International Journal of Database Theory and Application
Vol.10, No.6 (2017), pp.61-70
http://dx.doi.org/10.14257/ijdta.2017.10.6.05

SQL-to-MapReduce Translation for Efficient OLAP Query Processing with MapReduce

Hyeon Gyu Kim

Department of Computer Engineering, Sahmyook University,
Seoul 139-742, Republic of Korea
hgkim@syu.ac.kr

Abstract

Substantial research has addressed that frequent I/O required for scalability and fault-tolerance sacrifices efficiency of MapReduce. Regarding this, our previous work discussed a method to reduce I/O cost when processing OLAP queries with MapReduce. The method can be implemented simply by providing an SQL-to-MapReduce translator on top of the MapReduce framework and needs not modify the underlying framework. In this paper, we present techniques to translate SQL queries into corresponding MapReduce programs which support the method discussed in our previous work for I/O cost reduction.

Keywords*: MapReduce, SQL-to-MapReduce translation, OLAP, I/O reduction*

1. Introduction

MapReduce (MR) has emerged as a popular model for parallel processing of large datasets using commodity machines. By virtue of its simplicity, scalability and fault-tolerance, many enterprises have adopted it for their business analytics applications [1, 2]. By virtue of its simplicity and scalability, many enterprises such as Facebook, IBM, Walmart, Yahoo, and Microsoft were engaged to develop MapReduce-based solutions on massive amount of log data for their business analytics applications [3]. The number of enterprises adopting MapReduce for online analytical processing (OLAP) is growing nowadays.

On the other hand, substantial research has addressed that frequent I/O required to support scalability and fault-tolerance sacrifices efficiency of MapReduce [4, 5]. Regarding this, Pavlo *et al.* [6] showed that Hadoop, the most popular open-source implementation of MapReduce, is 2 to 50 times slower than parallel database systems except in the case of data loading. Anderson and Tucek [7] noted that Hadoop achieves very low efficiency per node, less than 5MBytes per second processing rate. Kang *et al.* [8] showed that I/O cost accounts for about 80% of the total processing cost when OLAP queries are executed under the MR framework.

To address this issue, our previous work [9] suggested a method to reduce I/O cost when processing OLAP queries with MapReduce. In the method, during the execution of a map or a reduce task, only attributes necessary to perform the task are transmitted to corresponding worker nodes. In general, a map task is used for filtering or preprocessing input records, while a reduce task is used to compute aggregation over the records selected from the map task. From this, only attributes participating in the record selection and shuffling are transferred to mapper nodes in the proposed method. Similarly, only attributes over which aggregation is computed are transferred to reducers.

In this paper, we present how to implement the method on top of the MapReduce framework. The method can be implemented by providing an SQL-to-MR translator and needs not modify the underlying framework. In what follows, we first describe our previous work to reduce I/O cost in the MapReduce framework (Section 2) and discuss how the method can be extended to deal with join and multi-query processing (Section 3). We

ISSN: 2005-4270 IJDTA

then discuss several rules to translate given SQL queries into their corresponding MapReduce programs in our method (Section 4). The paper is concluded with a brief mention of future work (Section 5).

2. Preliminaries

To perform an MR job, an input file is first loaded in the distributed file system (DFS) and is partitioned into multiple data segments, called *input splits*. A master node picks idle nodes and assigns each one a *map* or a *reduce* task. The following two steps are then performed.

- *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 (also, in the form of key-value pairs) are written into its own local disk for *checkpointing*.
- *Reduce phase*: After all mappers finish their jobs, reducers pull the mappers' outputs through the network, and merges them according to their keys. The record grouping is called *shuffling* in MapReduce. For each key, an aggregate result is produced.

As discussed above, frequent checkpointing is performed to increase fault-tolerance of long-time analysis, and at the same time, many partitioned data blocks move along distributed nodes [10, 11] in the MapReduce framework. Such frequent local and network I/Os sacrifice efficiency of MapReduce [12]. Regarding this, our previous work [9] suggested a method to reduce I/O cost when processing OLAP queries with MapReduce. The basic idea is to transmit only attributes necessary to perform a map or a reduce task to corresponding worker nodes. For example, consider the following SQL query where the schema of records consists of four attributes, (A, B, C, D).

Q1. SELECT A, AVG(C), MAX(D)
FROM R
WHERE B = "…"
GROUP-BY A

In MapReduce, a map task is generally used for filtering or preprocessing input records, while a reduce task is used to compute aggregation over the records selected from the map task. From this, in the example of Q1, the three attributes (A, C, D) will be used in reducers for aggregate computation, while the attribute B will be used in mappers for record selection. Note that A is also necessary for the map task because all records selected from mappers must be grouped according to their aggregate key values before being transmitted to reducers in MapReduce. The key A is called a *shuffling key* in MapReduce. As a result, attributes (A, B) are used for the map task in this example.

In the proposed method, attributes (C, D) are not transferred to mappers because those attributes are not used in the map task. This is different from the original MR approach, where all input data is transferred to mappers. On the other hand, mappers' outputs may not have all information necessary to perform a reduce task in our approach. This is because mappers do not receive a complete data set. Subsequently, 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 selected records, a list of selected record IDs is used for communication between mappers and reducers in this approach.

Figure 1 compares data flows in the original MR approach and the proposed method for Q1. Compared with the original approach, the proposed method requires an additional process to generate and transmit the list of selected record IDs. Regarding this, our previous work provided experimental results showing that its overhead is not significant, compared with the benefit from reduction of data transmission in the map phase.

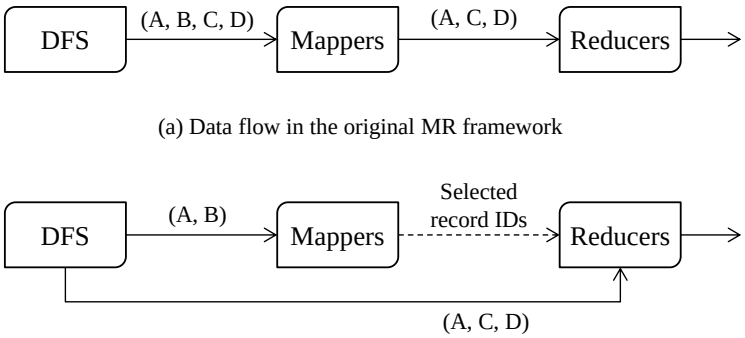

Figure 1. Data Flows in the Original MR Framework and Our Previous Method

Note that our previous work sketched a basic idea for I/O reduction and only considered the case of simple aggregate query processing. No discussion was made about how the method can be applied to join or multi-query processing. In the next section, we discuss a method to extend the idea to multi-query processing with the join.

3. Join and Multi-Query Processing

To support join queries, the *repartition join* [3] is adopted in the proposed method, which is the most popular join strategy in MapReduce. In this join, a join key is used as a shuffling key, so records with a same join key value can be delivered and grouped in a same reducer. Join outputs can then be generated by conducting a cross product over the records transmitted from mappers. Due to its simplicity, many existing data warehouse solutions such as Pig [13] and Hive [14, 15] adopted it to handle multi-way join queries in their systems.

Repartition join is classified into a *reduce-side join* [3, 16], where join takes place in the reduce task. When using it, a dedicated *reduce* () is required to perform the join. If a GROUP-BY clause is defined together in the join query, another *reduce*() is necessary to deal with it. In this case, two MR jobs are required to process the join aggregate query. In general, each stateful operator such as join or group-by aggregate are mapped to a *reduce*(), and the number of MR jobs to process a join aggregate query becomes equal to the number of stateful operators in the query.

To illustrate the process, consider the following query to join two relational tables, R1(A, B, C, D, E) and R2(E, F), over the common attribute E. In the query, aggregates are then computed over the join results.

Q2. SELECT A, AVG(C), MAX(D), MAX(F)
FROM R1, R2
WHERE R1.E = R2.E AND B = "…"
GROUP-BY A

To perform Q2, two MR jobs are necessary: one for join and the other for aggregates. In the first job, a map task is used for record selection, so attribute B of R1 is necessary in this job. Attribute E is also necessary because records selected from mappers must be grouped for each join key value before being transmitted to reducers. Subsequently, (B, E) and (E) should be projected from R1 and R2, respectively, and then transferred to mappers.

After mappers finish their job, IDs of selected records are transferred to reducers. Based on the ID information, reducers fetch the actual records selected from mappers. In this process, all attributes necessary to perform aggregates in the second job are required to be fetched. In the example of Q2, (A, C, D) and (F) are necessary for the aggregations,

so those attributes are transferred to reducers from the DFS. Over the fetched records, a cross-product is performed to generate join records. The results are stored in a temporary relation, T, consisting of four attributes (A, C, D, F).

The second job to compute aggregates can be performed as discussed in Section 2. In this case, there is no record selection in the map task because it is already performed in the first job. Thus, only attribute A is transferred to mappers for later GROUP-BY aggregation. The remaining reduce task is the same as discussed in our previous work.

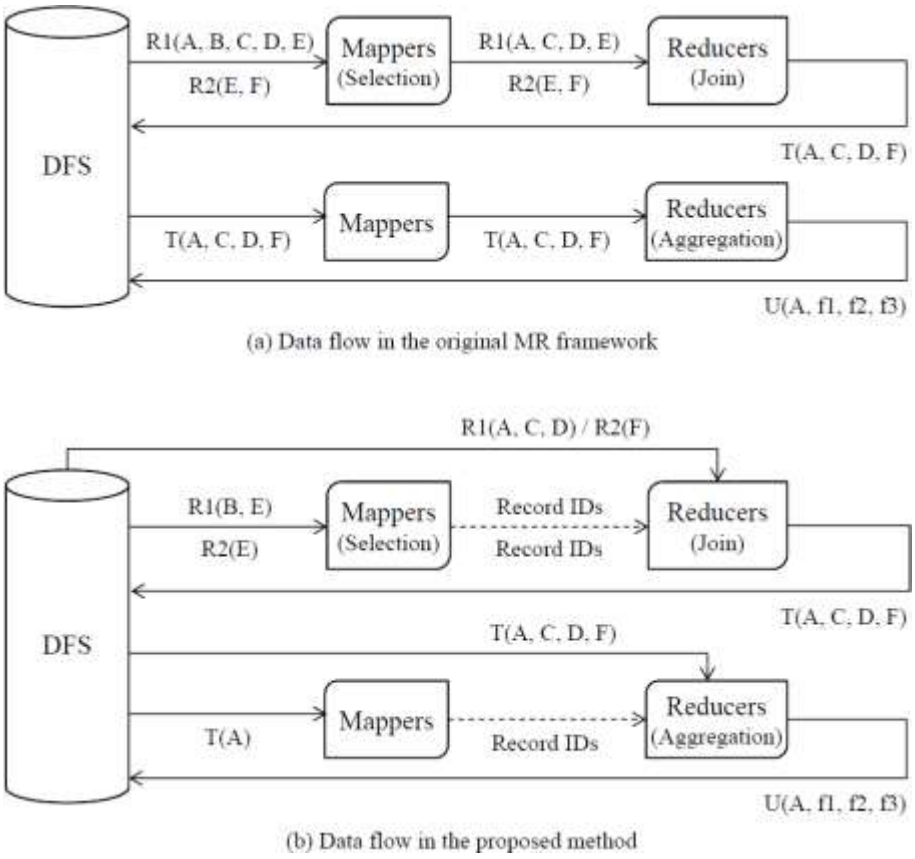

Figure 2. Data Flows in the Original and the Proposed Method for Q2

Figure 2 compares the data flows in the original and the proposed methods for Q2. In the original MR approach (Figure 2 (a)), all attributes necessary to perform the successive MR tasks are relayed to each next step. For instance, after mappers complete their record selection, attribute B is excluded from the record transmission to reducers. Similarly, after finishing the join, E is excluded from both R1 and R2 because it is not necessary for further processing. Join results are stored in the temporary relation, T(A, C, D, F), which are inputted for aggregation in the second MR job. Aggregations are finally performed over the join results and are stored in another relation, U, where f_i denotes a result of the *i*-th aggregation defined in the SELECT clause.

On the other hand, in the proposed method (Figure 2 (b)), only attributes necessary to perform each map or reduce task are transmitted to corresponding worker nodes. As discussed above, only (B, E) are transferred to mappers in the first job which are required to perform the record selection and the next join. After mappers complete their job, the selected record IDs are transmitted to reducers. Using the IDs, reducers fetch the actual selected records from the DFS. Distinguished from the original MR approach, attribute E is excluded from the fetched records. This is because a cross-product to generate join outputs can be performed based on the record IDs, and attribute E is not necessary in this case. The processing of the second job is also performed in the similar fashion.

Now, let us discuss translation of the query Q2. The query is translated into two MR jobs. The first is to join input records from R1 and R2, while the second is to perform aggregates over the records outputted from the first job. Since MapReduce is designed for processing a single input, *map*() of the first job should be written to deal with records received from both R1 and R2. The following pseudo code shows *map*() for the join part of Q2.

Function *map*(*rowID*, *record*)
    If *source* = "R1",
        Parse *record* into attributes (B, E)
        If B = "…", output(E, "R1::" + *rowID*)
    End If
    If *source* = "R2",
        Parse *record* into attributes (E)
        output(E, "R2::" + *rowID*)
    End If

Above, the processing of records is determined based on the *source* information. If a record is from R2, it is simply outputted to a reducer because no selection condition is defined over attributes of the table. In this case, *rowID* is outputted with attribute E as a key for join. Before sending out the value, a tag "R2::" is added to it, where ``::" is a delimiter to separate the table name from the tagged ID. The tag insertion is necessary because reducers also need to distinguish records from a single input channel. On the other hand, if the record is from R1, record selection is performed as specified in the WHERE clause of Q2. If the condition is satisfied, the record is outputted to a reducer. In this case, the attribute E is outputted as a key for join, and *rowID* with tag "R1::" is outputted as a value.

Using the tag name attached on the record IDs, reducers can identify the source table of selected records, and can group the records according to their tag name. After grouping is finished, a cross product is performed to generate join outputs. The following shows *reduce*() generated from Q2.

Function *reduce*(*key*, *taggedRowIDs*)
    $listID_{R1}$:= *null*, $listID_{R2}$:= *null*
    For each *id* in *taggedRowIDs*,
      Separate *id* into (*tag*, *rowID*)
      If *tag* = "R1", add *rowID* to $listID_{R1}$
      If *tag* = "R2", add *rowID* to $listID_{R2}$
    End If
    $listRec_{R1}$:= *fetchRecords*("R1", "A, C, D, E", $listID_{R1}$)
    $listRec_{R2}$:= *fetchRecords*("R2", "F", $listID_{R2}$)
    Output("T", *crossProd*($listRec_{R1}$, $listRec_{R2}$))

Above, $listID_{R1}$ and $listID_{R2}$ are the lists to store the record IDs of R1 and R2, respectively. The input parameter *taggedRowIDs* are the list of tagged record IDs. For each tagged ID, *reduce*() separates a record ID from its tag denoting the source table. Based on the tag name, the extracted record ID is assigned to one of $listID_{R1}$ and $listID_{R2}$. After all record IDs are grouped according to their tag information, actual records corresponding the record IDs are fetched from the DFS. In this process, the following three kinds of attributes are extracted, which are necessary to perform the join and the group-by aggregation:

- Join key attribute of each table
- GROUP-BY attribute necessary to group join outputs for aggregation in the next MR job
- Attributes appearing in aggregate function in the SELECT clause

In this example, (A, C, D, E) are extracted from R1, while (F) is extracted from R2. After the projected records are fetched from the DFS, a cross product is performed over the

records, and the results are stored in the DFS. The schema of the temporary relation to store the join records is (A, C, D, E, F) in this case.

Now, let us discuss the second MR program generated to compute the group-by aggregation from the join results. The *map*() and *reduce*() of the second MR program can be described as follows. Its mechanism was discussed in our previous paper [9], so we omit its discussion.

Function *map*(*rowID*, *record*)
    Parse *record* into attributes (A)
    output(A, *rowID*)
Function *reduce*(*key*, *rowIDs*)
    listRec := *fetchRecords*("T", "*", *rowIDs*)
    Output(*key*, *count*(*listRec*))

The above approach can easily be extended to multi-query processing. For example, a multi-query can be processed by connecting a series of MR jobs, each of which is generated from a unit query constituting the multi-query.

4. SQL-to-MapReduce Translation

To translate SQL queries into MR programs, the one-operation-to-one-job approach [17] is adopted in our implementation. In this approach, a plan tree generated from a given query consists of several relational operators including selection, projection, join, and aggregation operators. Each operator in the tree is then replaced by a pre-implemented MR program. To simplify discussion, we classify an SQL query into four types according to the existence of join or group-by/order-by aggregate operators in the query, which is shown in Table 1.

Table 1. SQL-To-MapReduce Translation Rules for Each Query Type

Join \ Group-by / order-by	No	Yes
No	Type.1 *map*() : [selection] *reduce*() : [simple aggregation]	Type.2 *map*() : [selection] *reduce*() : group-by/order-by aggregation
Yes	Type.3 *map*() : [selection] *reduce*() : [simple aggregation], join	Type.4 Job1 *map*() : [selection] *reduce*() : join Job 2 *map*() : – *reduce*() : group-by/order-by aggr.

Type-1 queries are the simplest form where there is no join or group-by aggregate in the queries. A type-1 query is translated into a single MR job. In this case, *map*() is written to perform the record selection specified in the query, and *reduce*() is to perform simple aggregations (defined without a GROUP-BY clause) over all the records selected from mappers.

Type-2 queries are queries where only group-by or order-by aggregations are involved. If more than one group-by or order-by aggregations are included in the query, each of them is translated into an individual MR job. For this type of query, *map*() is written to perform the record selection, as in the type-1 queries, and output the selected records according to their group-by or order-by attribute values. *Reduce*() is then written to perform a given group-by or order-by aggregation.

Type-3 queries are join queries without any group-by or order-by aggregation. This type of query can also be translated into a single MR job. For this query, *map*() is written to perform record selection and output the records based on their join key values, while *reduce*() is to perform a cross-product over the records to produce join results. In the reduce task, simple aggregation can be performed over the join outputs.

Type-4 queries include joins and group-by aggregations together. A type-4 query is translated into two MR jobs. The first job is similar to that of type-3 queries, where *map*() performs record selection and reduce() executes join. The second job is similar to that of type-2 queries, where *reduce*() computes aggregations over the records generated from the previous stage. In the second job, record selection does not occur in the map phase because it is already done in the first job.

In the above discussion, record selection is optionally performed in *map*(); it is performed only when the corresponding condition is specified in the query. To denote the optional execution, the square brackets are used in Table 1. Below, we do not discuss the ORDER-BY clause. To deal with it, another *reduce*() is necessary to sort records in type-2 and type-4 queries. But its addition to the MR jobs discussed above is straight-forward. In what follows, we present translation rules for type-1 and type-2 queries, and then describe rules for type-4 queries. Rules for type-3 queries can be derived from those of type-4 queries, from which their discussion is omitted.

4.1. Type-1 and Type-2 Queries

Let <*table*> denote a relational table where records are drawn from, <*aggr-keys*> be a list of group-by attributes, <*aggr-funcs*> be a list of aggregate functions over attributes of the <*table*>, and <*sel-conds*> denote a list of record selection conditions. Then, the query we consider can be represented as follows.

Q1. SELECT <*aggr-keys*> <*aggr-funcs*>
FROM <*table*>
WHERE <*sel-conds*>
GROUP-BY <*aggr-keys*>

Given *Q1*, *map*() can be generated as follows. Below, *getStr*() is a function to generate a composite string for attribute values in <*aggr-keys*>, such that *getStr*(*K*) = "k_1, k_2, ..., k_n" where K = <*aggr-keys*> = { k_i } ($1 \le i \le n$, $n = |K|$). This function is necessary when <*aggr-keys*> has more than one attributes. Note that in our previously proposed method (Figure 1), only IDs of selected records are outputted with their key values to the reduce phase.

Function *map*(*rowID*, *record*)
Parse *record* into attributes in <*aggr-keys* > and <*sel-conds*>
If <*sel-conds*> is satisfied,
Output(*getStr*(<*aggr-keys* >), *rowID*)
End If

The following shows *reduce*() generated from *Q1*. In our method, records selected from the map phase must be fetched from the DFS using the given record ID list (*i.e.*, *rowIDs*). The function *fetchRecords*() is used for this purpose. Below, *A* denotes the set of attributes defined in <*aggr-funcs*>, such that <*aggr-funcs*> = { $f_i(\alpha_i)$ }($1 \le i \le n$, $n = |A|$) where f_i is an aggregate function over attribute α_i.

Function *reduce*(*key*, *rowIDs*)
recs := *fetchRecords*(<*table*>, *getStr*(*A*), *rowIDs*)
For each aggregation f_i (α_i) in <*aggr-funcs*>,
val := calculate f_i (α_i) over *recs*
Output(*key*, *val*)
End If

Note that in our method, only aggregate attributes defined in the SELECT clause of a given query are included in the fetched records. In the example of Figure 1, only *C* and *D* are fetched from the attributes for aggregation. For this projection, the function *getStr*(*A*) is used to denote which attributes must be projected from the fetched records. After the projected records are obtained, actual computation is performed for each aggregation $f_i(\alpha_i)$. The computed value is outputted with a given key as a final output.

4.2. Type-4 Queries

Let <*tables*> denote a list of relational tables and <*join-conds*> denote a list of join conditions. Other parameters such as <*aggr-keys*>, <*aggr-funcs*>, and <*sel-conds*> are the same as in the previous subsection. Then, the join aggregate query we consider can be represented as follows.

Q2. SELECT <*aggr-keys*> <*aggr-funcs*>
FROM <*tables*>
WHERE <*sel-conds*> <*join-conds*>
GROUP-BY <*aggr-keys*>

Let us discuss the first MR program to deal with the join part of *Q2*. Since *map*() and *reduce*() are designed to receive records from a single input channel, they need to be written to deal with records inputted from multiple source tables. To enable this, the following information must be available for each table T_i before the generation of *map*() and *reduce*().

- $tname_i$: name of the *i*-th table T_i in <*tables*>
- $join\text{-}key_i$: join attribute of T_i appearing in <*join-conds*>
- $sel\text{-}attrs_i$: list of attributes in T_i appearing in <*sel-conds*>
- $sel\text{-}conds_i$: list of selection conditions defined over $<sel\text{-}attrs_i>$

The above parameters can be identified during the parsing of an SQL query, where their identification process is omitted due to the lack of space. Based on the parameters, *map*() can be generated as follows. Below, *source* denotes the name of a source table of a given record.

Function *map*(*rowID*, *record*)
If *source* = $<tname_1>$,
Parse *record* into attributes { $<join\text{-}key_1> \cup <sel\text{-}attrs_1>$ }
If $<sel\text{-}conds_1>$ is satisfied,
Output($<join\text{-}key_1>$, $<tname_1>$ + "::" + *rowID*)
End If
End If
If *source* = $<tname_2>$,
…

Above, each record can be processed differently according to its source table name. Suppose that the record selection condition is defined over attributes of table T_1 and the current input record is drawn from T_1. Then, we parse the record and check whether it satisfies the given selection condition, *i.e.*, $<sel\text{-}conds_1>$. If so, it is outputted to the reduce phase. In this case, the output key becomes a join key attribute because the next step performs the repartition join as discussed above. Before sending out the record ID, the table name $<tname_1>$ is added with a delimiter string "::" in front of the ID. The record tagging is necessary because reducers also receive input records from a single input channel, so they also need to identify the table name of each input record.

Using the tag name attached on each record ID, reducers group record IDs having the same table name. After the grouping is finished, a cross-product is performed to generate join outputs. The following shows *reduce*() generated from *Q2*. To simplify discussion, we discuss the case of binary join. The method presented below can easily be extended to the case of multi-way join.

Function *reduce*(*key*, *taggedRowIDs*)
    $listID_1$:= *null*, $listID_2$:= *null*
    For each *id* in *taggedRowIDs*,
      Separate *id* into (*tag*, *rowID*)
      If *tag* = <$tname_1$>, add *rowID* to $listID_1$
      If *tag* = <$tname_2$>, add *rowID* to $listID_2$
    End If
    $listRec_1$:= *fetchRecords*(<$tname_1$>, *getStr*(A_1), $listID_1$)
    $listRec_2$:= *fetchRecords*(<$tname_2$>, *getStr*(A_2), $listID_2$)
    Output("*_joinRes*", *crossProd*($listRec_1$, $listRec_2$))

Above, the parameter *taggedRowIDs* denote the list of tagged record IDs. For each ID in the list, *reduce*() first separates the record ID and its table name. According to the table name, record IDs are grouped into the temporary list, called $listID_i$. After the ID lists are obtained, *reduce*() fetches the selected records from the DFS using the lists. For this purpose, *fetchRecords*() is also used where A_i is a union of attributes used for join and aggregates in table T_i. Over the fetched records, a cross-product is performed to produce join results.

The *map*() and *reduce*() of the second MR program are similar to those discussed in the previous. The only difference lies in that *map*() needs not consider the record selection in this case.

5. Conclusion

In this paper, we presented SQL-to-MR translation techniques for efficient OLAP query processing with MapReduce. The MapReduce programs generated from our translator are organized to support the method proposed to reduce I/O cost where the idea was discussed in our previous work. To implement translation, the one-operation-to-one-job approach was adopted and repartition join was used to deal with join queries. From this, the number of the generated MR jobs becomes equal to the number of join and group-by/order-by aggregations defined in a given query. The translation rules presented in this paper are designed to support the method suggested in our previous paper for I/O cost reduction, but they also can be applied to build a common SQL-to-MR compiler.

Acknowledgments

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B1009792).

This paper was also supported by the Sahmyook Research Fund in 2015.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", Commun. ACM., vol. 51, no. 1, **(2008)**, pp. 107-113.
[2] S. Madden, "From database to big data", IEEE Internet Computing, vol. 16, no. 3, **(2012)**, pp. 4-6.
[3] S. Blanas, "A Comparison of Join Algorithms for Log Processing in MapReduce", Proc. of the ACM SIGMOD, **(2010)**, pp. 975-986.
[4] C. Doulkeridis and K. Norvag, "A survey of large-scale analytical query processing in MapReduce", VLDB J., vol. 23, no. 3, **(2014)**, pp. 355-380.
[5] K. H. Lee, "Parallel data processing with MapReduce: a survey", ACM SIGMOD Record, vol. 40, no. 4, **(2012)**, pp. 11-20.
[6] A. Pavlo, "A comparison of approaches to large-scale data analysis", In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD'09), **(2009)**, pp. 165-178.
[7] E. Anderson and J. Tucek, "Efficiency matters!", ACM SIGOPS Oper. Syst. Rev., vol. 44, no. 1, **(2010)**, pp. 40-45.
[8] W. Kang, H. Kim, and Y. Lee, "I/O cost evaluation of OLAP query processing with MapReduce", Lect. Notes Electr. Eng., vol. 330, no. 2, **(2015)**, pp. 997-1002.
[9] W. Kang, H. Kim, and Y. Lee, "Reducing I/O cost in OLAP query processing with MapReduce", IEICE T. Inf. Syst., vol. E-98D, no. 2, **(2015)**, pp. 444-447.
[10] Kim, J. M., "Compromise scheme for assigning tasks on a homogeneous distributed system", J. lnf. Commun. Converg. Eng., vol. 9, no. 2, **(2011)**, pp. 141-149.
[11] J. T. Kim, "Analyses of characteristics of u-healthcare system based on wireless communication", J. lnf. Commun. Converg. Eng., vol. 10, no. 4, **(2012)**, pp. 337-342.
[12] J. Dawei, O. B. Chin, S. Lei, and W. Sai, "The performance of MapReduce: An in-depth study", Proceedings of the VLDB Endowment, vol. 3, no. 1-2, **(2010)**, pp. 472-483.
[13] C. Olston, "Pig Latin: A not-so-foreign language for data processing", Proc. of the ACM SIGMOD, **(2008)**, pp. 1099-1110.
[14] A. Thusoo, "Hive: a warehousing solution over a map-reduce framework", Proc. of the VLDB Endowment, vol. 2, no. 2, **(2009)**, pp. 1626-1629.
[15] A. Thusoo, "Hive - a petabyte scale data warehouse using Hadoop", In: Proceedings of the IEEE 26th International Conference on Date Engineering (ICDE'10), **(2010)**, pp. 996-1005.
[16] T. Lee, H. Bae, and H. Kim, "Join processing with threshold-based filtering in MapReduce", J. Supercomput., vol. 69, **(2014)**, pp. 793-813.
[17] R. Lee, "YSmart: yet another SQL-to-MapReduce translator", Proc. of the ICDCS, **(2011)**, pp. 25-36.

Author

Hyeon Gyu Kim, received B. Sc. (1997) and M. Sc. (2000) degrees in Computer Science from the University of Ulsan and received his Ph.D. degree (2010) in Computer Science from the Korea Advanced Institute of Science and Technology (KAIST). He was a Chief Research Engineer at LG Electronics from 2001 to 2011, and a Senior Researcher at the Korea Atomic Energy Research Institute (KAERI) from 2011 to 2012. He joined the faculty of the Department of Computer Engineering at Sahmyook University, Seoul, Korea in 2012, where he is currently an assistant professor. His research interests include databases, data stream processing, mobile computing, and probabilistic safety assessment.

