Improving the Performance of Aggregate Queries with Cached Tuples in MapReduce

Dunlu Peng, Kai Duan and Lei Xie

Shanghai Key Lab of Modern Optical System, School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai 200093, China
dunlu_peng@163.com, duankai@yeah.net, leixie2007@126.com

Abstract

As an essential approach for extracting valuable summarized information from massive data set, aggregate query plays important roles for data-intensive applications in cloud computing. As a popular cloud computing platform, MapReduce is a promising paradigm for processing massive data. However, executing aggregate query over massive data sets is very time-consuming and it is also inefficient to run aggregate query directly on MapReduce platform. In order to process an aggregate query efficiently, this work proposes a cache-based approach for improving the performance of aggregate queries on MapReduce platform. This approach enhances the performance of processing aggregate queries on MapReduce platform by caching the pre-processing results before executing the aggregate query. The pre-process results are partitioned into different parts which are cached on different nodes in the cluster. Some strategies are presented to maintain the cached tuples when the original data changes. The experimental results demonstrate that the proposed approach has better performance compared with some existing cache managing approach, such as LRU and LFU.

Keywords: *Aggregate query, MapReduce, LRU, cloud computing, massive data set*

1. Introduction

As an essential approach for extracting valuable summarized information from massive data set, aggregate query plays very important roles for data-intensive applications in cloud computing. The exponential increasing size of data has brought a big challenge to run aggregate queries efficiently against the data. It is because executing the aggregate queries is very time-consuming [1]. Answering the queries in appropriate time puts forward high demands for both hardware and software, which probably needs new investment. Cloud computing is regarded as a promising computing diagram for processing massive data set with low price. It refers to the use of large-scale computer clusters which is often built with low-cost hardware and network equipment [2]. Though MapReduce has been proposed for processing large-scale data [3], we cannot efficiently implement aggregate queries on the platform without any extension.

At present, how to improve the efficiency of aggregate queries has become a highlight both in academic and industrial areas. An OLA system model based on MapReduce platform was presented to estimate the bounds for searching accurate results for queries [4]. In [5], a pipeline was added between *Map* phase and *Reduce* phase in MapReduce model, so that the *Reduce* phase could begin before the complete results being returned from the *Map* phase. By this means, the performance was improved a lot for a certain size of data. In a data-intensive application, when the data collection is increased from tens of terabytes to petabytes, this

approach cannot assure the efficiency of ad-hoc query processing. Generally, online analysis needs to get the summarized data aggregated from fine-grained data, such as the real-time transaction data from an online shop. Therefore, the performance cannot be guaranteed for the computation if there is no any pre-processing before executing aggregate queries over massive fine-grained data set.

In order to improve the performance, in our work, we execute some pre-processing and cache the results on different cluster nodes. When computing some aggregate queries, the cached aggregated data can be accessed directly instead of being computed online, thus the queries can be accomplished in an efficient way. In general, we often execute aggregate queries against relational database. As we know, relational database is unsuitable to dispose real-time applications. This paper mainly discusses processing aggregate queries over massive set of key-value data which can be easily used to process real-time data, such as MongoDB. The aggregate query algorithms are implemented on MapReduce platform, in which we cache the results on different MapReduce nodes. We develop an algorithm to implement the strategy on accessing the cached data efficiently and effectively replacing cache as well.

The following sections are organized as follows: Section 2 presents the related work. Section 3 shows the framework of our proposed query model. The aggregation algorithms based on the MapReduce programming model for processing the aggregate queries are described in Section 4. Section 5 discusses how to update the cache's content and the relevant algorithms are also presented in the same section. We verify the performance of our approach with experiments in Section 6. Finally, we draw the conclusions and present our future work in Section 7.

2. Related Work

Our goal of this work is to improve the performance of executing aggregate queries against massive size of data on the MapReduce platform. Recently, there has been lots of research work aiming to improve the performance of MapReduce. In [8], the authors suggested using distributed memory to cache data both at *Map* and *Reduce* phases. At *Map* phase, the data was written into the distributed memory cache and at *Reduce* phase the corresponding data was read from the distributed memory cache. This work was similar to that of [5] which we mentioned in Section 1. A new scheduler was designed to improve the performance of Hadoop running on heterogeneous clusters in [9]. A shared-memory online MapReduce framework and its further modification were proposed to handle algorithms which required multiple MapReduce phases in [10]. However, with respect to the aggregate queries on massive data set, the above approaches cannot be exploited directly to process them efficiently. Some recent work focuses on online aggregation, such as the approaches presented in [4, 5, 11]. It's known to us that generally the total size of data on the MapReduce cluster is very large so that the aggregation operations are not easy to be accomplished with time-consuming computation. In our approach, we discuss how to store the results and cache part of the data to improve the whole performance of a query.

A crucial issue for cache-based aggregate query is how to replace the cached data. Inspired by the mechanism employed in the materialized view selection, in our approach, we use some of its strategies to replace the cached data. A materialized view caches a pre-computed summarized data which transparently allows users to query summarized data rather than the original data [12]. In [13], the authors provided a way to select access control rules attached to the materialized views which were based on access control rules on the elementary tables. However, they resorted to the basic bucket algorithm which cannot be used to derive all relevant access control rules. The traditional cache replacement policies, such as LFU, LRU,

and RAND and so on, only take some single factors into consideration. For example, the LRU-K policy was a variant of LRU policy considering the last access time and access frequency of a document [14]. Two parameters *K* and *RP* have to be determined in the LRU-K policy. *K* refers to the last *K* access time of every document. With this approach, the data with access frequencies less than *K* would be replaced firstly. Subsequently, the LRU-K policy replaces documents that have not been requested recently. If the time for the eviction of a document is greater than *RP*, the access tuples of the document will be deleted. However, in MapReduce, for the large size of data, the access delay time should be taken into consideration when trying to maintain the cache efficiently.

3. Overview of the Model

As we aforementioned, this work focuses on improving the performance of processing aggregate queries against massive data set. We implement our approach by allocating caches to store the results of aggregate queries on different MapReduce cluster nodes. Figure 1 depicts the model compliant to the approach and we give an overview of the framework as follows.

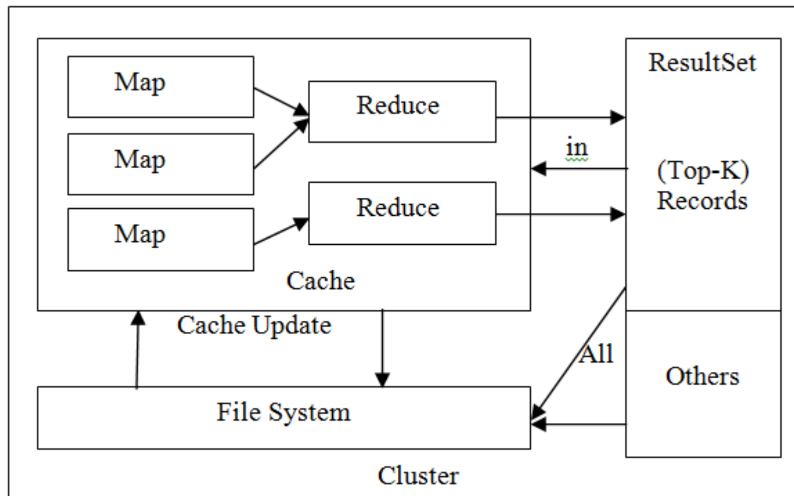


Figure 1. Cache Allocation on MapReduce Node

After allocating the caches on the MapReduce cluster's nodes, we run the preloaded aggregate algorithms against the data stored on the MapReduce. The top-*k* results (for example, the first *k* greatest numbers in the output of *Count*( ) operation) are picked on each node in descending order with respect to the predicates of queries and cached on different cluster nodes. We refer this step as caches' initialization. Next, we present how to accomplish an aggregate query based on the cached results.

When a MapReduce cluster's node receives an aggregate query, the system firstly searches the results cached on each node distributed in the cluster. If the result is obtained successfully, it will be returned immediately and a message will be sent to other nodes. The message informs other nodes that the result has be found, so the search process can be stopped. Otherwise, the system will have to fetch the data stored on disks and compute online. However, fetching data from the disks needs time-consuming I/O operations which will make the entire system inefficient. In order to make the system more applicable, we propose a strategy for replacing the cached data.

Assume there is a large size of data already stored in the cluster and distributed on different *Map* data nodes. The cluster has m nodes which are responsible for *Map* tasks and r nodes which are in charge of *Reduce* tasks, and $r<<m$. Therefore, in the cluster we have n (where $n=r+m$) nodes need to allocate the caches. The size of the cache should be allocated on a node depends on the running state of the node. Let's suppose the average size of cache on each node is s so the total size of cache in cluster is $s\times n$. Without losing generality, we suppose the stored data objects have k attributes and are denoted as $O(A_1, A_2, \ldots, A_k)$. As we know, it does not make sense to run aggregate operations on some attributes, such as the student ID. Therefore, we build a list *Alist* to contain the attributes which are probably aggregated on. After aggregate operations are carried out on every attribute, the results will be cached as the initial value.

4. Implementation of Aggregate Query in MapReduce

4.1 Aggregation over Relational Data Set

For simplicity, we take a relational table as an example which is shown in Table 1. From the example table, we know that executing aggregate operations on *gender*, *age* and *class* is making sense, but not on *number* and *name*. Therefore, given a relation $R(A_1, A_2, \cdots, A_i, \cdots, A_k)$, we create an attribute set *AList* to list attributes which the aggregate operations can be meaningfully executed on. For Table 1, *AList={gender, age, class}* and their possible values are *gender={male, female}, age={21,22}*and *class={1,2}*, respectively. After that, aggregate operations are implemented on each attribute contained in *AList* and the results will be cached on different nodes.

Table 1. Example Table

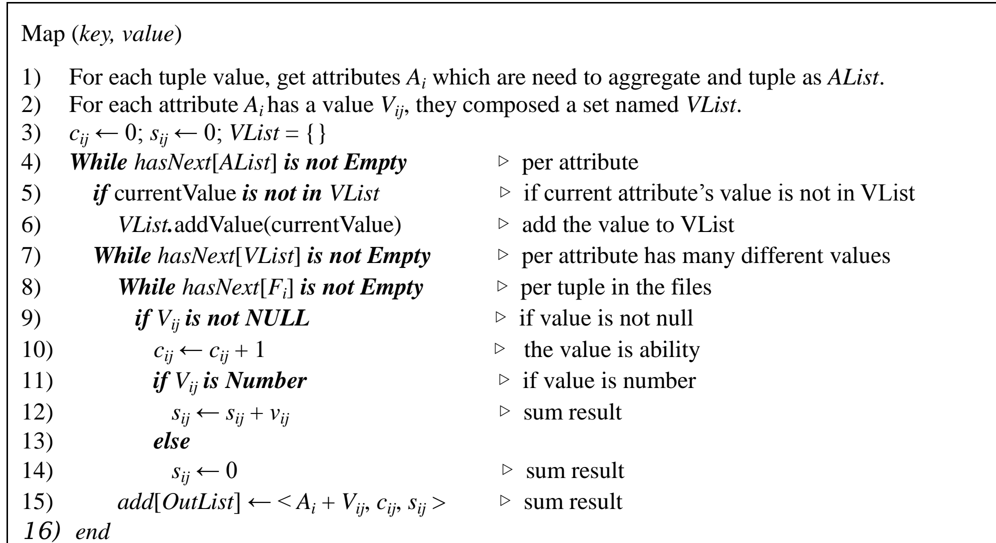
number	name	gender	age	class
110711	Tom	male	21	1
110712	Jerry	female	22	2
110713	Tony	male	21	1
110714	Jone	male	21	2
110715	Maly	female	22	2

4.2 Aggregate Query in MapReduce

In Section 3.1, we discussed executing aggregate operations on relational table. Data processed under MapReduce environment is in <*key, value*> style. In our system, the data we use are like those shown in Table 1. For a given tuple t of the table, we take data of t as the *value*, and t's position as the *key*. Assume that the cluster's *Map* nodes and *Reduce* nodes are deployed on different nodes. On *Map* nodes, aggregate operations are executed on single node, while *Reduce* nodes combine all the results returned from the *Map* nodes of the whole cluster. In a *Map*( ) function, the attributes which should be aggregated are stored in a list *AList*. We take two aggregate queries *Count*( ) and *Sum*( ), as the examples. We run them on each attribute contained in *AList* over the entire file. Of course, the *Sum*( ) function is only run on numeric attributes. In the *Reduce*( ) function, the results are added up to form the final result which is output to the disk.

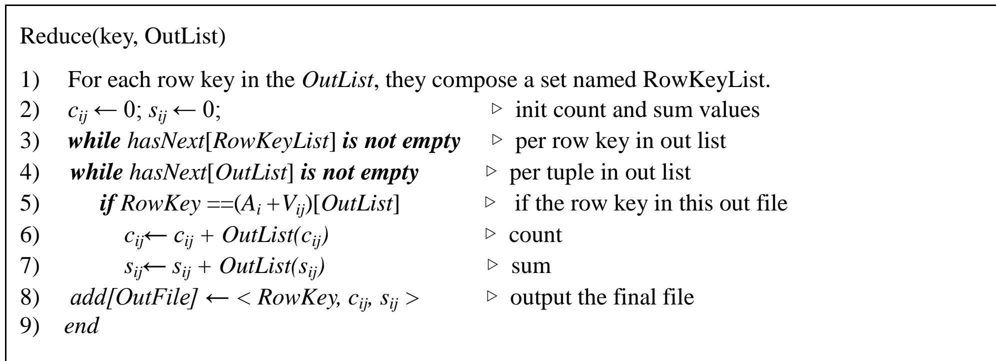
Suppose that the original data is distributed on different *Map* nodes of MapReduce cluster. That is, the data is partitioned into n data files, denoted as $f_1, f_2, f_3, \ldots, f_n$, and stored on the *Map* nodes $d_1, d_2, d_3, \ldots, d_n$, respectively. The *Map*( ) function is shown in Algorithm 1 and the *Reduce*( ) function is shown in Algorithm 2. For Algorithm 1, as we know, *AList* is a subset of attributes and its size is much smaller than the number of data tuples. The domain of

each attribute is also finite and the number of different values of the attribute is also less than the number of data tuples. According to the analysis, we know that the time complexity of *Map( )* function (see Figure 2) is *O* (*n*). Figure 3 presents the *Reduce( )* function whose the output tuples are $\sum A_iV_j$ for A_i aggregation attributes and V_j values, and its complexity is also *O* (*n*).



```
Map (key, value)
1)  For each tuple value, get attributes A_i which are need to aggregate and tuple as AList.
2)  For each attribute A_i has a value V_ij, they composed a set named VList.
3)  c_ij ← 0; s_ij ← 0; VList = { }
4)  While hasNext[AList] is not Empty              ▷ per attribute
5)    if currentValue is not in VList              ▷ if current attribute's value is not in VList
6)      VList.addValue(currentValue)               ▷ add the value to VList
7)    While hasNext[VList] is not Empty            ▷ per attribute has many different values
8)      While hasNext[F_i] is not Empty            ▷ per tuple in the files
9)        if V_ij is not NULL                      ▷ if value is not null
10)         c_ij ← c_ij + 1                        ▷ the value is ability
11)         if V_ij is Number                      ▷ if value is number
12)           s_ij ← s_ij + v_ij                   ▷ sum result
13)         else
14)           s_ij ← 0                             ▷ sum result
15)     add[OutList] ← < A_i + V_ij, c_ij, s_ij >  ▷ sum result
16) end
```

Figure 2. Map( ) Function



```
Reduce(key, OutList)
1)  For each row key in the OutList, they compose a set named RowKeyList.
2)  c_ij ← 0; s_ij ← 0;                        ▷ init count and sum values
3)  while hasNext[RowKeyList] is not empty     ▷ per row key in out list
4)  while hasNext[OutList] is not empty        ▷ per tuple in out list
5)    if RowKey ==(A_i +V_ij)[OutList]         ▷ if the row key in this out file
6)      c_ij← c_ij + OutList(c_ij)             ▷ count
7)      s_ij← s_ij + OutList(s_ij)             ▷ sum
8)  add[OutFile] ← < RowKey, c_ij, s_ij >      ▷ output the final file
9)  end
```

Figure 3. Reduce( ) Function

With the above two algorithms, we can get the simple aggregate results with format as <*RowKey*, $C_{i,j}$, $S_{i,j}$ >, where *RowKey* is formed with the attribute and its value, $C_{i,j}$ is count of the tuples whose attribute A_i is value V_j and $S_{i,j}$ is the sum of the corresponding attributes of all tuples. For example, in table1, the final results are <*gender male, 3, null*>, <*gender female, 2, null* >, <*age 21, 3, 63*>, <age 22, 2, 44>, <class 1, 2, null>, <class 2, 3, null>, where null means that *Sum*( ) cannot be executed on the corresponding attribute. Consider the following SQL statement,

Select count (*) from Table1 where class = 2 and age = 22 (SQL 1)

which is obviously trying to get the number of students whose age is 22 in Class 2. This kind of aggregate queries is often met in real applications. However, it's non-trivial to obtain the

results using the above approach. Therefore, we extend the format by adding the tuple ID list *Keylist* at the end of the final results, that is, $<RowKey, C_{ij}, S_{ij}, KeyList>$. Thus, the final results become as *<age 21, 3, 63, (110711, 110713, 110714)> ,<age 22, 2, 44, (110712, 110715)> <class 1, 2, , (110711, 110713)>, <class 2, 3, , (110712, 110713, 110715)>* . To answer SQL1-like queries, we can find, *KeyList*, all IDs of the students in Class 2 and those of the students whose age is 22, respectively. As we can see the *KeyList* of the former are (110712, 110713, 110715) and that of the latter are (110712, 110715). By intersecting the two *KeyLists*, it is easy to get the final results 110712 and 110715.

5. Cache Management

5.1 Initializing the Cache

After executing aggregate computation with *Map*( ) function (see Figure 2) on MapReduce, we get the number of tuples with respect to a certain value of a given attribute in a node files. If the attribute is numeric, the *Sum*() can be conducted, *i.e.*, $List<A_i+ V_{ij}, C_{ij}, S_{ij}>$. The return values are sorted by their count in descendant order. The top k tuples are chosen and cached in the cluster, where the value of k is corresponding to the size of cache. Assume there are n allocated cache nodes and the average size of cache on each node is a, obviously, the total cache size S in the cluster is:

$$S=n \times a \qquad (1)$$

Let the size of tuple in $List<A_i+ V_{ij}, C_{ij}, S_{ij}>$ be l and k is computed as :

$$k=S/l \qquad (2)$$

Generally, the number of selected aggregated tuples at each node is not more than k and the exact number is determined at runtime.

5.2 Algorithm for Updating the Cache

Traditional popular cache updating algorithms [6, 7], such as LRU, LFU, SIZE and so on, only take some single factors into consideration which make them replace the cache in an inefficient way. Here, we propose an approach to estimate *the value degree* $Value_i$ of a cache tuple i:

$$Value_i=F_i \times T_i/(Tc_i-Tl_i) \qquad (3)$$

where F_i is the frequency of tuple i *being accessed*, T_i is the delay time of fetching tuple i from the disk, Tc_i is current time and Tl_i is the last access time. From Formula 3, we observe that $Value_i$ increases as the access frequency F_i and delay time T_i increase and decreases as the interval of fetching data (Tc_i-Tl_i) *increases*. The following updating algorithm is based on the concept of *value degree* formulated in Formula 3.

The data stored on a node changes continuously in the practical context, in order to timely cache the aggregated data it needs to regularly update the cache. During the update, we replace the last n tuples ranking with the *value degree* computed with Formula 3. The last n tuples cached on the nodes but seldom used in aggregate computation. These tuples are replaced by some other data stored on the disk. This operation is executed during the process of periodic update. Figure 4 describes the process of updating cache data.

```
Algorithm: ALG3—update cached tuples
Input: aggregation query Q;
Output: query result R;
 1)UpdateCache(Q)
 2)while hasNext[cache] is not empty          ▷ search all tuples in cache
 3)   if key[Q] == key[cache(i)]              ▷ if find
 4)       R ← cache(i)
 5)       Getback(R)                          ▷ return result to client
 6)       F[cache(i)] ← F[cache(i)] + 1       ▷ access frequency add 1
 7)       Tl[cache(i)] ← currentTime          ▷ update access time
 8)       Return                              ▷ search finish
 9)while hasNext[file] is not empty           ▷ can't find in cache, start search in disk file
10)   if key[Q] == key[file(i)]               ▷ got it
11)      R ← file(i)
12)       Getback(R)                          ▷ return result to client
13)       DeleteFromCache(random(min(Vi)))    ▷ delete the min value in the cache
14)       cache(i) ← InsertIntoCache(file(i))
15)       F[cache(i)] ← 1                     ▷ access frequency add 1
16)       Tl[cache(i)] ← currentTime
17)       Return                              ▷ search finish
18)end
```

Figure 4. Algorithm for Updating Cached Data

5.3 Maintenance of the Cache Coherency

High fault-tolerance and scalability are the two main characteristics of MapReduce platform. Thus, let's see what will happen when a new node is added to or an existing node is removed from the cluster. When a new node is added to the cluster, the system will process aggregation operation over the data and cache the result on the new node if the tuples are not in the cached results. Otherwise, the cache will be updated and the corresponding data files on disk are updated simultaneously.

In MapReduce platform, data files are duplicated on different nodes, so it is unnecessary to worry about the loss of data files when a node is failure. However, how to recover the lost cached data on a failure node is a critical issue. Generally, one possible solution to address this issue is to cache the same data on a backup node and put the backup node into the cluster as a standby. If a node is failure, its standby will join the cluster soon and take the place of it. There is an obvious shortage of this approach, that is, we need to double the nodes which will cause too many redundant nodes in the cluster. To overcome this disadvantage, in our system, we add a flag in the aggregation result files to label whether a tuple has already been cached. If a node is failure, all we need to do is to find the backup of data files on the failed node in the cluster and re-cache it according to the flag. In this way, the cached data can be recovered in a short time.

Now, we consider the update of a single tuple. If there some tuples are deleted, inserted or updated, we need to update both the corresponding aggregated data file on the disk and the cached data on the cluster node. For deleting, we find the tuples with the key composed of the name and the value of the attributes, and update the aggregated data with corresponding operations, such as minus 1 for the result of *Count*( ) and the value of numeric attributes from

the result of *Sum*( ). Then, the updated data will be saved on the disk; meanwhile, if the data is cached, the cached data is also be updated. For updating, only the values of aggregate operations, such as S*um*( ) and *Average*( ), based on the numeric attributes need to be updated. For inserting, we need to do opposite operations for deleting, such as add 1 to the result of *Count*( ) and add the new values to the result of S*um*( ). All updated data are saved on disk and cached on corresponding nodes.

6. Experimental Evaluation

To investigate the performance of our proposed approach, we conducted two series of experiments. One is to verify the system performance by measuring the delay time of accessing the data file on the disk and the other is to evaluate the efficiency of updating cache algorithm (see Figure 4, the algorithm is denoted as ALG3) by comparing it with LRU and LDU. We ran the experiments on 40 virtual nodes each of which was with Linux operating system and its version was CentOS 5.8, 800M Memory, and the number of physical machines are 20 and the operating system was Windows 7 Ultimate 32-bit, with AMD Athlon ™ 64 × 2Dual Core Processor 4000 2.10GHz, 2G Memory. The dataset was generated with a simulating system and the whole system was implemented in Hadoop platform.

6.1 File Access Latency

When the system cannot find the answer for a query request, it tries to search it in data file on the disk. Under such circumstance, the position of the queried tuples in a data file is the main factor affecting on the response time.

Figure 5 shows how the response time varies with the position of tuples. The horizontal ordinate is the data scale measured as logarithm of the number of tuples; and the vertical coordinate is the response time in seconds. From the figure, we observed that when there are 10 million tuples, response time of the first tuple and the last tuple differs with 16.69 seconds. The delay time increases linearly with the size of data file. Let's consider what will happen when the data scale is 200M and some of the tuples are cached. Assume that both the first tuple and the last tuple satisfy the LRU replacing condition. If the replacing operation is operated on the last tuple instead of the first tuple, the delay time of the next query for searching the last tuple is about 16.69 seconds which is longer than those needed for searching the first tuple. This period is two times longer than the I/O operation of 200M file. However, if the replace operation is operated on the first tuple, the next query for searching the first tuple time only needs one I/O operation time. To guarantee the performance of access data files, we rank the data though evaluating the *value degree* of the tuples and replace the tuples with minimal *value degree* in the cache.

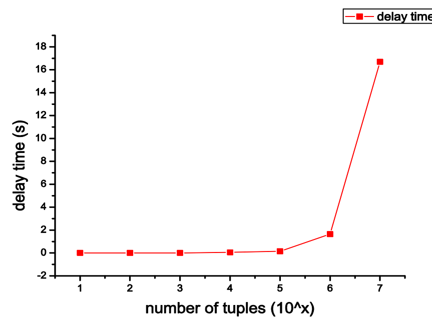
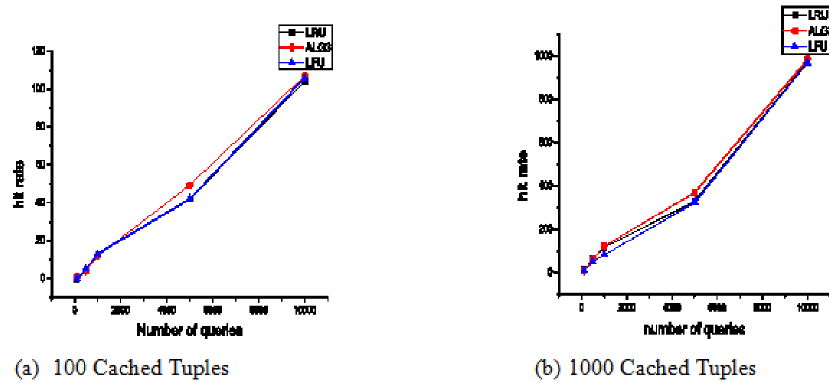


Figure 5. Delay Time for Query

6.2 Comparison of Hit Rate

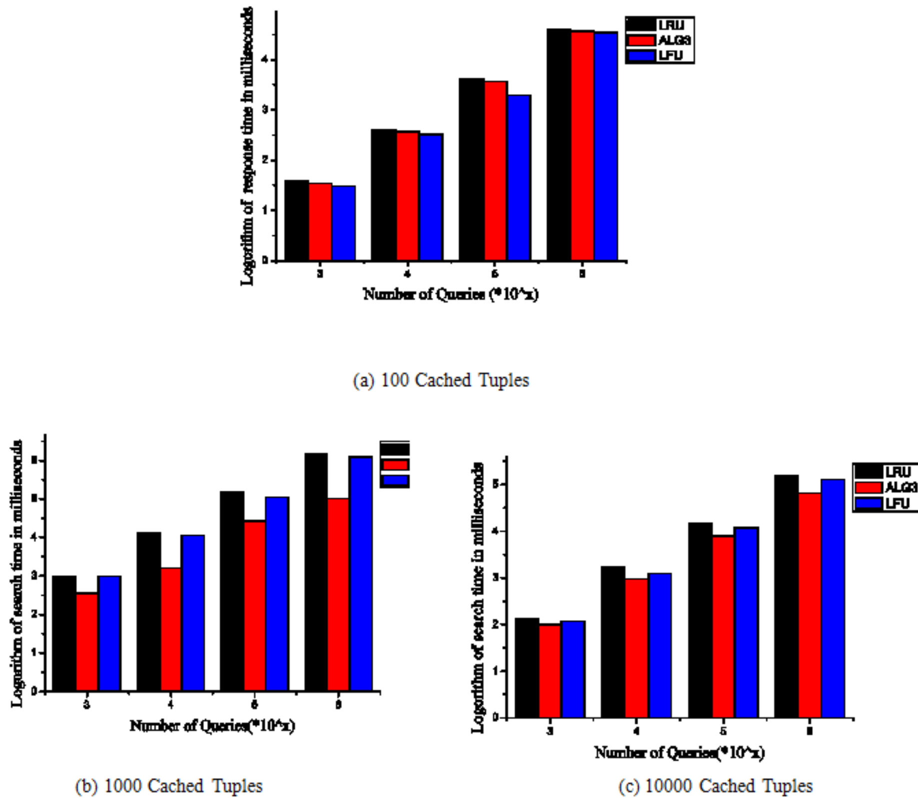
We compare the hit rate of ALG3 with that of LRU's and LFU's. The aggregate results are got from the Hadoop platform, and we divided the result file into 40 small files, each of which contains 1 million tuples. The small files are separately placed on 40 different virtual nodes.

Figure 6 shows the hit rate (count) of LRU, LFU and ALG3 when the number of cached tuples is 100 and 1000. During the experiment, we generated the query by random. From the figure, the three algorithms get the hit rate similar, even ALG3 a little better.



(a) 100 Cached Tuples (b) 1000 Cached Tuples

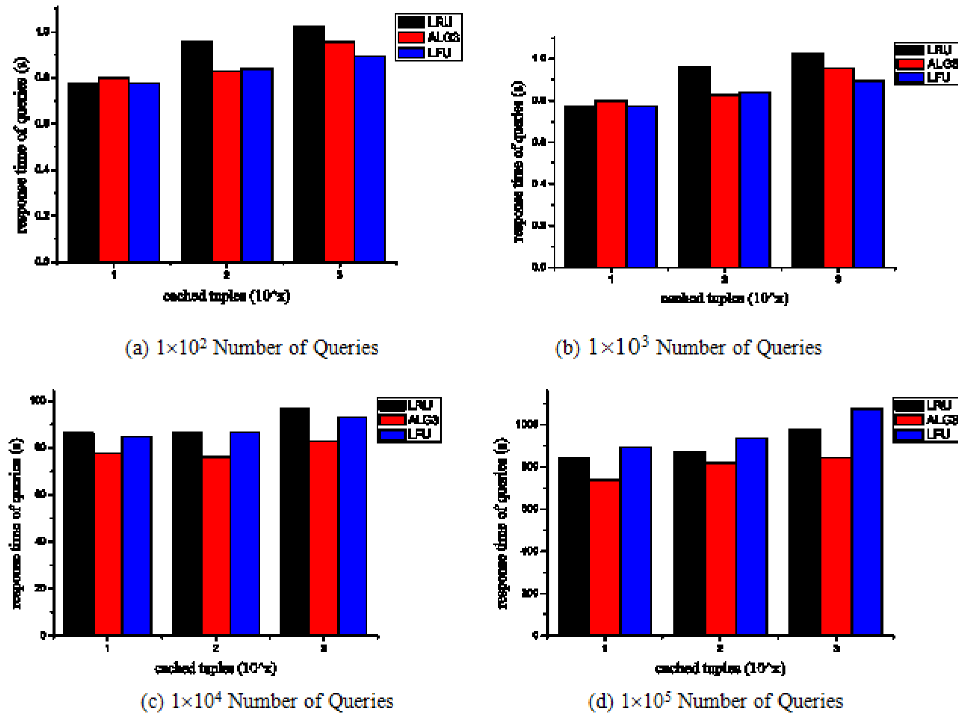
Figure 6. Comparison of the Hit Rate of Different Algorithms



(a) 100 Cached Tuples

(b) 1000 Cached Tuples (c) 10000 Cached Tuples

Figure 7. Comparison of Search Time of Different Algorithms



(a) 1×10^2 Number of Queries

(b) 1×10^3 Number of Queries

(c) 1×10^4 Number of Queries

(d) 1×10^5 Number of Queries

Figure 8. Comparison of Average Response Time of Queries using Different Algorithms

6.3 Comparison of Scalability

To verify the scalability and performance of our searching algorithm, ALG3, an experiment was conducted to investigate the time of searching cache of ALG3, LRU and LFU. The cached tuples were set to 100, 1000 and 10000. For each number of cached tuples, we executed 1×10^3, 1×10^4, 1×10^5, 1×10^6 queries with the three algorithms, respectively.

Figure 7 shows the experimental results. The figures illustrate that the ALG3 has better scalability compared with LRU and LFU. From Figure 7, we can see when the cached tuples are 100, ALG3 takes more time than LFU and less time than LRU to search the cache. When the cached tuples are 1000 and 10000, ALG3 spends less time than both of the other two algorithms LFU and LRU. This observation demonstrates that ALG3 has better scalability than the other two algorithms.

6.4 Comparison of Average Response Time

Response time is an important measure for query performance. In our experiments, we demonstrated the performance of ALG3 by comparing the average response time of query using ALG3 with that of using LRU and LFU. The average response time was measured for different number of queries 1×10^2, 1×10^3, 1×10^4, and 1×10^5 and the cached tuples were set to 10, 100 and 1000.

Figure 8 depicts that the average response time (ART) changes with different workload. From the figures we find out that the ART increases as the number of tasks (queries) increases whichever algorithm is used. However, ART of ALG3 does not increase as dramatically as that of LRU and LFU. When workload is small, the performance of ALG3 cannot surpass LRU and LFU. While as workload raises, ALG3 performances better than the

other two algorithms. It is because that ALG3 is optimized by the replacing strategies rule discussed in Section 4.2.

7. Conclusions

This paper proposes a cache-based approach to aggregate queries processing on MapReduce platform. Some algorithms are presented to improve performance of aggregate operations by caching the top-*k* aggregate results. When receiving an aggregate query, the system searches the cached results in the cluster and confirms immediately whether the tuples are hit or not. If not, the system will try to fetch the result from the data file in the disk. We also propose a strategy for cache replacement in which the value degree is used to judge which tuples should be replaced. Some experiments were conducted to verify the performance of our approach.

In our future work, we will try to replace the disk files with BigTable [15] which was developed to manage large data set efficiently in memory. We will also improve the system performance by building cached index [16]. However, building cached index will cause some extra memory cost, how to balance the memory cost and the performance is another challenge.

Acknowledgements

This project is supported by Municipal Nature Science Foundation of Shanghai under Grant (No.10ZR1421100) and Foundation of Core Course Construction of USST-Database Principle B.

References

[1] J. Paulo, B. Carlos and S. A. Paulo, “A Survey of Distributed Data Aggregation Algorithms”, CoRR, vol. abs/1110.0725, **(2011)**.

[2] P. Mika and G.Tummarello, “Web Semantics in the Clouds”, IEEE Intelligent Systems, vol. 23, no. 5, **(2008)**, pp. 82-87.

[3] D. Jeffrey and G. Sanjay, “MapReduce: Simplified Data Processing on Large Clusters”, Proceedings of the 6th Symposium on Operating Systems Design and Implementation, **(2004)** December 6-8; San Francisco, USA, pp. 137-150.

[4] P. Niketan, R. B. Vinayak, J. Chris and C. Tyson, “Online Aggregation for Large MapReduce Jobs”, PVLDB, vol. 4, no. 11, **(2011)**, pp. 1135-1145.

[5] C. Tyson, C. Neil and A. Peter, “Online Aggregation and Continuous Query Support in MapReduce”, Proceedings of SIGMOD, **(2010)** June 6-11; Indianapolis, Indiana, pp. 1115- 1118.

[6] F. Ye, Q. Li and E. Chen, “Benefit Based Cache Data Placement and Update for Mobile Peer to Peer Networks”, World Wide Web Journal, vol. 14, no. 3, **(2011)**, pp. 243-259.

[7] L. Zhou, H. Geng and M. Xu, “An Improved Algorithm for Materialized View Selection”, Journal of Computers, vol. 6, no. 1, **(2011)**, pp. 130-138.

[8] S. Zhang, J. Han and Z. Liu, “Accelerating Map-reduce with Distributed Memory Cache”, Proceedings of IEEE 15th International Conference on Parallel and Distributed Systems, **(2009)** December 8-11; Shenzhen, China, pp. 472-478.

[9] M. Zaharia, A. Konwinski and A. D. Joseph, “Improving MapReduce Performance in Heterogeneous Environments”, Proceedings of the 8th USENIX on Symposium on Operating Systems Design and Implementation, **(2008)** December 8-10; San Diego, California, USA, pp. 29-42.

[10] J. Böse, A. Andrzejak and M. Högqvist, “Beyond Online Aggregation: Parallel and Incremental Data Mining with Online Map-Reduce”, Proceeding of International Workshop on Massive Data Analytics over the Cloud, **(2010)** April 26; Raleigh, NC, USA.

[11] T. Condie, N. Conway and P. Alvaro, “Mapreduce online”, Proceedings of 7th USENIX Symposium on Networked Systems Design and Implementation, **(2010)** April 28-30; San Jose, California, USA, pp. 21-21.

[12] T. V. V. Kumar and M. Haider, “A View Recommendation Greedy Algorithm for Materialized Views Selection”, Communications in Computer and Information Science, vol. 141, **(2011)**, pp. 61-70.

[13] A. Cuzzocrea, M. -S. Hacid and N. Grillo, “Effectively and Efficiently Selecting Access Control Rules on Materialized Views Over Relational Databases”, Proceeding of Fourteenth International Database Engineering and Applications Symposium, **(2010)** August 16-18; Montreal, Quebec, Canada, pp. 225-235.

[14] T. Chen, "Obtaining the Optimal Cache Document Replacement Policy for the Caching System of an EC Website", European Journal of Operational Research, vol. 181, no. 2, **(2007)**, pp. 828-841.
[15] F. Chang, J. Dean and S. Ghemawat, "Bigtable: A Distributed Storage System for Structured Data", Proceedings of 7th Symposium on Operating Systems Design and Implementation, **(2006)** November 6-8: Seattle, WA, USA, pp. 205-218.
[16] L. Chen, B. Cui and L. Xu, "Distributed Cache Indexing for Efficient Subspace Skyline Computation in P2P Networks", Proceedings of 15th International Conference Database Systems for Advanced Applications, **(2010)** April 1-4; Tsukuba, Japan, pp. 3-18.

Authors

Dunlu Peng is a professor of University of Shanghai for Science and Technology, Shanghai, China. He received his Ph.D degree from Fudan University, Shanghai, China in June 2006.He served as a PC member of SCC, EDOC, APSCC, WHICEB, CLOUD, etc. His research interests include Cloud Computing, Web applications, service-oriented computing, XML data management and Web mining.

Kai Duan is a master degree candidate of University of Shanghai for Science and Technology, Shanghai, China. He received his bachelor's degree from University of Shanghai for Science and Technology, Shanghai, China in June 2011. His research interests include Cloud Computing, Service-oriented computing, and Web mining.

Lei Xie is a master degree candidate of University of Shanghai for Science and Technology, Shanghai, China. She received her bachelor's degree from University of Shanghai for Science and Technology, Shanghai, China in June 2011. Her research interests include Cloud computing and Web mining.