, NoSQL, Memcached(Cache), Flash Memory in the industry and MapUpdate, D-Streams, RAMCloud in the academia.

NoSQL represents a kind of non-relation databases. Some successful NoSQL solutions include Cassandra [4], HBase [5], BigTable [6], Dynamo [7], but NoSQL solutions are hardly in common use like the relational solutions, and they are still limited by the performance of disk storage.

Web caching is common adopted to improving the performance of Web application. However, the datasets that are not hit still are cached on the disk, so that a less miss will bring great performance loss. Moreover, the datasets generated by the Web applications have more and more complicated relationship, which makes it difficult to utilize the locality to reduce the caching overhead. Lets take Facebook for example, in August 2009, there is about 25% of all online datasets to be cached on the Memcached server [8], if considering the datasets cached on the database, actually there is almost 75 percent of datasets to be cached on the memory.

Flash Memory [9] is a longevity and nonvolatile memory, which can store the datasets even if the power is off., but there are still gaps with regard to latency and throughput compared with the Random Access Memory(RAM). As to the cost, RAM has the least cost under the condition of high query rate and small datasets; while disk has he least cost under the condition of low query rate and big dataset. In the two cases, Flash Memory always is the second choice.

In a word, whatever NoSQL or Cache or Flash Memory, all of them could not support the real-time query for data stream. Existing data stream processing systems, such as Esper, Storm, Yahoo S4 [10], place much emphasis on the single tuple processing and neglect the aggregation query optimization, so that these systems would be limited to implement aggregation query on key performance index. For this reason, users are in urgent need of an efficient aggregation query system for data stream. To this end, we have to compromise flexibility to design the special query, and there are several representative works as following:

MapUpdate [11] is a fast stream processing framework. Similar to MapReduce [12], developers only need to write several functions to use MapUpdate, especially Map and Update function, the MapUpdate can automatically implement these functions in the cluster. However, MapUpdate is different from MapReduce in the following aspects. Firstly, MapUpdate is used to process the data stream, the Map and Update functions must be defined based on the data stream. Secondly, since the data stream has no end, the Updater use the memory which is called slates to record the synopsis of data stream that have passed. In MapUpdate, slates actually are the updaters' memory, can be distributed on multiple nodes implementing the Map/Update. In addition, the slates can be persisted to the

TABLE I
THE DEVELOPMENT TREND OF DISK ACCESS PERFORMANCE

	the Mid of 1980s	2009	improvement of performance (times)
Disk capacity	30MB	500GB	16,667
Maximum transmission rate	2MB/sec	100MB/sec	50
Delay	20ms	10ms	2
Capacity/ Bandwidth (big block)	15s	5,000s	333
Capacity/ Bandwidth (1KB block)	600s	58days	8,333
Jim Gray rule(1KB block)	5min	30hours	360

key-value database in order to provide convenience for the future update.

D-streams [13] is an efficient and fault-tolerant model for stream processing, which consider the data stream processing as a series of batch data processing in small time interval. D-Stream stored the intermediate state on scalable distributed datasets RDD to implement fault tolerant. RDD is a no replication storage abstraction which could reconstruct the lost data by re-calculating the lost data lineage. In addition, D-Stream provides a special parallel fault recovery technology for RDD which makes RDD can quickly be recovered from a failure.

RAMCloud [14] is scalable high-performance storage and query system. In RAMCloud, all datasets are stored on RAM anytime, and the system can be automatically extended to thousands of servers, moreover, the number of the servers is transparent to the users, so that it seems as a system to the users. However, RAMCloud still have to face persistence and availability challenges: RAM is a kind of volatile memory, and the persistence and availability like disk is important. A servers failure and power fail in data center should not cause dataset loss and service interruption. A simple solution is to copy multiple backups stored in different servers, but the cost is too high, and if all servers of the data centers are power off, the dataset will still be lost.

As one can see, a common point of above three solutions is that all of them had a data abstraction cached on memory to improve performance. However, data stream has no end and the data volume is large scale, it is impossible to store all datasets in memory, so it is practical to combine memory with disk to design our system to support the efficient aggregation query for data stream.

The outline is as follows: Section II presents an overview of the architecture of MRAQ, including data model, data query. Section III presents the storage strategy. Section IV presents performance experiments. Section V presents our conclusions.

II. THE ARCHITECTURE OF MRAQ

This paper proposes a memory-based real-time aggregation query system named MRAQ, which adopts the shared-nothing architecture to support the scalability. The fundamental storage unit in MRAQ is segment. In MRAQ, each table will be divided into a collection of segments, where each segment contains about 10 thousand lines, as shown in Table II. MRAQ simplifies the data distribution, storage and queries with a timestamp column. MRAQ partitions data sources 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. The metadata of segment is composed of data source identifier, the time interval of the data, a version string that increases whenever a new segment is created, and a partition number. The read operation always access to the data in the segments with the latest version identifier for the time range. In MRAQ, most of segments are persistence segment. These segments are stored permanently in the 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 query during incremental indexing process. The memory segment could periodically be converted into persistence segment. In MRAQ, persistence segment and memory segment are created by incremental indexing process. 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; each node performs a specific function. The architecture of MRAQ is shown in Figure 1. The memory query node is responsible for data injection, storage, and response to queries for the most recent data. Similarly, the persistence query node is responsible for loading and responses to queries for historical data. Data in the MRAQ is stored in the storage node; the storage node may be a persistence query node or a memory query node. A query will firstly be sent to the master node, which is responsible for finding and routing the query to the storage nodes containing related data, the storage nodes execute their portion of the query in parallel and return the results to the master node, then the master node receives the results and mergers them, and finally returns the final result to the users.

The master node, compute node and memory query node are considered as queryable nodes. In addition, MRAQ also has a management node to manage the segment assignment, distribution and replication, but the management node is unqueryable node, it is mainly used to maintain the stability of the cluster. The management node depends on the external

TABLE II
THE SEGMENT EXAMPLE

Timestamp	Publisher	Advertiser	Gender	Country	Impressions	Clicks	Revenue
2017-04-03 T01:00:00Z	sina.com	baidu.com	Male	China	1800	25	15.70
2017-04-03 T01:00:00Z	sina.com	baidu.com	Male	China	2912	42	29.18
2017-04-03 T01:00:00Z	yahoo.com	google.com	Male	USA	1953	17	17.31
2017-04-03 T01:00:00Z	yahoo.com	google.com	Male	USA	3914	170	34.01

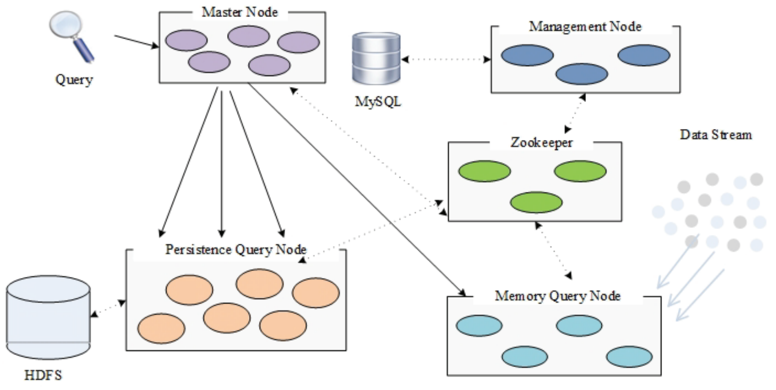

Fig. 1. An overview of the MRAQ architecture

MySQL database and the Apache Zookeeper [15] to achieve coordination.

A. *Persistence Query Node*

The persistence query node is the main workers of the MRAQ and does not depend on external components. The persistence query nodes load persistence segments from a permanent storage and make them queryable. Since persistence query nodes do not know each other, and there is no competition of single point between the nodes. The persistence query nodes only need to know how to perform their assigned tasks. To help other service discovery persistence query nodes and the segments they provide, each persistence query node maintains a connection with the Zookeeper. The persistence query nodes create a temporary node under specifically configured Zookeeper paths to publish their online status and served segments. A persistence query node loads new segments or drops existing segments by creating a temporary znodes under a special load queue path associated with the persistence query node. Figure 2 shows a simple interaction of a persistence query nodes and the Zookeeper. Each persistence query node has an associated load queue path. When they come online, they will publish their served segments in the path.

In order to make the segment queryable, a persistence query node must firstly possess a local copy of this segment. Before a persistence query node starts to download a segment from HDFS, it firstly checks the local disk directory (also known as cache) to determine whether this segment has been in the local storage. If the cache information of this segment does not exist, then the persistence query node will download metadata of this segment from Zookeeper. This metadata includes information about where the segment is located in HDFS and about how to decompress and process the segment. Once the persistence query node completes this process of this segment, it will publish that it can serve this segment in Zookeeper. At this moment, this segment is queryable.

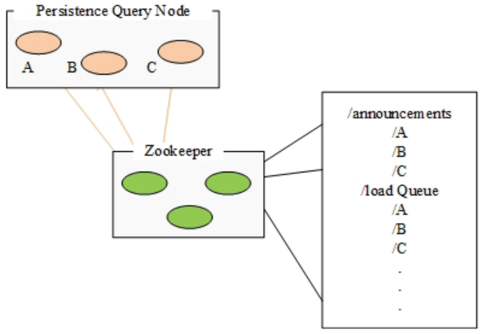

Fig. 2. The interaction of a Persistence Query Node and Zookeeper

B. *Memory Query Node*

The memory query node encapsulates the functions of real-time data stream injection and query. Data indexed via memory query node can be queried immediately. The memory query node consumes data, so it needs a corresponding producer to provide data. Typically, for the purpose of data persistence, a message queue, such as Kafka [16] placed between the producer and the memory query node, as shown in Figure 3.

Message queue shown in Figure 3 can be regarded as a buffer for incoming data stream. The message queue can maintain offsets indicating the location that the memory query node has read up to and the memory query node can periodically update this offsets. The message queue can also be seen as a backup storage for recent data stream. The memory query node

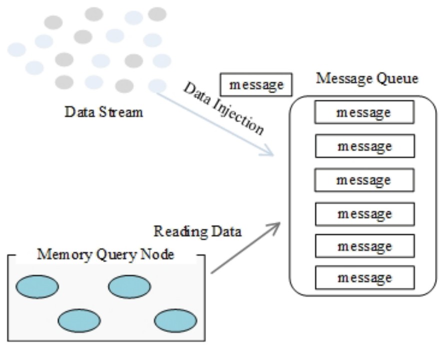

Fig. 3. Real-time data injection

injects data by read message from the message queue. The time from message creation to the message queue storage to message consumption may be about hundreds of milliseconds. The real-time data node maintains an in-memory index buffer for all injected message, as new message is injected into the message queue, these indexes are incrementally created and can also be directly queried. The memory query node persists periodically these indexes into disk. After persist, a memory query node uses the offset of last message of the most recently persisted index to update the message queue. Each persisted index is unchangeable. If a memory query node fails, when it starts to recover, it just needs to reload any index which has been persisted to disk and then reads the message queue from the point which the last offset is committed. Periodically committing offsets can reduce the amount of re-scanned data after a memory query node fails. The memory query node will upload this segment to HDFS and simultaneously provides a signal to the persistence query nodes to indicate the segment could be queried. When a memory query node transforms a memory segment into a persistence segment, there is no data loss.

Figure 4 shows this process. Similar to the persistence query nodes, the memory query node also publishes segments in the Zookeeper. Unlike the persistence segments, the memory segments can represent a period of time that extends to the future. The memory query node does not immediately merge and build a persistence segment for the previous hour until after some window times have passed. With a window time, the memory query node can disperse the data points to come and reduce the risk of data loss. At the end of this window time, the memory query node will merge all persisted indexes, and build a persistence segment for the previous one hour, and then send the persistence segment to persistence query nodes to serve. Once the segment on the persistence query node can be queried, then the memory query node will delete all the information of this segment and never serve this segment.

The memory query node is highly scalable. If the injection rate of a given data source exceeds the maximum capacity of the memory query node, additional memory query nodes will be added. Multiple memory query nodes simultaneously consume the data from the same data stream, and each memory query node is only responsible for a part of the data source.

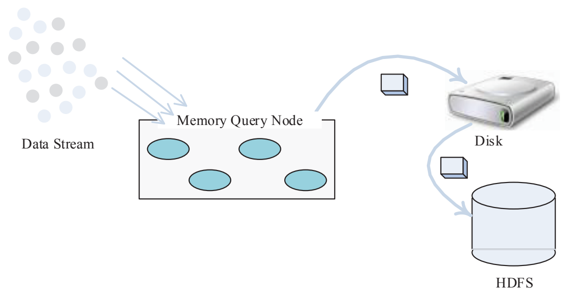

Fig. 4. Real-time data persistence

This naturally creates partitions across nodes. Each memory query node publishes the memory segment it is serving and each memory segment has a partition number. The data from all memory query nodes will be merged at the master node.

C. Master Node

The master node functions as query router, it can route query to the queryable node, such as the persistence query nodes and memory query nodes. The master node gathers the metadata published in Zookeeper about what segments exist and where the segments. The master nodes route incoming queries such that the queries hit the right storage nodes. The master node is also responsible for merging the query results from each node, before returning final result to the users. In addition, the master node provides data persistence layer through maintaining a cache for recent results. When multiple compute nodes fail and all copies of a segment are lost, if the information has already been stored in the cache, then the segment results can still be returned.

D. Management Node

The management node is primarily responsible for the management and distribution of segments, including loading of new segments, dropping outdated segments, the management of the replicated segments and load balancing of segments. The management node periodically checks the current status of the cluster. At the runtime, the management node compares the expected state of the cluster and the actual state of the cluster to make decision. The management node maintains a Zookeeper connection to obtain information of all nodes in the cluster. Meanwhile, the management node also maintains a MySQL database connection to get the information of operational parameters and configuration.

In the context of Big Data, a query often involves dozens or even hundreds of segments. Since single persistence query node is resource limited, the persistence segments must be distributed to different nodes of the cluster to ensure the overall load balancing. To achieve optimal load distribution, it is necessary to understand the query pattern. In general, the queries will cover the recent data of adjacent time intervals for a single data source. In generally, queries that access smaller segments get faster response. These query patterns indicate that replicating the recent persistence segments at a higher rate, distributing the large segments that are close in time to different persistence query nodes, and puts the segments from different data sources into one place.

III. Data Storage

MRAQ adopts column-oriented storage format. Considering aggregates over a large number of data, storing data as columns has the advantage in storing data as rows. Column-oriented storage could make the CPU more efficient as a result of only needed data are loaded and scanned. In a row-oriented data store, all columns associated with a row must be scanned as a part of the aggregate. The extra scanning time may degrade the performance as high as 250%.

A. Column Type

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. In the example as shown in Table II, the publisher, advertiser, gender and country column contains only are strings. String column is dictionary encoding. Dictionary encoding is a common method to compress data, for example, in Table II. We map each publisher into a unique integer identifier. 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.

$$sina.com \rightarrow 0$$
$$yahoo.com \rightarrow 1$$

In MRAQ, we use the string compression algorithm. Similar compression methods can be applied to the numeric columns. For example, the clicks and revenue column in the table can be transformed into an array respectively.

$$Clicks \rightarrow [25, 42, 17, 170]$$
$$Revenue \rightarrow [15.70, 29.18, 17.31, 34.01]$$

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 in Table II. For each unique publisher in Table II, 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:

$$sina.com \rightarrow rows[0, 1] \rightarrow [1][1][0][0]$$
$$yahoo.com \rightarrow rows[2, 3] \rightarrow [0][0][1][1]$$

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.

$$[0][1][0][1]\ OR\ [1][0][1][0]\ =\ [1][1][1][1]$$

The method to perform Boolean operations on a large bitmap set is often used in search engines. In MRAQ, we use the Roaring bitmaps algorithm [17] to compress the size of the bitmap by more than 80%. To achieve high availability and scalability, MRAQ adopts the following technologies to ensure there is no single point of failure.

B. Availability

MRAQ replicates persistence segments on multiple nodes. The management nodes uses load distribution algorithm to distribute replicates to the compute node. The master node sends the query to the first node which contains the segment the query needs.

Since the memory segment is changeable, the memory segment has a different replication mode. Multiple memory query nodes can read from the same message bus and data stream, each node maintains a unique offset and consumer id, which would create multiple replicates of a memory segment. This is different from multiple memory query node reads from the same message bus and data stream, and shares the same offset and consumer id, which would generate multiple segment partitions. If the memory query node fails, it can reload any indexes persisted to disk and read data from the message bus from the offset point which is submitted the last time.

If a compute node fails and becomes unavailable, it will be deleted from the temporary znodes created on Zookeeper. The coordinator node will notice that the replicates of some segments are insufficient or already lost. Additional replicates will be created, and redistributed throughout the cluster.

MRAQ discovers whether memory segment can be copied by coordinator node and automatically create additional memory query node as a redundant backup. The management node and master node have redundant backup when the primary one fails. The backup node is usually idle until it is reminded by Zookeeper to take the responsibility of the failed primary.

C. Scalability

MRAQ adds and deletes nodes through the initiation and termination of Java processes, so the overhead of adding nodes in batch is small. Scaling up and down the cluster by in proportion is usually done one node at a time with some waiting time between shutdowns. This allows coordinator have sufficient time to redistribute the segment load and create the additional replications. Shutting down nodes in batch is not recommended because it may destroy all copies of a segment to lead data loss.

TABLE III
THE QUERY STATEMENTS

No.	Query
1	SELECT count(*) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$?
2	SELECT count(*), sum(metric1) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$?
3	SELECT count(*), sum(metric1), sum(metric2), sum(metric3), sum(metric4) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$?
4	SELECT high_card_dimension, count(*) AS cnt FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$? GROUP BY high_card_dimension ORDER BY cnt limit 100
5	SELECT high_card_dimension, count(*) AS cnt, sum(metric1) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$? GROUP BY high _card_dimension ORDER BY cnt limit 100
6	SELECT high_card_dimension, count(*) AS cnt, sum(metric1), sum(metric2), sum(metric3), sum(metric4) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$? GROUP BY high_card_dimension ORDER BY cnt limit 100

IV. Experimental Evaluations

To test the performance of MRAQ, 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).

- The scope of timestamp of queries covers all data;
- Each machine has 16GB of RAM and 1TB of disk and 16 cores. The machine is configured to use 15 threads to process queries;
- A memory-mapped storage engine is used.

Data is firstly divided on the time stamp, and then on dimension value to create thousands of segments, each segment is about 10,000 lines. Testing benchmark cluster contains 6 compute 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. The query statements in Table III describe the purpose of each query.

Figure 5 shows the cluster scanning rate, and Figure 6 shows the core scanning rate. In Figure 5, we find the results of the expected linear scaling based on the result of the 5 nodes cluster. In particular, we inspect the performance of the marginal revenue decreases with the scale of the cluster increasing. Under the expected linear scaling; Query 1 on a cluster with 55 nodes would achieve scanning rate of 37 million rows per second. In fact, the scanning rate is 26 million rows per second. However, the Query 2-6 keep a linear speedup until up to 30 nodes, while in Figure 6 the core scanning rate of the query remains almost stable. According to the Amdahls Law, the increase of the speed of a parallel computing system is often limited by the time requirements for the sequential operations of the system. In Table III, the first query is a simple counting, achieving scan rate of 330 thousands lines per second per core.

In fact, we consider that the cluster with 55 nodes is actually over provisioned for the test datasets, which explains the growth is slower than the cluster with 30 nodes. Concurrency model of MRAQ is based on the segment: one thread scan a segment. If the number of segments on a node modulo the number of cores is small (such as 17 segments and 15 cores), during the last round of calculation, some of the core will be idle. When more aggregation metrics are added, we find performance degrade. This is because MRAQ uses a column-oriented storage format. For the count (*) query, MRAQ has to check timestamp column to determine whether it satisfies the "where" clause. When we add metrics, MRAQ has to load those metric values and scan over them, which takes up the memory being scanned.

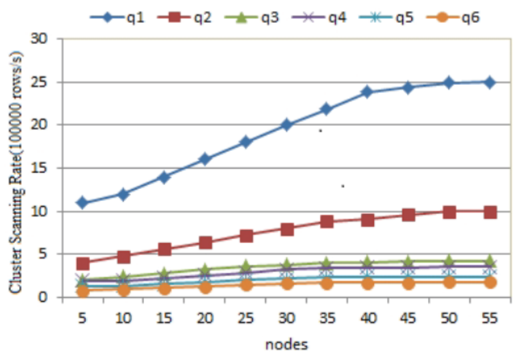

Fig. 5. The cluster scanning rate

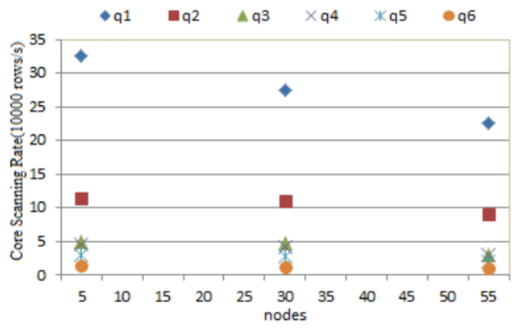

Fig. 6. The core scanning rate

V. Conclusion

In this paper, we propose a memory-based real-time aggregation query system named MRAQ, which adopts the shared-nothing architecture to support the scalability. The experiments show the MRAQ has good performance on online aggregation queries. In future, we will consider the segment assignment, and plan to design a cost-based optimization assignment algorithm.

Acknowledgment

The research work is supported by the National Key R&D Program with (No.2016YFB081304, 2016Y650031105), National Natural Science Foundation of China (No. 61402464),Youth Foundation of National Computer Network Emergency Response Technical Coordination Center (No.2016QN-19)

References

[1] X. Wu, X. Zhu, and W. G. Q. et al, “Data mining with big data,” *IEEE transactions on knowledge and data engineering*, vol. 26, no. 1, pp. 97–107, 2014.

[2] X. Meng and X. Ci, “Big data management: Concepts, techniques and challenges,” *Journal of computer research and development*, vol. 50, no. 1, pp. 146–169, 2013.

[3] G. Jim and G. Goetz, “The five-minute rule ten years later, and other computer storage rules of thumb,” *ACM Sigmod Record*, vol. 26, no. 4, pp. 63–68, 1997.

[4] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM Sigmod Record*, vol. 44, no. 2, pp. 35–40, 2010.

[5] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Analysis of hdfs under hbase: a facebook messages case study,” in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, 2014, pp. 199–212.

[6] D. Congjin, G. Haodong, and Y. Qiu, “Bigtable: A distributed storage system for structured data,” in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[7] D. Giuseppe, H. Deniz, J. Madan, K. Gunavardhan, L. Avinash, P. Alex, S. Swaminathan, V. Peter, and V. Werner, “Dynamo: amazon’s highly available key-value store,” *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[8] B. Fitzpatrick and V. Anatoly, “Memcached: a distributed memory object caching system,” *ACM SIGOPS operating systems review*, pp. 1–4, 2011.

[9] L. Seung-Ho and K.-H. Park, “An efficient nand flash file system for flash memory storage,” *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 906–912, 2006.

[10] G. Vincenzo, J.-P. Ricardo, P.-M. Marta, S. Claudio, and V. Patrick, “Streamcloud: An elastic and scalable data streaming system,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[11] L. W, L. L, P. S. T. S, and et al, “Muppet: Mapreduce-style processing of fast data,” *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1814–1825, 2012.

[12] K. Shvachko, H. Kuang, S. Radia, and et al, “Analysis of hdfs under hbase: a facebook messages case study,” in *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–10.

[13] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, “Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters,” in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, 2012, pp. 1–6.

[14] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, “The case for ramclouds: scalable high-performance storage entirely in dram,” *Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2009.

[15] S. Chintapalli, D. Dagit, and R. Evans, “Pacemaker: When zookeeper arteries get clogged in storm clusters,” in *Proceedings of the IEEE 9th International Conference on Cloud Computing*, 2016, pp. 448–455.

[16] J. Kreps, N. Narkhede, and J. Rao, “Kafka: A distributed messaging system for log processing,” in *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB)*, 2011.

[17] C. S, L. D, K. O, and et al, “Better bitmap performance with roaring bitmap,” *Software: practice and experiences*, pp. 1–11, 2015.