IEICE TRANS. INF. & SYST., VOL.E98–D, NO.2 FEBRUARY 2015

LETTER

Reducing I/O Cost in OLAP Query Processing with MapReduce

Woo-Lam KANG[†], *Nonmember*, **Hyeon-Gyu KIM**[††a)], *Member*, *and* **Yoon-Joon LEE**[†], *Nonmember*

SUMMARY This paper presents a method to reduce I/O cost in MapReduce when online analytical processing (OLAP) queries are used for data analysis. The proposed method consists of two basic ideas. First, to reduce network transmission cost, mappers are organized to receive only data necessary to perform a map task, not an entire set of input data. Second, to reduce storage consumption, only record IDs are stored for checkpointing, not the raw records. Experiments conducted with TPC-H benchmark show that the proposed method is about 40% faster than Hive, the well-known data warehouse solution for MapReduce, while reducing the size of data stored for checkpoining to about 80%.

key words: *MapReduce, Hadoop, OLAP, data warehouse, TPC-H benchmark*

1. Introduction

Nowadays, Google's MapReduce (MR) has become a de facto framework for analysis of *big data*, which can be characterized by three properties, including volume, velocity and variety [1]. For example, to analyze accumulated SNS data, Facebook developed Hive [2] on top of Hadoop [3], which is the most popular open-source implementation of MapReduce. Walmart also used MapReduce to find users' interests from a huge amount of social media feeds [4].

On the other hand, much research has addressed the efficiency issue of MapReduce due to frequent local and network I/Os required for fine-grained fault-tolerance [5]. To perform an MR job, an input file in the distributed file system (DFS) is first partitioned into multiple data segments, called *input splits*. A master node picks idle nodes and assigns each one a map or a reduce task. Then, the following two steps are performed.

(1) *Map phase*: Each input split is transferred to a mapper. Each mapper performs filtering or preprocessing input records (key-value pairs). A mapper's outputs are written into its own local disk for checkpointing.
(2) *Reduce phase*: After mappers finish their jobs, reducers read the mappers' outputs through the network, and merges them according to their keys. For each key, an aggregate result is produced.

As shown in the above, MapReduce performs frequent checkpointing to increase fault-tolerance of long-time analysis, which may reduce efficiency significantly. Regarding this, Pavlo et al. [6] showed that Hadoop is 2 to 50 times slower than parallel database systems except in the case of data loading. Anderson and Tucek [7] noted that Hadoop is scalable but achieves very low efficiency per node, less than 5 MBytes per second processing rate.

This paper presents a method to improve performance of job processing in MapReduce. More specifically, the proposed method focuses on how to reduce I/O cost in MapReduce when online analytical processing (OLAP) queries are used for data analysis.

2. Proposed Method

The basic idea of the proposed method is to avoid transmission of unnecessary data as much as possible. For example, consider a data set where the schema is (*A*, *B*, *C*, *D*): *A* is a key, *B* is an attribute used for selection in mappers, and *C* and *D* are attributes used for aggregation in reducers. Then, attributes *C* and *D* are not transferred to mappers in the proposed method, because those attributes are not used for the map task. Currently in the MR framework, all input data is transferred to mappers.

Note that, in this method, mappers' outputs may not have all information necessary to perform a reduce task. This is because mappers do not receive a complete data set. Due to this, reducers should read their inputs from the DFS, not from mappers. At the same time, they must know which records are selected from mappers. To notify reducers as to the records selected, a list of record IDs is generated from mappers and transmitted to reducers.

To support the method, SQL-to-MR translation is required. To translate an SQL query into a corresponding MR program, attributes necessary for map and reduce phases must be identified first from the query. For example, consider a data set shown in Table 1, including logs of stock exchanges. Suppose we want to get the sum of quantities where the stock name is equal to "Samsung"; the sum needs to be calculated for each buyer. This requirement can then be specified as the following SQL query.

```
Q1. SELECT Buyer, SUM(Quantity)
    FROM StockExchanges
    WHERE Stock = "Samsung"
```

Manuscript received July 14, 2014.
Manuscript revised September 30, 2014.
Manuscript publicized October 22, 2014.

[†]The authors are with the Department of Computer Science, KAIST, 291 Daehak-ro, Yuseong-Gu, Daejeon 305–701, Republic of Korea.
[††]The author is with the Department of Computer Engineering, Sahmyook University, 815 Hwarang-ro, Nowon-gu, Seoul 139–742, Republic of Korea.
a) E-mail: hgkim@syu.ac.kr
DOI: 10.1587/transinf.2014EDL8143

Table 1 *StockExchanges*: a data set including logs of stock exchanges.

ID	Stock	Buyer	Seller	Quantity	…
1	Samsung	hkim76	kwl3042	200	…
2	Amazon	jwang23	gleehelp	40	…
3	Apple	sanuk	stockgen55	5	…
4	Samsung	hkim76	jwang23	10	…
5	Samsung	sanuk	parksj	1000	…
6	Apple	parksj	juntrade2	350	…
…	…	…	…	…	…

```
GROUP BY Buyer
```

The attributes for the map phase include (i) a key used for aggregation and (ii) attributes used for record selection. The aggregate key is necessary to generate the list of IDs for the records selected from the map phase. In general, the key becomes an attribute defined in the GROUP BY clause. This is because aggregate results are typically calculated based on the GROUP BY attribute values. On the other hand, selection attributes can easily be identified from the WHERE clause. In the example of Q1, *Buyer* and $Stock$ will be identified as attributes necessary for the map phase.

The attributes necessary for the reduce phase include those that need to be outputted as a final result. The output attributes can be identified from the SELECT clause in the query. For Q1, *Buyer* and *Quantity* are those attributes, so they will used in the reduce function.

After the attributes are identified, an MR program can be generated from the SQL query. The program consists of two MR functions, including map() and reduce(). The following shows map() generated from $Q1$. For simplicity, a pseudo code is used for discussion.

Function *Map*(*rowID*, *record*)
  Parse *record* into (*Buyer*, $Stock$)
  If $Stock$ = ”Samsung”
    Output (*Buyer*, *rowID*)
  End If

Map() has two input parameters: *row ID* and *record*. The former is a logical ID denoting the position where the record is stored in the DFS, while the latter is the record data itself. The function is invoked for each record in run time. When map() is executed, it first parses a given record, and then checks whether the record should be transferred to a reducer. For the record selection, the condition in the WHERE clause is used.

When the selection condition is satisfied, a key-value pair is outputted. As a key, the aggregate key (e.g., *Buyer* for $Q1$) is outputted. As discussed above, when the GROUP BY clause is defined with some attributes in a query, those attributes become the (composite) aggregate key. If there is no GROUP BY clause in the query, it can be viewed as there is a single large group including all input records. In this case, a constant key “null” is outputted as a key.

As a value of the key-value pair, the row ID is outputted in the proposed method. This is distinguished from the original MR approach, where a whole record is outputted as a value from map(). In MapReduce, each mapper’s outputs are stored in its own local disk for checkpointing. From this, storage consumption can be reduced significantly in the proposed method, because only record IDs are stored in the method.

Based on the record IDs generated from mappers, reduce() can identify which records are selected from the map phase. Reduce() is called whenever a distinct key with list of values is outputted from map().

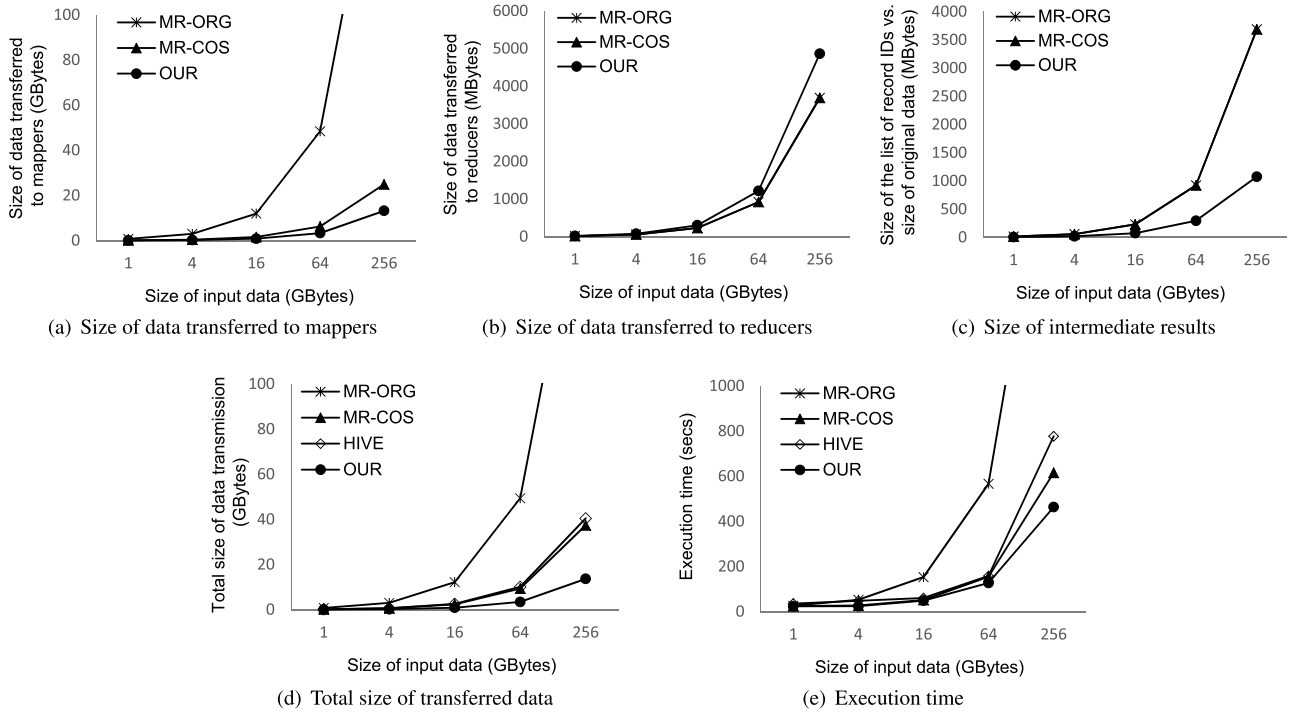
For example, assume that there is only one mapper and all records are fed into it. Then, for each key, map() is invoked and a key-value pair is outputted. For the first record, a key-value pair <“hkim76”, 1> is outputted because it satisfies the selection condition, i.e., $Stock$ = “Samsung”. On the other hand, for the second record, there will be no output. After map() is applied to all records, reduce() is then invoked. It is called for each distinct key generated from map(). For each key k, a list of values with key k is given as an input parameter. In this example, for the selected records with key “hkim76”, reduce() will be invoked with two parameters “hkim76” and the list of row IDs, (1, 4).

The following describes a pseudo code of reduce() generated from $Q1$. The function has two input parameters, *key* and *rowIDs*, denoting an aggregate key and a list of record IDs with the key, respectively.

Function *Reduce*(*key*, *rowIDs*)
  sum_Duration := 0
  records := $getRecordsFromDFS$(*rowIDs*)
  For each *record* in *records*
    Parse *record* into (*Buyer*, *Quantity*)
    sum_Quantity := *sum_Quantity* + *Quantity*
  End for
  Output (*key*, *sum_Quantity*)

Above, $getRecordsFromDFS()$ is a function to read the selected records from the DFS. For each record returned from the function, *Buyer* and *Quantity* are parsed. Then, the sum of *Quantity* is calculated. Note that the postfix “_Quantity” is added to all variables used to calculate the aggregate. In this way, attribute names are added to distinguish various aggregates in the generated function. If aggregates are defined with “*”, no postfix is added.

As discussed above, storage consumption is reduced in the proposed method. This leads to economic benefit. Suppose that the size of input data is 1 TByte, and half of the input records are outputted from the map phase. Then, the size of output records in the original MR approach is 500 GBytes. On the other hand, the size of the record ID list in the proposed method is relatively small. In our experiments, the size is about 25% of the original data, from which the intermediate data size can be calculated as 125 GBytes.



(a) Size of data transferred to mappers

(b) Size of data transferred to reducers

(c) Size of intermediate results

(d) Total size of transferred data

(e) Execution time

Fig. 1 Performance of MR-ORG, MR-COS, Hive, and OUR for TPC-Q1.

Assume we use a commercial cloud storage service which costs about 3 dollars a Gigabyte[†]. In this case, about $1125 can be saved. As the data size increases, more cost can be reduced.

3. Experimental Results

Experiments were conducted over a cluster consisting of 8 nodes; one was a master and the other 7 nodes were slaves. All nodes were equipped with an Intel i5-2500 3.3 GHz processor with 8 GB memory and a 1 TBytes hard disk, running on CentOS 6.4. They were connected via a Gigabit switching hub. As an MR framework, Hadoop 1.2.1 was used. Two map/reduce slots were assigned to each node, and 6 reducers were assigned and run for a job.

For experiments, TPC-H benchmark [8] was employed, which has been widely used for performance evaluation of OLAP queries. Among its queries, TPC-Q1 was used. The proposed method was compared with three systems: MR-ORG, MR-COS, and HIVE. The first two systems are the MR framework without and with column-oriented storage, respectively. The last is a well-known data warehouse solution for Hadoop, developed in Facebook [2]. As the input file format, TextInputFormat (a default input format in Hadoop) was used for MR-ORG. As the column-oriented storage format, CIF was used for MR-COS and our method, while RCFile was used for HIVE. To compare their performances, the following four measures were checked.

(1) Size of input data transferred to mappers
(2) Size of data transferred to reducers (including the list of record IDs in the proposed method)
(3) Size of intermediate results
(4) Total size of transferred data
(5) Execution time

In case of Hive, only the last two parameters were measured, due to difficulty of capturing run-time parameters; to measure internal data sizes (i.e., the first two parameters), mechanisms of query translation and data transmission in Hive need to be fully understood. But without those parameters, superiority of the proposed method can be shown properly.

First, we checked how much data transfer can be reduced by excluding unnecessary attributes in the map phase. In MR-ORG, all input data is transmitted to mappers. On the other hand, in MR-COS, projection can be performed to exclude unnecessary attributes from the data. In TPC-Q1, 7 attributes are used in the query, where their size is about 30% of a record[†]. From this, about 70% of data transfer to mappers can be reduced in MR-COS.

In the proposed method, more data reduction is possible. In TPC-Q1, only 3 attributes are used for the map phase. The size of those attributes is only 6 bytes for each record, where the total size of a record is 140 bytes. Thus, about 95% of input data needs not be transferred to mappers. Figure 1 (a) shows the sizes of data transferred to mappers in the three approaches; the proposed method is denoted as OUR in the figure.

[†]*Amazon EC2* currently provides cloud service with 640 GBytes for about $1800 a year, which works out to about 3 dollars a Gigabyte.

[†]For more details, refer to TPC-H benchmark specification [8].

We also checked the size of data transferred to reducers, whose result is shown in Fig. 1 (b). In MR-ORG and MR-COS, the size is equal to the size of records which are selected from mappers and stored in their local disks. On the other hand, in the proposed method, the list of record IDs is stored in the mappers' local disks and needs to be transferred to reducers. Selected records indicated by the record IDs must also be fetched from the DFS to reducers. Therefore, the size of data transmission slightly increases in the proposed method. The difference only lies in the size of data transmission to fetch the record ID list, because the cost to transfer selected records is the same in all the three methods.

Note that storage consumption decreases in the proposed method. This is because our method only stores the list of record IDs as intermediate results in mappers' disks. Regarding this, Fig. 1 (c) shows the sizes of intermediate results generated from mappers in the three methods. In the result, the size of the ID list was on the average 20% of the data sizes shown in MR-ORG and MR-COS. This implies that the proposed method requires smaller storage than the existing methods.

Figure 1 (d) shows the total size of data transmission in the four methods, which is the sum of the sizes shown in Figs. 1 (a) and (b). In the result, the size of data transmission was smallest in the proposed method. It was about 6% of MR-ORG, and was about 35% of MR-COS and Hive. This shows that the benefit from reducing the size of input data transferred to mappers is much larger than the overhead to transfer the ID list to reducers in our method.

Finally, we measured execution time of each method, where the result is shown in Fig. 1 (e). Figures 1 (d) and (e) together also show that the execution time is proportional to the total size of data transmission. In the execution time, the proposed method also provided the best performance. Its execution time was about 4 times and 25% faster than MR-ORG and MR-COS, respectively. It was also about 40% faster than Hive.

4. Conclusion

In this paper, we proposed a method to reduce I/O cost when processing OLAP queries in MapReduce. In the proposed method, only data necessary to perform a map task are transferred to mappers. Reducers are organized to read their inputs from the DFS, not from mappers. To notify reducers as to information about selected records, record IDs are outputted from mappers, not the raw records.

To support this method, SQL-to-MR translation is required. In this process, attributes necessary for mappers and reducers are first identified from a given SQL query. Based on the information, a corresponding MR program is generated, which consists of map() and reduce(). Map() has a selection condition defined in the WHERE clause of the query, while reduce() includes aggregation functions defined in the GROUP-BY and SELECT clauses.

To see the performance benefit of the proposed method, experiments were conducted with TPC-H benchmark. In the experiments, the method was compared with the original MR approach (adopting the column-oriented storage), denoted MR-COS, and Hive. In the results, the proposed method showed the best performance in both storage consumption and execution time. Compared with MR-COS, about 80% of data stored in mappers for checkpointing was reduced. In case of execution time, the method was 25% and 40% faster than MR-COS and Hive, respectively.

Acknowledgments

This work was partly supported by the R&D program of MSIP/COMPA [2014K000139, Building an education service platform for learning history] and the NRF grant (No. 2011-0016282).

References

[1] S. Madden, "From databases to big data," INTERNET COMPUT, IEEE, vol.16, no.3, pp.4–6, May 2012.
[2] A. Thusoo et al., "Hive - A petabyte scale data warehouse using Hadoop," 2010 IEEE 26th International Conference on Data Engineering (ICDE), pp.996–1005, 2010.
[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp.1–10, 2010.
[4] R. Adduci et al., "Big data: Big opportunities to create business value," Technical report, EMC Corporation, 2011.
[5] K.H. Lee et al., "Parallel data processing with MapReduce: A survey," SIGMOD REC, vol.40, no.4, pp.11–20, 2012.
[6] A. Pavlo et al., "A comparison of approaches to large-scale data analysis," Proc. 2009 ACM SIGMOD International Conference on Management of data, pp.165–178, 2009.
[7] E. Anderson and J. Tucek, "Efficiency matters!," Sigops Oper Syst Rev, vol.44, no.1, pp.40–45, 2010.
[8] M. Poess and C. Floyd, "New TPC benchmarks for decision support and web commerce," SIGMOD REC, vol.29, no.4, pp.64–71, 2000.