Multi-Way Join using Map Reduce for Big Data Applications

Suren kumar Sahu
Department of Computer Science and Engineering
Gandhi Engineering College,Bhubaneswar,Odisa,India,

Sarbajit Chhualsingh
Gandhi Engineering College,Bhubaneswar,Odisha,India

S.Gayatri Dora
Department of Computer Science
Regional Institute of Education,Bhubaneswar,Odisha,India

Abstract - Multi-way join is an important and frequently used operation for many big data applications including data mining and knowledge discovery, Since join processing is expensive, mainly for large data sets, multi-way join is a costly operation. When processing multi-way joins of big data, a natural join to ensure the reasonable response time is parallel processing. As a parallel programming model, Map Reduce becomes the popular big data programming model for its simplicity, flexibility, fault tolerance and scalability. In this paper we have tested multi-way join using phases map side join and reduced side join.

Keywords: Bigdata, Map Reduce, Hadoop, Semi joins, partitioner,HDFS

I. INTRODUCTION

Join processing in Map Reduce [1] has attracted the attention of researchers in recent years. This is because MapReduce does not support join operations directly, although it is a useful framework for large-scale data analysis. In particular, joining multiple datasets in MapReduce has been a challenging problem because it may amplify the disk and network overheads. Multiple datasets can be joined in the following two ways:

1. using a cascade of two-way (or smaller multi-way) joins and
2. with a single multi way join.

However, both methods have some drawbacks. A cascade of two-way joins has to write the intermediate join results to the underlying distributed file system, which generally replicates multiple records to ensure high availability and fault tolerance. To process multi way joins in a single Map Reduce job, the map output records have to be replicated multiple times, instead of writing only the final join results to the distributed file system.

II. MAPREDUCE

2.1 *MapReduce* [1] is Google's programming model for large-scale data processing, which is run on a shared-nothing cluster. MapReduce liberates users from the responsibility of implementing parallel and distributed processing features by providing them automatically. Thus, users only have to write MapReduce programs with two functions: map and reduce. The map function takes a simple key/value pair as its input and it produces a set of intermediate key/value pairs. The reduce function takes an intermediate key and a set of values that correspond to the key as its input, and it generates the final output key/value pairs.

A MapReduce cluster comprises one master node and a number of worker nodes. When a MapReduce job is submitted, the master node creates the map, reduces tasks, and assigns each task to idle workers. A map worker reads the input split and executes the map function submitted by the user. The map output records are grouped and sorted by the key and then stored in partitions for each reduce worker. A reduce worker reads its corresponding partitions from all the map workers, merges the partitions, and executes the reduce function. When all of the tasks are complete, the MapReduce job is finished.

Hadoop [9] is a popular open-source implementation of the Map Reduce framework. In Hadoop, the master node is called the job tracker and the worker node is called the task tracker. Task trackers run one or more

mapper and reducer processes, which execute map and reduce tasks, respectively. The proposed method was implemented using Hadoop, so the Hadoop terminology is used in the remainder of this paper.

2.2 Join Processing in Map Reduce

Join algorithms in Map Reduce are classified roughly into two categories: map-side joins and reduce-side joins [10]. Map-side joins produce the final join results in the map phase and do not use the reduce phase. They do not need to pass intermediate results from mappers to reducers, which means that map-side joins are more efficient than reduce-side joins, although they can only be used in particular circumstances. Hadoop's map-side join [11], referred to as the map-merge join [10], merges input datasets that are partitioned and sorted on the join keys in the same manner, which is similar to the merge join in traditional DBMS. An additional MapReduce job is required if the input datasets are not partitioned and sorted in advance. The broadcast join [12] distributes the smaller of the input datasets to all of the mappers and performs the join in the map phase. This approach is efficient only if the input dataset is small.

Reduce-side joins can be used in more general cases, but they are inefficient because large intermediate records are sent from mappers to reducers. The repartition join [12] is the most common join algorithm in MapReduce, but all of the input records have to be sent to reducers, including redundant records that are not relevant to the join. This may lead to a performance bottleneck. The semi join in MapReduce [12] works in a similar manner to semi join in traditional DBMS. This approach may reduce the size of the intermediate results by filtering out the unreferenced records with unique join keys. Therefore, it is efficient when small portions of records participate in joins. However, the semijoin requires three MapReduce jobs, which means that the results of each job are written and read in the next job. This incurs additional I/O overheads.

III. JOIN ALGORITHMS

Two-way Joins - Given two dataset P and Q, a two-way join is defined as a combination of tuples pЄP and qЄQ, such that p.a = q.b. a and b are values from columns in P and Q respectively on which the join is to be done. Please note that this is specifically an 'equi-join' in database terminology. This can be represented as-

$$P \bowtie_{a=b} Q$$

Multi-way Joins - Given n datasets P1, P2... Pn, a multi-way join is defined as a combination of tuples p1 ЄP1; p2 ЄP2,... pnЄ Pn, such that p1.a1 = p2.a2 =... = pn.an. a1, a2,... an are values from columns in P1, P2,... Pn respectively on which the join is to be done. Notice once again that this is specifically an 'equi-join'. This can be represented as-

$$P_1 \bowtie_{a_1=a_2} P_2 \bowtie_{a_2=a_3} \cdots \bowtie_{a_{n-1}=a_n} P_n$$

The algorithms thus described in this chapter have been divided into two categories-

1. Two-Way Joins - Joins involving only two tables
2. Multi-Way Joins - Joins involving more than two tables

3.1 Reduce-Side Join

In this algorithm, as the name suggests, the actual join happens on the Reduce side of the framework. The 'map' phase only pre-processes the tuples of the two datasets to organize them in terms of the join key.

3.1.1 Map Phase

The map function reads one tuple at a time from both the datasets via a stream from HDFS. The values from the column on which the join is being done are fetched as keys to the map function and the rest of the tuple is fetched as the value associated with that key. It identifies the tuples' parent dataset and tags them. A custom class called TextPair has been defined that can hold two Text values. The map function uses this class to tag both the key and the value. The reason for tagging both of them will become clear in a while. Here is the code for the map function.

```
void map(Text key , Text values ,
OutputCollector <TextPair, TextPair> output , Reporter reporter) throws IOException {
```

```
output.collect(new TextPair(key.toString (), tag),
new TextPair(values.toString (), tag));
}
```

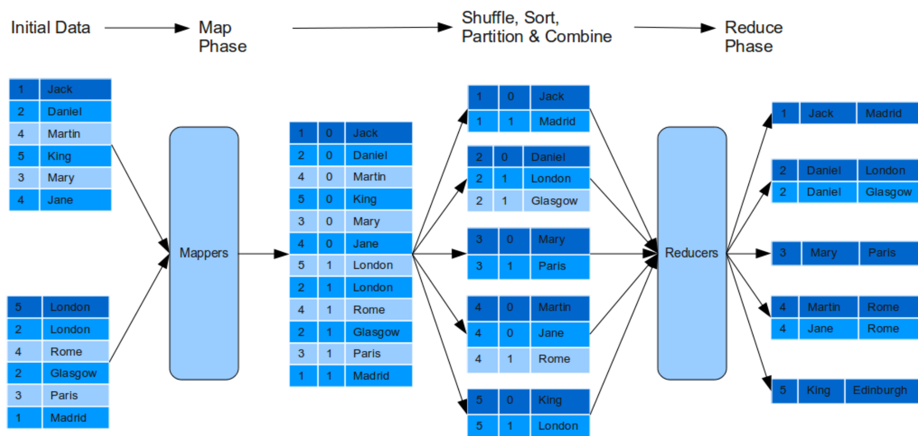

Fig-1 Reduce-Side Join for Two way join

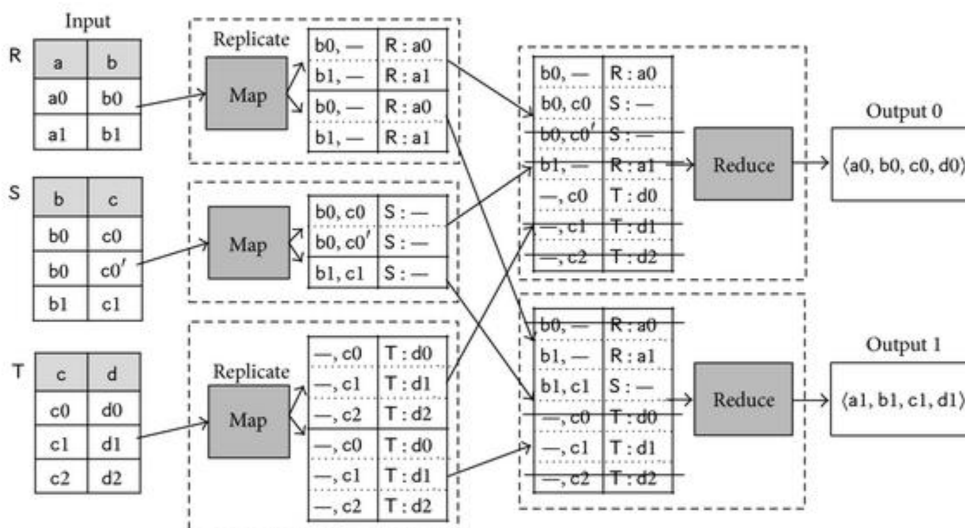

Fig-2 Reduce-side join for Multi-Way join

3.1.2 Partitioning and Grouping Phase

The partitioner partitions the tuples among the reducers based on the join key such that all tuples from both datasets having the same key go to the same reducer .. The default partitioner had to be specifically overridden to make sure that the partitioning was done only on the Key value, ignoring the Tag value. The Tag values are only to allow the reducer to identify a tuple's parent dataset.

```
int getPartition (TextPair key , TextPair value , int numPartitions ) {
return (key.getFirst ().hashCode () & Integer.MAX_VALUE)% numPartitions ;
}
```

But this is not sufficient. Even though this will ensure that all tuples with the same key go to the same Reducer, there still exists a problem. The **reduce** function is called once for a key and the list of values associated with it. This list of values is generated by grouping together all the tuples associated with the same key (see section 2.4.6). We are sending a composite TextPair key and hence the Reducer will consider (key, tag) as a key. This means, for eg., two different **reduce** functions will be called for [Key1, Tag1] and [Key1, Tag2]. To overcome

this, we will need to override the default grouping function as well. This function is essentially the same as the partitioner and makes sure the reduce groups are formed taking into consideration only the Key part and ignoring the Tag part.

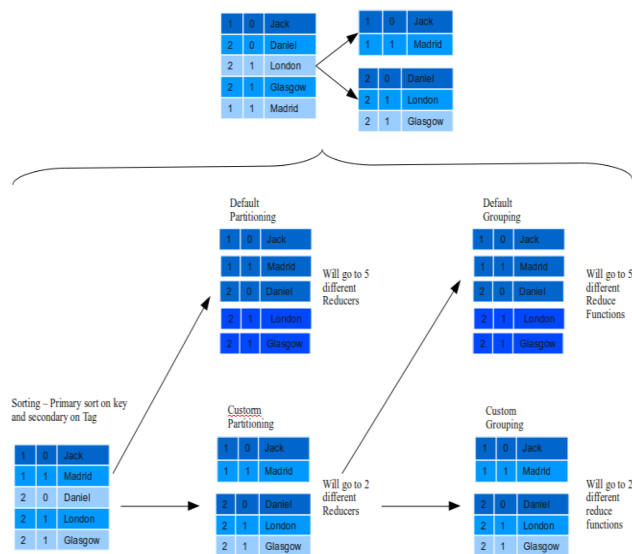

Fig- 3 Custom Partitioning and Grouping

3.1.3 Reduce Phase

The reduce phase is slightly more involved than the two sided join. The Reducer as usual, gets the tuples sorted on the (key, tag) composite key. All tuples having the same value for the join key, will be received by the same Reducer and only one **reduce** function is called the job configuration, the Reducer dynamically creates buffers to hold all but the last datasets. The last dataset is simply streamed from the file system. As a necessary evil, required to avoid running out of memory, the buffers are spilled to disk if they exceed a certain pre-defined threshold. This quite obviously will contribute to I/O and runtime.
Once the tuples for a particular key are divided as per their parent datasets, it is now a Cartesian product of these tuples. Subsequently, the joined tuples are written to the output.

Advantages and Drawbacks
Advantages

- Joining in one go means no setting up of multiple jobs No intermediate results involved which could lead to substantial space savings.

Disadvantages

- Buffering tuples can easily run in to memory problems, especially if the data is skewed
- If used for more than 5 datasets, its highly likely that the buffer will overflow.

Reduce-Side Cascade Join

This is a slightly different implementation of the Reduce-Side Join for Multi-Way joins. Instead joining all the datasets in one go, they are joined two at a time. In other words, it is an iterative implementation of the two-way Reduce-Side Join The implementation is the same as the Reduce-Side Join mentioned in section 3.2.1. The calling program is responsible for creating multiple jobs for joining the datasets, two at a time. Considering n tables, T1;T2;T3; ::;Tn, T1 is joined with T2 as part of one job. The result of this join is joined with T3 and so on. Expressing the same in relational algebra –

$$(\ldots(((T_1 \bowtie T_2) \bowtie T_3) \bowtie T_4) \ldots \bowtie T_{n-1}) \bowtie T_n$$

Advantages and Drawbacks
Advantages

- Data involved in one Map/Reduce job is lesser than the algorithm mentioned in section 3.3.2. Hence lesser load on the buffers and better I/O.
- Datasets of any size can be joined provided there is space available on the HDFS.

- Any number of datasets can be joined given enough space on the HDFS.

Disadvantages

- Intermediate results can take up a lot of space. But this can be optimized to some extent as explained in the next section.
- Setting up the multiple jobs on the cluster incurs a non-trivial overhead.

3.2 Reduce-Side Cascade Join - Optimization

Simply joining datasets two at a time is very inefficient. The intermediate results are usually quite large and will take up a lot of space. Even if these are removed once the whole join is complete, they could put a lot of strain on the system while the cascading joins are taking place. Perhaps there is a way of reducing the size of these intermediate results. There are two ways this can be achieved -

- Compressing the intermediate results
- Join the datasets in increasing order of the output cardinality of their joins, i.e., join the datasets that produce the least number of joined tuples first, then join this result with the next dataset which will produce the next lowest output cardinality.

Optimization using compression

Compressing the results will not only save space on the HDFS, but in case the subsequent join's Map task needs to fetch a non-local block for processing, it will also result in lower number of bytes transferred over the network.

Optimization using Output Cardinality

Deciding the order of joining datasets using the output cardinality is more involved.

Lets consider two sample datasets –

Key	Value
1	ABC
2	DEF
2	GHI
3	JKL

Table-1

Key	Value
1	LMN
2	PQR
3	STU
3	XYZ

Table-2

The output of joining these two datasets will be -

Key	Table -1	Table -2
1	ABC	LMN
2	DEF	PQR
2	GHI	PQR
3	JKL	STU
3	JKL	XYZ

Table-3

The thing to notice is that the output will have

$$T_1(K_1) * T_2(K_1) + T_1(K_2) * T_2(K_2) \ldots T_1(K_n) * T_2(K_n)$$

$$= \sum_{i=1}^{n} T_1(K_i) * T_2(K_i)$$

number of rows. where Tm(Kn) represents the number of tuples in table Tm having the key Kn. In our case, it will be

$$1 * 1 + 2 * 1 + 1 * 2 = 5$$

Hence, if we find out the number of keys in each dataset and the corresponding number of tuples associated with each key in those datasets, we can find the number of output tuples in the join, or in other words, the output

cardinality of the joined dataset. This can be very easily done as part of a pre-processing step.. To accomplish this, output cardinalities are found pair-wise for all datasets at pre-processing time and stored in files. When the join is to be executed, the datasets with the lowest output cardinality are chosen and joined first. This intermediate result this then joined with the dataset that has the lowest join output cardinality with either of the two tables joined previously. In the next round, the dataset with the lowest join output cardinality with any of the three datasets joined so far is chosen to join with the intermediate result and so on. The next chapter shows the results of experimental evaluation and compares and contrasts the gains that
such optimization leads to.

3.3 Map-Side Join

The Reduce-Side join seems like the natural way to join datasets using Map/Reduce. It uses the framework's built-in capability to sort the intermediate key-value pairs before they reach the Reducer.
But this sorting often is a very time consuming step. Hadoop offers another way of joining datasets before they reach the Mapper. This functionality is present out of the box and is arguably is the fastest way to join two datasets using Map/Reduce, but places severe constraints (see table 3) on the datasets that can be used for the join.

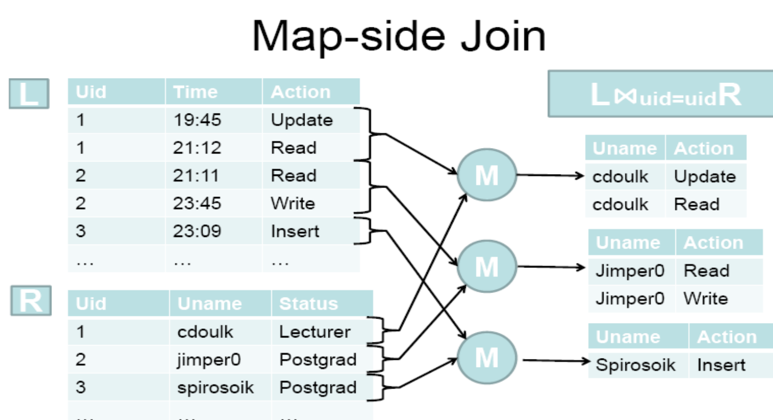

Fig. 4-Mapside Join

Table 3: Limitation of Map-Side Join

Limitation	Why
All datasets must be sorted using the same comparator.	The sort ordering of the data in each dataset must be identi-cal for datasets to be joined.
All datasets must be parti-tioned using the same parti-tioner.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.
The number of partitions in the datasets must be identical.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.

These constraints are quite strict but are all satisfied by any output dataset of a Hadoop job. Hence as a pre-processing step, we simply pass both the dataset through a basic Hadoop job. This job uses an Identity Mapper and an Identity Reducer which do no processing on the data but simply pass it through the framework which partitions, groups and sorts it. The output is compliant with all the constraints mentioned above. Although the individual map tasks in a join lose much of the advantage of data locality, the overall job gains due to the potential for the elimination of the reduce phase and/or the great reduction in the amount of data required for the reduce.
This algorithm also supports duplicate keys in all datasets

3.4 Reduce-Side One-Shot Join

This is basically an extension of the Reduce-Side join .
A list of tables is passed as part of the job configuration to enable the Mapper and Reducer to know how many tables to expect and what tags are associated with which table. For instance, to join n datasets, T1,T2,T3,…Tn, this algorithm in simple relational algebra can be represented as –

$$T_1 \bowtie T_2 \bowtie \ldots \bowtie T_{n-1} \bowtie T_n$$

IV. EVALUATION AND ANALYSIS

4.1 Environment

Experiments were conducted on 3 different kinds of clusters .There were two types of machines used that were slightly different from each other –

	Type 1	Type 2
CPU	Intel ^{R} Xeon^{TM}CPU 3.20GHz	Intel ^{R} Xeon^{TM}CPU 3.20GHz
Cores	4	4
Memory	3631632 KB	3369544 KB
Cache size	2 MB	1 MB
OS	Linux	Linux
Linux Kernel	Ubuntu 14.04	Ubuntu 14.04

Table - 4 Types of machines used in the Experimental Cluster

All machines were running Clouders Inc.'s distribution of Hadoop – version 0.18.3+76. File system used in all the experiments was HDFS.

4.2 Experimental Setup

Cluster Setup

There were three types of clusters used –

	Data Node(s)	Name Nodes	Job-Trackers	Total Node(s)
Type 1	1	1	1	1
Type 2	3	1	1	5
Type 3	6	1	1	8

Table -5 Types of clusters used for the experiments

4.3 Multi-way Join algorithms across different clusters

For the multi-way join algorithms , we ran the three algorithms on different types of clusters . The datasets that we used had -

- 1 million tuples each.
- The same number of keys - #keys= 1/10 #tuples spread across the same key space.

Figure 5 shows the results that were observed. In result 1-machine cluster performed better than the 5-machine cluster with the performance then improving again for the 8-machine cluster. On checking the logs, we found the very same reasons for this anomaly. There was no network traffic involved in the case of 1-machine cluster and this was a dominant factor in case of the 5-machine cluster. The 8-machine cluster performed better since the smaller units of work that were well distributed across the cluster took lesser time to complete and brought down the overall time.

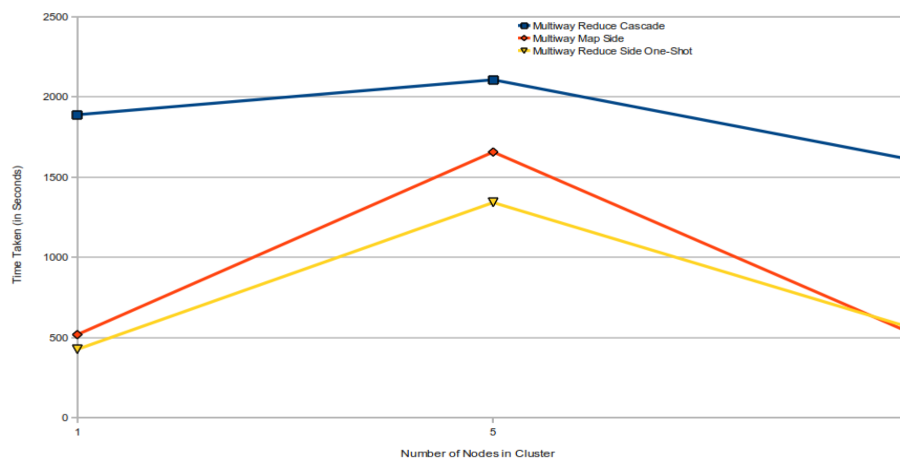

Fig-5 Multi-way Join algorithms across different clusters

No of nodes	Reduce-side one-shot	Data Shuffle Ratio	Map-side Join	Pre-Processing	Reduce-side Cascade join	Pre-processing
1	427	0	519	83	1889	85
5	1342	1.147	1657	243	2107	96
8	565	1.147	534	71	1612	32

Table -6 Multi-way Join algorithms across different clusters

V. CONCLUSION

We found some quite interesting results from our experiments on Join algorithms using Map/Reduce. Even though our clusters were of relatively small sizes, we could notice from our experiments how the Hadoop framework exploited the cluster architecture when more nodes were added . This gave us a first-hand feel of why Hadoop seems to be enjoying such a high popularity among data warehousing specialists.

The behavior of the 1-machine cluster was observed in the case of multi-way joins as well while being tested on different clusters. When we tested the multi-way algorithms with increasing data sizes, we successfully showed that our optimization for Reduce-Side Cascade Join using output cardinality improved the performance of the algorithm. Although this came at the cost of pre-processing step and was the slowest of the algorithms, it was successful in joining datasets that contained over 2.5 million tuples whereas Reduce-Side One-Shot Join gave Out Of Memory exception a number of times.

REFERENCES

[1] Apache cassandra.
[2] Apache hadoop. Website. http://hadoop.apache.org.
[3] Cloudera inc. http://www.cloudera.com.
[4] Hadoop wiki - poweredby. http://wiki.apache.org/hadoop/PoweredBy.
[5] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin.Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. Proc. VLDB Endow., 2(1):922–933, 2009.
[6] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In SIGMOD ’10: Proceedings of the 2010 international conference on Management of data, pages 975–986, New York, NY, USA, 2010. ACM.
[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. Proc. VLDB Endow., 1(2):1265–1276, 2008.
[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In OSDI ’06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.
[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters.Commun. ACM, 51(1):107–113, 2008.
[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. SIGOPS Oper. Syst. Rev., 37(5):29–43, 2003.
[11] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a notso-foreign language for data processing. In SIGMOD ’08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 1099–1110, New York, NY, USA, 2008. ACM.
[12] K. Palla. A comparative analysis of join algorithms using the hadoop map/reduce framework. Master’s thesis, School of Informatics, University of Edinburgh,2009.
[13] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In SIGMOD ’09: Proceedings of the 35th SIGMOD international conference on Management of data, pages 165–1078, New York, NY, USA, 2009. ACM.
[14] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? Commun. ACM,53(1):64–71, 2010.
[15] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P.Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow., 2(2):1626–1629, 2009.
[16] J. Venner. Pro Hadoop. Apress, 1 edition, June 2009.
[17] S. Viglas. Advanced databases. Taught Lecture, 2010. http://www.inf.ed.ac.uk/teaching/courses/adbs.
[18] T. White. Hadoop: The Definitive Guide. O’Reilly Media, 1 edition, June 2009.
[19] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In SIGMOD ’07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data,pages 1029–1040, New York, NY, USA, 2007. ACM.