Improvement of Join Algorithms for Low-Selectivity Joins on MapReduce

Akiyoshi Matono(✉), Hirotaka Ogawa, and Isao Kojima

National Institute of Advanced Industrial Science and Technology,
1-1-1 Umezono Tsukuba, Ibaraki, Japan
{a.matono,h-ogawa,isao.kojima}@aist.go.jp

Abstract. So far, many studies on join operations on MapReduce have already been proposed. Some of those studies focus on the low-selectivity joins that are frequently used in query processing for datasets among several management domains, such as those used with Linked Open Data. We found there is room for improvement of the state-of-the-art approach for the low-selectivity join on MapReduce called the per-split semi-join[5]. In this paper, we first thus extend the per-split semi-join to improve performance. Our approach can reduce three costs for low-selectivity joins: the amount of network traffic, the number of jobs, and the amount of disk I/O. Moreover, when the number of input relations is large, the selectivity becomes low and thus the effect of our proposed reductions is maximized. Therefore, we also propose an extension of the reduce-side join, which can easily apply three or more inputs, based on the extension of the per-split semi-join. In our experiments, we evaluated two comparisons: the per-split semi-join and our extension of it and the reduce-side join and our extension of it. The first experiment shows that our extension is better than its competitor in any case. In the second experiment, we found that our extension is superior to its competitor when the selectivity is low and the number of inputs is three or more.

1 Introduction

Today, the amount of data is rapidly increasing. Parallel processing frameworks, such as MapReduce[8], increase the importance of such frameworks to analyze such large volumes of data. MapReduce is a parallel programming model on a cluster, where an operation is composed of a set of job pairs of a *map* task and a *reduce* task. In MapReduce, a file is divided into a list of segments, called *split*s, and each task is assigned a split and is run in parallel. In a map task, a split is input, a map method that has been written by a user is performed, and then the output of the map method is partitioned into a set of partitions based on a *partitioner*. In a reduce task, partitions that have been assigned to the reduce task are received from all map tasks, all of the records in all partitions are sorted and merged into a single file, then a reduce method that has been defined by a user reads the file and the output of the reduce method is written into a file.

MapReduce operations work on a distributed file system (DFS). Basically, files that input or output in any operation on MapReduce are put on DFS even

M.A. Sharaf et al. (Eds.): ADC 2015, LNCS 9093, pp. 117–128, 2015.
DOI: 10.1007/978-3-319-19548-3_10

temporary files. So, data transfer between tasks or nodes goes through the DFS. In other words, every data transfer uses both network traffic and disk I/O.

The join operation is one of the most important and the most frequently used operations in query processing in relational databases. There also have been many studies of join operations in a distributed parallel environment. In this paper, we focus especially on the low-selectivity equi-join. An equi-join must determine the intersection between two sets of join keys and then have matching records that have the same key in the intersection. Low-selectivity means that the column is highly selective, in other words, there are a lot of variations in the values in the column. Thus, in general, a low-selectivity equi-join takes very large input relations but the output size is very small.

A low-selectivity equi-join is often used when joining between relations created in different management domains, such as when Linked Open Data [4] is used. In the query processing for Linked Open Data, querying users do not know the structure of the datasets and they also do not know whether a dataset can join to another dataset despite the fact that those datasets are targets of the query processing. Thus, they have to query those datasets to determine the structures or decide whether those datasets can be joined. For example, if you want to make a join between SDBS[1], which is a spectral database for organic compounds, and DBpedia[2], which is a linked dataset that is generated from Wikipedia, then you have to try an equi-join operation using the CAS registry number, which is a unique numerical identifier assigned to every chemical substance. The number of compounds in SDBS is 34,000 and the number of things of DBpedia is 4.58 million[3] but the number of chemical substances assigned a CAS registry number in DBpedia is 1,774[4], and all of them may possibly be contained in SDBS, thus the selectivities are 0.05 and 3.9×10^{-6}, respectively. So the selectivity is expected to be very low.

Several approaches for low-selectivity equi-joins have already been proposed, such as *semi-join*[5] and *Bloom filter join*[14]. All of them are approaches that focus on decreasing the amount of data transfer among nodes by removing unnecessary data before transfer using filters. In MapReduce, the data that is output by a job is written in the distributed file system. Thus a decreasing in the output data of a job leads to decreasing both the amount of network traffic and the amount of disk I/O, and thus directly improves the performance. So in order to reduce the output data, the dangling records that are not contained in the results and will not be used in the following jobs are deleted and are not output in temporary files received by the next job. Thus the more decrease in the number of dangling records sent on to the next job, the more the improvement in the performance. In the processing of a low-selectivity equi-join, there are a lot of dangling records. In other words, there is room for improvement in the low-selectivity equi-join.

[1] http://sdbs.db.aist.go.jp/
[2] http://wiki.dbpedia.org/
[3] http://blog.dbpedia.org/2014/09/09/dbpedia-version-2014-released/
[4] http://en.wikipedia.org/wiki/Category:Chemical_pages_needing_a_CAS_Registry_Number

In this paper, we extend the state-of-the-art join algorithms for MapReduce to introduce an index based on position. Per-split semi-join[5] have been proposed for low-selectivity joins, which reads a relation L to extract unique keys, does a semi-join per split of another relation R base on the unique keys, and then finally does an actual join between the results of the semi-join and a split of the relation L. However, we found there are three improvable problems in the per-split semi-join: the number of jobs, the amount of disk I/O, and the amount of network transfer. In particular, because the approach requires three jobs, the I/O cost must be large because the relation L must be read twice, and thus there is room for decreasing of the amount of network transfer because it currently uses keys as they are for the filters' data structure. Therefore, we extend the state-of-the-art in order to address this aspect as well as other factors contributing to the three problems.

Moreover, we considered that if the three or more input relations are joined at one time, then the join selectivity becomes low. However, the per-split semi-join does not support three or more inputs, but the reduce-side joins often used in conventional approaches can be easily extended. Thus we also extend a reduce-side join in order to be able to handle three or more inputs and apply that aspect of the extension of the per-split semi-join.

The remainder of this paper is structured as follows. Section 2 explains several join algorithms as related work. In Section 3, we propose our extensions of the existing algorithms. In Section 4, we evaluate the performance of our extensions through a series of experiments. Finally, we conclude in Section 5.

2 Join Algorithms

So far, many join algorithms using MapReduce have already been proposed [5, 7, 11, 13, 16]. We classify join algorithms using MapReduce into three categories: *reduce-side joins*, *map-side joins*, and *filtering joins*.

2.1 Reduce-Side Join

Reduce-side joins are the most commonly used join strategies using MapReduce; their concept is based on an idea proposed about 30 years ago by [9, 12]. The concept is that records of input relations are partitioned according to the hash values of the join keys, and then a join is performed for each node in parallel.

Standard repartition join[5] is one of the most general methods, and is mentioned in many studies [1, 2, 11, 16, 17], where it may be called a reduce-side join or a different name. Following methods introduced in this section are also have several aliases, but we omit the aliases. Standard repartition join uses both a map task and a reduce task. In the map task, each record of the input relations is read, the name of the input relation is tagged with a value to identify which input the record is from, and then the tagged record is shuffled. In the map task, each record of the input relations is read and then the record is shuffled. In the reduce task, the reduce method receives a key and a list of all records that have

the same key, builds buffers to store the records for each relation, and then takes equi-join between the buffers. In this approach, all records with the same key must fit in memory to store them into buffers.

Improved repartition join (IRJ) [5,10,16] was proposed in order to address the problem. In the improved repartition join, values from one relation are guaranteed to occur first in the list of values received by the reduce method, then those from the other relation occur, because this approach sorts values based on composite keys, that is, a comprised of a key and its tag. From this, a buffer for the first relation can be built completely. Then, you can probe the buffer using the values from the other relation.

2.2 Map-Side Join

A Map-side join, which is implemented as a map-only job, is also a well-known join method.

Directed join [3,5,16,17] assumes that input relations are pre-partitioned based on the join keys using the same hash function, and thus all records with the same key exist on a computing node. First, the initialization method in the map task reads a pre-partitioned split of a relation and builds a hash table. Next, the map method scans each record of a pre-partitioned split of another relation, and probes the hash table to match the record.

Fragment replicate join [2,3,10,16] is another map-side join and assumes that the size of one relation must be as small as possible to fit in memory because the relation is broadcast to all nodes and is stored in the memory on the received nodes. Each map task builds a hash table to store the broadcast relation in the initialization method, and then probes the hash table for each record of the split of the larger relation in the map method.

The restriction that the amount of a smaller relation must fit in memory is so difficult to meet that some solutions have been proposed. *Reversed map join*[15] builds a hash table for each split of the larger relation in the map method, and then probes the hash table to find the records of the smaller relation in the close method. *Broadcast join*[5] is a hybrid of both the fragment replicate join and the reversed map join. If the size of the smaller relation is smaller than the split of the larger relation, the fragment replicate join is performed. Otherwise, the reversed map join is done.

2.3 Filtering Join

*Filtering join*s are used to decrease the amount of records transferred by not sending dangling records in order to minimize communication costs for an equi-join. Almost all filtering joins are composed of a few jobs. The former job(s) generate(s) filtering information to be used to determine whether a record is necessary for the final results or not, and remove(s) unnecessary records using the filtering information. We call the necessary records *joinable* records. The latter job(s) perform(s) the actual join between joinable records in order to obtain the correct results since the filtered joinable records may contain incorrect records.

Filtering joins incur overhead since they contain some additional processes, and thus the filtering joins must be used only the lower the join selectivity for the performance.

Semi-join[5] is a filtering join that requires two given input relations and three jobs. In the first job, the unique join keys of one relation are extracted and a single file containing the unique keys only is generated. In the second job, the unique key file is broadcast, then the dangling records of the other relation are filtered using the unique key data and the files containing only joinable records are generated. In the third job, the final join is executed by broadcasting of the joinable files.

There are similar approaches with the semi-join[14,16,18], where a *Bloom filter*[6] is used to decrease the communication costs. Using Bloom filters instead of the unique key file of semi-join, the data after filtering may contain some incorrect records, however the amount of data transfer can be decreased. In addition, the approach[14] lets JobTracker merge Bloom filters into a single filter, that is, the process is operated by the first reduce task in the semi-join. As the result, the approach can reduce the number of jobs.

The semi-join and its similar approaches have a problem due to generating the key data using only one process to construct a single filter. If the input relation is large and there are many splits then the generating process becomes a bottleneck. To cope with this problem, *per-split semi-join (PSSJ)*[5] have been proposed. The first job extracts unique join keys and generates a unique key file for each L's split, but does not merge the unique key files into a single file differently from the semi-join. In the second job, a map task reads an R's split and filters dangling records using all of the unique key files, and thus a reduce method reads R's joinable records and then partitions the records based on L's splits. The third job performs the directed join [5] between each R's partition and L's split as the finally actual join.

3 Proposed Algorithms

3.1 Hash-and Position-Based Per-Split Semi-Join

We found that there is room for improvement in the per-split semi-join. We then extended the per-split semi-join by introducing three reductions: a reduction in the total amount of network traffic, a reduction in the number of jobs, and a reduction of the amount of disk I/O.

In order to reduce the amount of network traffic, we introduce a hash-based approach that sends a set of hash values instead of a set of unique keys. We may be able to say that this approach is a kind of Bloom filter, but we do not use a bit array, because we thought that it is difficult to determine the same number of hash functions and the same length of bit array to be used among the entire set of nodes before reading the input data.

In order to reduce the number of jobs, we execute the actual join in the second reduce task instead of the third map task.

In order to reduce the amount of disk I/O, we introduce a position-based index approach. Per-split semi-join reads the relation L twice, once for extracting unique keys and once for the actual join. For this, the second reading can be skipped using the index that is generated at the first reading.

Algorithm 1. Hash-and Position-based Per-Split Semi-Join (Ext-PSSJ)

Job 1: Generate a file with hash values from keys and position for each L's split

```
1  class Mapper
2      run() // Override run method to obtain position
3          setup();
4          for each key & value (k, v) in an L's split L_i do
5              p ← Get k's position in L_i;
6              map(k, v);
7          cleanup();
8      map(k: a key, v: a value from an L's split L_i)
9          h_L ← hash (k);
           // Create a hash table H_L with a hash value
              and its a set of positions
10         P ← probe H_L with h_L;
11         P ← P ∪ p;
12         H_L ← add (h_L, P) to a hash table;
13     cleanup()
14         write (H_L, null); // Output file L_i.hp
```

Job 2: Filter dangling records and perform the actual join

```
15 class Mapper // Use L_i.uk to filter referenced R
16     map(k: a key, v: a value from an R's split R_j)
17         h_R ← hash (k);
18         r ← (k + v); // concatenate key and value
19         H_R ← add (h_R, r) to a hash table;
20     cleanup()
21         L.hp ← all L_i.hp files from Job 1;
22         for each file L_i.hp in L.hp do
23             for each (h_L, P) in L_i.hp do
24                 r ← probe H_R with h_L;
25                 write(R_{L_i}, r + P);
26 class Reducer // Execute the actual join
27     reduce(k: R_{L_i}, V: a list of pairs of a record and a set
       of L's positions with the same key)
28         for each v in V do
29             Extract record r_R and positions P from v;
30             Create map M(position p, records R_R);
31         Sort M by p; // To access in position order
32         for each (p, R_R) in M do
33             Seek p and read record r_L at p in L_i;
34             for each r_R in R_R do
35                 Join r_L and r_R;
```

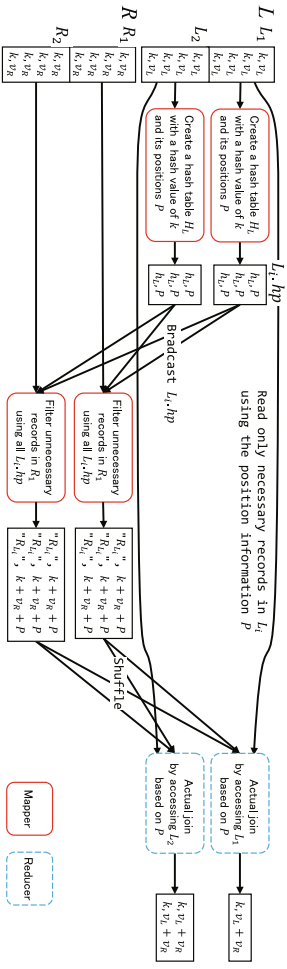


Fig. 1. Ext-PSSJ

We explain our approach shown at Fig. 1 and Algorithm 1. Our approach is composed of two jobs; the first job generates the filters and indices, and the second job filters dangling records and performs the actual join. At the first map task, we overrode the run method to obtain position information p in a split L_i. The map method calculates the hash value of k and a set of all k's positions P and adds them into hash table H_L, and H_L is output as a file $L_i.hp$ in the

cleanup method. In the second job, the map method creates a hash table H_R to store each hash value and its record. The cleanup method filters dangling records using all $L_i.hps$ generated in the first job, and thus outputs R_{L_i} and $r + P$. R_{L_i} is a split name for L, and $r + P$ is the concatenation of a joinable record of R and a set of L_i's positions where the records have the same key with r. First, the data structure must be changed from the list V to a map M with position as the key and its records as values, in order to enable us to sort the map by position order, because from this the second L_i reading can be accessed in the position order for improvement of the performance. Then we seek the position p from which to read the record of L_i. Finally, the actual join is executed.

3.2 Hash-and Position-Based Filtering of a Reduce-Side Join

In filtering joins, the lower the join selectivity is, the more the performance improves. Increasing the number of input relations joined at one join operation can lead to decreasing the join selectivity generally. However, the per-split semi-join and the semi-join do not support three or more input relations. On the other hand, reduce-side joins can easily be extended to handle three or more inputs[5].

Multiple join algorithms in MapReduce have already been studied in [1]. They focus on the multi-way join, that is, relations are joined on different keys. Our focus is on the situation where all the relations are joined on the same key since the purpose of this is to reduce the selectivity.

We extend a reduce-side join to be able to filter dangling records using hash-based filters and position-based indices. Fig. 2 and Algorithm 2 show the algorithm, which is extended based on the improved repartition join[5]. In the first job, the map task overrides the run method in order to obtain position information in the same way as the extension of the per-split semi-join in Algorithm 1. The map method reads a key and a value for each input relation, calculates the hash value h of the key, gets a split name f as a tag, and outputs h as a key and the pair of f and the record's position p as a value. In the reduce task, the reduce method receives a hash value k and a list V of $f + p$ that have the same hash values, extracts a set of all file names F from V, and writes k and V if F contains all input relations. In the second job, the map task also overrides the run method in order to read joinable records and are sorted by the position in the setup method in order to improve the performance of reading disks. The explanation of the following process is omitted because it is similar to that of the improved repartition join[5]. The difference is that our approach is assuming three or more input relations. Thus the actual join performed in the reduce task is different a little. The reduce method inputs the key k and the list of tagged records V, creates buffers $\{B_1, B_2, \dots\}$ for each relation except the last relation, and finally executes the actual joins using the buffers.

4 Experimental Evaluation

We evaluated the performance of our extended approaches through a series of experiments. In all our experiments, we used m1.medium instances of Amazon

Algorithm 2. Hash-and Position-based Filtering of a Reduce-Side Join (Ext-IRJ)

```
Job 1: Generate a file with hash values from keys and positions for each input
1  class Mapper
2      run()// Override run method to obtain position
3          setup();
4          for each key & value (k, v) in all relations do
5              p ← Get current position;
6              map(k, v);
7          cleanup();
8      map(k: a key, v: a value from a split)
9          h ← hash (k);
10         f ← Get split file name;
11         write(h, f + p);
12 class Reducer // Extract intersection keys
13     reduce(k: a key, V: a list of pairs of a file name and a position)
14         F ← Get a set of all file names from V;
15         if F ⊇ {All input relations} then
16             write(k, V)// Output file called U
   Job 2: Filter dangling records and execute actual join
17 class Mapper
18     run()// Random access to joinable records only
19         setup();
20         for position p in U′ do // U′ is from setup()
21             Seek p;
22             (k, v) ← Extract key & value;
23             map(k, v);
24         cleanup();
25     setup()
26         Read U;
27         U′ ← Sort U by position;
28     map(k: a key, v: a value)// joinable records only
29         OMIT: same as IRJ[5]
30 class Partitioner // Partition based on only key
31     OMIT: same as IRJ[5]
32 class Reducer // Execute actual join in a similar way
33     reduce(k: a key, V: a list of tagged records)
34         for v′ in V do // V is sorted by tag
35             (t, v) ← Extract a tag and a value from v′;
36             Create buffers {B1, B2, ...} for each relation except the last;
37             if t is the last relation tag then
38                 Execute actual join among v and {B1, B2, ...};
```

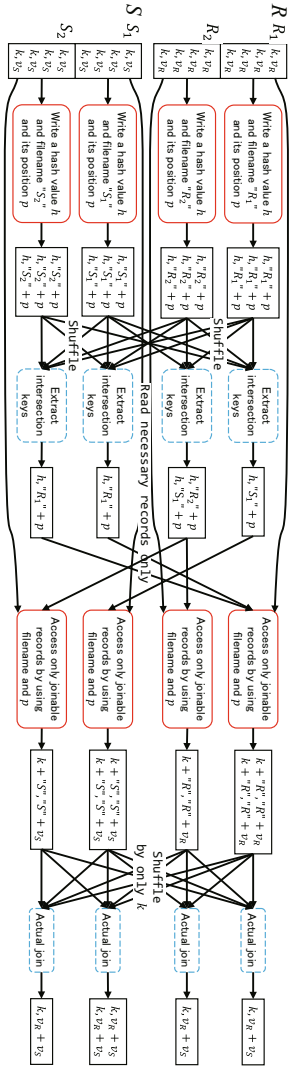


Fig. 2. Ext-IRJ

Web Services (AWS)[5], which has 1 vCPU, 2 EC2 Computing Units (ECU), 3.75 GiB memory, and a 410 GB hard disk storage. Ubuntu 12.04 running Linux 3.2.0 is used as the operating system. The cluster for each experiment is constructed in an availability zone in a region. We used CDH (Cloudera's Distribution including Apache Hadoop) 4.6.0 with Apache Hadoop 2.0.0 running YARN and MR2 and the Oracle Java SE Development Kit 7. We used HDFS as the default configuration, that is, the block size is 128 MB, the number of replications of a block is

[5] http://aws.amazon.com/

three and the maximum size of a split is 128 MB. We used 24 or 48 computing nodes running DataNode and NodeManager, 1 node running ResourceManager, and 1 node running NameNode.

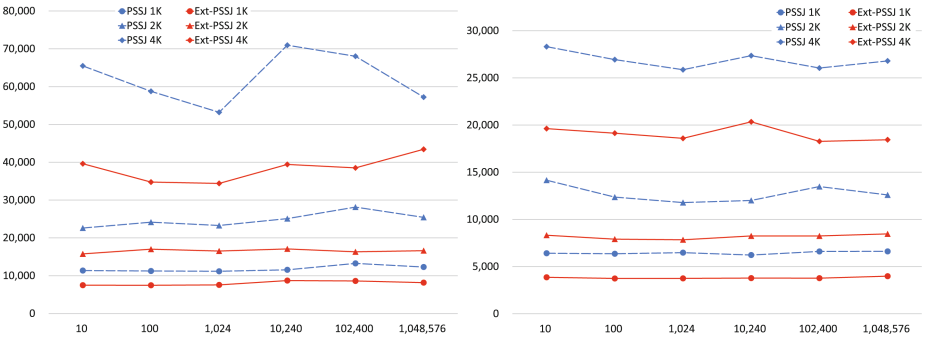
We implemented the per-split semi-join and the improved repartition join proposed in [5] as the state-of-the-art baseline join algorithms and their extensions mentioned in Algorithm 1 and Algorithm 2, and compared them respectively. In this section, we call them PSSJ and Ext-PSSJ, and IRJ and Ext-IRJ.

For the experimental dataset, we randomly generated synthetic datasets for all experiments. For the experiment using two relations, we generate three files (e.g., A, B, and S) that are composed of a list of records with a key and a value, whose sets of keys in the three files are disjoint, and merge them into two relations (e.g., $A+S$ and $B+S$) to control the number of intersections, that is, the join selectivity, so that the result of an equi-join between $A+S$ and $B+S$ is S. In our experiment, the number of records in A and B is fixed at 100 million, while the size of S for the intersection varies from 10 to 1 million. In other words, the join selectivity varies from 1×10^{-7} to 0.01. Thus, these selectivities do not depart from the selectivities of the motivational example of DBpedia and SDBS mentioned in Section 1. The size of the key in each record is fixed at 64 bytes, while the size of each record varied: 1024 bytes, 2048 bytes and 4096 bytes. To compare with PSSJ and Ext-PSSJ, we use two relations. In the comparison of IRJ and Ext-IRJ, we use two, three, and four input relations.

Fig. 3 shows the processing times of PSSJ and its extension. The y-axis is the processing times. The x-axis is the size of the intersection, that is, it means the selectivity varies. And the size of record also varies, and the differences are depicted by the shape of the markers of the lines. The circle means that the size of a record is 1 KB, the triangle means that it is 2 KB, and the diamond means that it is 4 KB. Fig. 3a (right) is the result when using 24 computing nodes, and Fig. 3b (left) is that when using 48 nodes. The number of inputs is two, both of whose size are 100 MB plus the size of the intersections. From these figures, we know that our approach improves on PSSJ as the state-of-the-art regardless of the selectivity. And we know that the larger the size of each record is, the bigger the difference in the performance is.

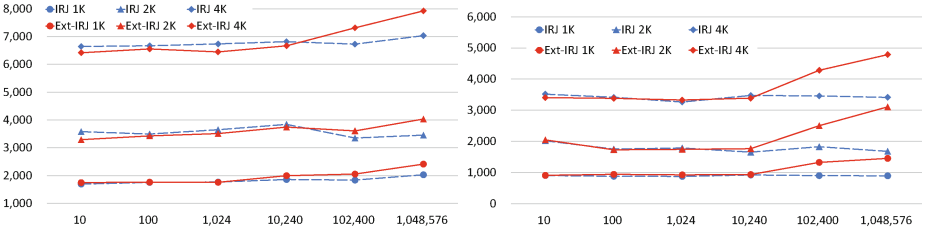
Fig. 4 shows the processing times between IRJ and its extension, Ext-IRJ. The y-axis is the processing times and the x-axis is the size of the intersection. The shape of the markers shows the difference of the size of the record. Fig. 4a show results using 24 computing nodes, and Fig. 4b using 48 nodes. The number of inputs is two, both of whose size is 100 MB plus the size of the intersections. In other words, the conditions are the same as those in Fig. 3. These figures tell us that the difference of the processing times between them is not so large when the selectivity is very low. When the selectivity is high, our approach becomes slow. From only this figure, there is no advantage using our approach over IRJ.

Fig. 5 shows also the processing times between IRJ and Ext-IRJ. The y-axis is the processing times and the x-axis is the size of the intersection. Fig. 4a shows results using 24 computing nodes, and Fig. 4b using 48 nodes. The difference with Fig. 4 is that the size of a record is fixed at 2 KB and the number of input



(a) 24 nodes (b) 48 nodes

Fig. 3. The processing times between PSSJ and Ext-PSSJ



(a) 24 nodes (b) 48 nodes

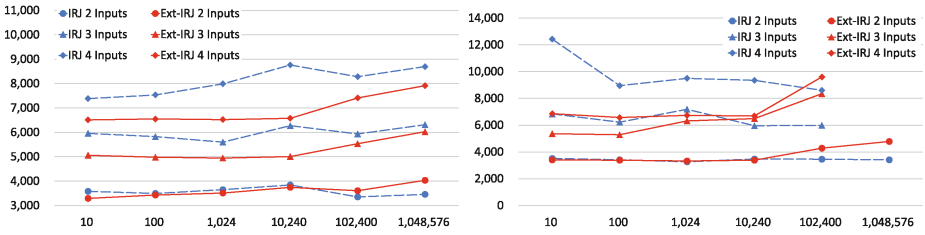
Fig. 4. The processing times between IRJ and Ext-IRJ

relations varies in Fig. 5. The shape of the markers shows the difference in the number of input relations. The circle means that the number of inputs is 2, the triangle means that it is 3, and the diamond means that it is 4. The size of every input is 100 MB plus the size of the intersections. From these figures, we know that the more the number of input relation used in an equi-join operation, the better Ext-IRJ is than IRJ.

Thus, as a result, when the number of input relations are two, our proposed approach, Ext-PSSJ, is better than the state-of-the-art approach, PSSJ. And when the number of input relations are 3 or more, if the selectivity is very low, then our approach, Ext-IRJ, must be used instead of IRJ.

5 Conclusions

MapReduce is one of the most powerful programming models for parallel processing. As a matter of course, join operations that need the most expensive processing cost are also performed in MapReduce. Thus several join algorithms on MapReduce have already been proposed.



(a) 24 nodes (b) 48 nodes

Fig. 5. The processing times between IRJ and Ext-IRJ while #inputs varies

In this paper, we focus on the low-selectivity equi-join, which is frequently used in query processing between datasets for which management domains are different, such as those in Linked Open Data. The per-split semi-join proposed in [5] is the state-of-the-art among low-selectivity equi-joins on MapReduce. However, we thought there is still room for improvement in the approach, and we felt we could achieve three reductions: in the amount of the network traffic, in the number of jobs, and in the amount of disk I/O. We thus proposed an extension of the per-split semi-join that achieves those reductions.

In order to decrease the selectivity of an equi-join, if three or more input relations are given and they will be joined by the same key, then you should execute these joins at one time. However, the per-split semi-join does not support joins using three or more input relations. On the other hand, reduce-side joins can support joins performed three or more input relations at one time. Thus we extended a reduce-side join for low-selectivity joins based on our extension of the per-split semi-join.

We evaluated the performance of our proposed approaches through a series of experiments. As a result, we know our extension of the per-split semi-join always surpasses the state-of-the-art approach per-split semi-join. Moreover, our extension of the reduce-side join is better than a reduce-side join when the number of input relations are three or more, and the selectivity is low.

Acknowledgments. An part of this work was supported by JSPS KAKENHI Grant Numbers 24680010, 24240015.

References

1. Afrati, F.N., Ullman, J.D.: Optimizing joins in a map-reduce environment. In: Proceedings of the 13th International Conference on Extending Database Technology, EDBT 2010, pp. 99–110. ACM, New York (2010). http://doi.acm.org/10.1145/1739041.1739056
2. Andreas, C.: Designing a Parallel Query Engine over Map/Reduce. Master's thesis, Informatics MSc, School of Informatics, University of Edinburgh (2010)
3. Atta, F.: Implementation and Analysis of Join Algorithms to handle skew for the Hadoop Map/Reduce Framework. Master's thesis, Informatics MSc, School of Informatics, University of Edinburgh (2010)

4. Berners-Lee, T.: Linked data (August 27, 2006). http://www.w3.org/DesignIssues/LinkedData.html
5. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 975–986. ACM, New York (2010). http://doi.acm.org/10.1145/1807167.1807273
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970). http://doi.acm.org/10.1145/362686.362692
7. Chandar, J.: Join Algorithms using Map/Reduce. Master's thesis, Informatics MSc, School of Informatics, University of Edinburgh (2010)
8. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating System Design & Implementation, OSDI 2004, vol. 6, p. 10. USENIX Association, Berkeley (2004). http://dl.acm.org/citation.cfm?id=1251254.1251264
9. DeWitt, D.J., Gerber, R.H.: Multiprocessor hash-based join algorithms. In: Proceedings of the 11th International Conference on Very Large Data Bases, VLDB 1985, vol. 11, pp. 151–164. VLDB Endowment (1985). http://dl.acm.org/citation.cfm?id=1286760.1286774
10. Gates, A.: Programming Pig. O'Reilly, Media (September 2011)
11. Jadhav, V., Aghav, J., Dorwani, S.: Join algorithms using mapreduce: a survey. In: International Conference on Electrical Engineering and Computer Science, ICETACS 2013, Coimbatore, pp. 40–44, April 2013
12. Radu, V.: Application. In: Radu, V. (ed.) Stochastic Modeling of Thermal Fatigue Crack Growth. ACM, vol. 1, pp. 63–70. Springer, Heidelberg (2015)
13. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with mapreduce: A survey. SIGMOD Rec. **40**(4), 11–20 (2012). http://doi.acm.org/10.1145/2094114.2094118
14. Lee, T., Kim, K., Kim, H.J.: Join processing using bloom filter in mapreduce. In: Proceedings of the 2012 ACM Research in Applied Computation Symposium, RACS 2012, pp. 100–105. ACM, New York (2012). http://doi.acm.org/10.1145/2401603.2401626
15. Luo, G., Dong, L.: Adaptive join plan generation in hadoop. Tech. rep., Duke University (2010), cPS296.1 Course Project
16. Pigul, A.: Generalized parallel join algorithms and designing cost models. In: Proceedings of the Spring Young Researcher's Colloquium On Database and Information Systems, SYRCoDIS 2012, pp. 29–40 (2012)
17. White, T.: Hadoop: The Definitive Guide. O'Reilly Media / Yahoo Press, 3rd edn. (May 2012)
18. Zhang, C., Wu, L., Li, J.: Efficient processing distributed joins with bloomfilter using mapreduce. International Journal of Grid and Distributed Computing **6**(3), 43–58 (2013)