Expert Systems With Applications 51 (2016) 286–299

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Handling data skew in join algorithms using MapReduce

Jaeseok Myung[a,1,*], Junho Shim[b], Jongheum Yeon[c], Sang-goo Lee[c]

[a] Corporate Design Center, Samsung Electronics Co., Ltd., South Korea
[b] Division of Computer Science, Sookmyung Women's University, South Korea
[c] School of Computer Science and Engineering, Seoul National University, South Korea

ARTICLE INFO

Keywords:
MapReduce
Join algorithm
Skew handling
Multi-dimensional range partitioning

ABSTRACT

One of the major obstacles hindering effective join processing on MapReduce is data skew. Since MapReduce's basic hash-based partitioning method cannot solve the problem properly, two alternatives have been proposed: range-based and randomized methods. However, they still remain some drawbacks: the range-based method does not handle join product skew, and the randomized method performs worse than the basic hash-based partitioning when input relations are not skewed. In this paper, we present a new skew handling method, called multi-dimensional range partitioning (MDRP). The proposed method overcomes the limitations of traditional algorithms in two ways: 1) the number of output records expected at each machine is considered, which leads to better handling of join product skew, and 2) a small number of input records are sampled before the actual join begins so that an efficient execution plan considering the degree of data skew can be created. As a result, in a scalar skew experiment, the proposed join algorithm is about 6.76 times faster than the range-based algorithm when join product skew exists and about 5.14 times than the randomized algorithm when input relations are not skewed. Moreover, through the worst-case analysis, we show that the input and the output imbalances are less than or equal to 2. The proposed algorithm does not require any modification to the original MapReduce environment and is applicable to complex join operations such as theta-joins and multi-way joins.

1. Introduction

Although join is a fundamental database operation, it is not directly supported by the MapReduce framework (Dean & Ghemawat, 2008). This is because (1) the framework is basically designed to process a single input data set, and (2) MapReduce's key-equality based data grouping makes it difficult to support complex join conditions.

One of the major obstacles hindering effective join processing on MapReduce is data skew. As in traditional shared-nothing systems, the processing time of a join operation is dictated by its longest running task. If the underlying data is sufficiently skewed, any of the gains from parallelism will become meaningless.

Although there have been a significant number of proposals for implementing join on MapReduce, this is still an active topic of research with opportunities for improvements. Recent surveys by Lee (2012) and Doulkeridis and Norvag (2014) provide insightful summaries on this problem and proposed solutions. These algorithms can be broadly categorized into three groups according to how the input relations are partitioned; hash-based, range-based and random.

Many of the proposed join algorithms such as repartition join (Beame, 2014) use hash-based partitioning. This may be attributable to the fact that Hadoop – the most popular MapReduce implementation – uses hash partitioning as a default option. However, hash-partitioning cannot handle certain type of skews in the data sets. Tuples with the same value in the join attribute will be hashed to the same reducer, which can lead to serious imbalances across machines, resulting in unacceptable performance.

An alternative method is range-based partitioning (DeWitt, 1992). In this method, the join attribute of one of the input relations is divided into subranges in such a way that all subranges have (approximately) the same number of tuples. Then the same partitioning is applied to the second relation as well. The relation with more data skew is selected for range determination for maximum effect. An attractive aspect of the range-based method is that it is relatively easy to determine *approximate* subranges via sampling, as shown in DeWitt (1992). Thus, with a small cost for sampling, the range-based method can be effective regardless of

* Corresponding author. Tel.: +8228805133.
E-mail addresses: jsmyung@europa.snu.ac.kr (J. Myung), jshim@sookmyung.ac.kr (J. Shim), jonghm@europa.snu.ac.kr (J. Yeon), sglee@europa.snu.ac.kr (S.-g. Lee).
[1] This work was done when the author was a Ph.D. student at Seoul National University.

http://dx.doi.org/10.1016/j.eswa.2015.12.024
0957-4174/

the actual distribution of join attribute values. As a result, range partitioning has been widely used in shared-nothing systems and has been recently adopted for MapReduce (Atta, 2011).

The third method is randomized partitioning (Okcan & Riedewald, 2011). It exploits a cross-product space between the two input relations S and T. Relation S is randomly partitioned into m blocks, while T is done likewise into n blocks. A row in the cross-product space represents one of the m blocks while a column represents one of the n blocks. The cross-product space is covered by k rectangles, each of which represents the blocks covered by a reducer. By keeping the areas covered by the rectangles as even as possible, we can expect similar number of input tuples for each reducer. Random distribution is fairly robust to skews, since repeated (and nearby) join key values will not necessarily aggregate onto one single block. It is also applicable to arbitrary join conditions.

A limitation of the aforementioned range-based method is that the size of join results is not taken into consideration when determining the subranges. This is because range partitioning only exploits samples from the most skewed relation. Join is a binary operation involving two relations and without information on both relations, a serious imbalance, such as join product skew can arise.

The randomized method also has its limitation. Because the tuples are partitioned randomly, each of the m blocks of relations S must be joined with each of the n blocks of relation T. This means that each of the m blocks is duplicated for every reducer whose rectangle intersects its row, and likewise for the n columns. Thus while random selection provides strong load balancing, it does so by incurring high input duplication. The situation always happens regardless of data skew and becomes more serious for larger relations where more reducers must be employed resulting in more duplications. To reduce the duplication, a stochastic method, called M-Bucket, has been proposed by Okcan and Riedewald (2011), but it requires additional MapReduce jobs to collect data statistics.

In this paper, we present a new skew handling method, called multi-dimensional range partitioning (MDRP), to overcome the limitations of the methods. We extend range partitioning by adopting the cross-product space from random partitioning. We use a partitioning matrix instead of the one-dimensional partitioning vector used in traditional range partitioning. Each dimension in the matrix represents an input relation, and a cell represents the cross-product of the subranges of the two relations. Using samples from both relations, we can estimate the workload of a cell in terms of both input and output tuples without requiring full scans of data. Because partitioning is deterministic (range-based), we know exactly which combinations of input blocks will produce output results and, hence, no longer have to duplicate blocks of input tuples for every intersecting reducer as in the randomized method.

In summary, the proposed method provides several benefits: (1) It is efficient. In our scalar skew experiment, our algorithm is about 6.76 times faster than the range-based algorithm when join product skew exists and about 5.14 times than the randomized algorithm when input relations are not skewed. (2) It is scalable. Regardless of the size of input data, we can create subranges that can fit in memory. In addition, the speed-up and scale-up performance is quite fair as we will show with experimental results. (3) It does not require any modification to the original MapReduce environment and can be applied to other join operations such as theta-joins and multi-way joins.

The rest of the paper is organized as follows. We briefly review the MapReduce framework and representative skew handling methods in Section 2. In Section 3, we present our MDRP method and discuss pros and cons of the proposed method. We explain implementation details in a MapReduce environment in Section 4 and validate the proposed method in Section 5 with experiments using Hadoop MapReduce. Section 6 reviews related work, and Section 7 concludes the paper.

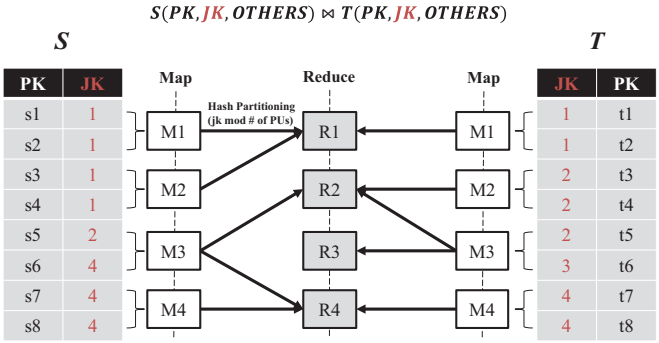


Fig. 1. Hash-based partitioning method.

2. Preliminaries

In this section, we briefly review the MapReduce framework. More importantly, we elaborate limitations of current skew handling methods with intuitive examples.

2.1. MapReduce and effects of data skew

MapReduce provides a simple programming model for large-scale data analysis tasks on a shared-nothing environment (Dean & Ghemawat, 2008). The programming model consists of two primitives, Map and Reduce:

```
map      (k1,v1)        →   list(k2,v2)
reduce   (k2,list(v2))  →   list(k3,v3)
```

The Map function is applied to an individual input record in order to produce a list of intermediate key/value pairs. The Reduce function receives a list of all values with the same *key* and produces a list of new output key/value pairs.

A general and representative join algorithm using MapReduce is the repartition join (Beame, 2014) (a.k.a. reduce-side join, White, 2009). Let us consider a two-way equi-join example as illustrated in Fig. 1:

Example 1 (Repartition join)**.** Relations S and T have eight data tuples respectively, and we have $k = 4$ machines in a cluster. The data schema is very simple, which contains three attribute fields (*pk, jk, others*) where the *jk* attribute is the join attribute. Input relations are split and stored in a distributed file system. Once the join begins, input tuples are fed to a Map function. Let s be an input tuple from S. The Map function takes s and creates a key-value $(s.jk, s)$ pair. Then, the intermediate key-value pair is delivered to a Reduce function in the $(s.jk \bmod k)$th reducer. An input record t from the relation T is processed similarly. Eventually, the Reduce function receives a list of input tuples that have the same *jk* attribute value, and we can use any single-machine join algorithm in order to produce join results.

In the repartition join, data skew can degrade the system performance seriously. With the hash-based partitioning, a reducer may receive too many input records than the other reducers. In our example, a reducer $R1$ receives four input tuples $\{s1, s2, s3, s4\}$ from S and two input tuples $\{t1, t2\}$ from T respectively. The $R1$ reducer has to produce eight output tuples, which is very heavy compared to other reducers. For example, $R3$ receives an input tuple $\{t6\}$ and produces zero output tuples. The completion time of a MapReduce job depends on the last finished reducer. Therefore, handling data skew is very important in join algorithms in MapReduce.

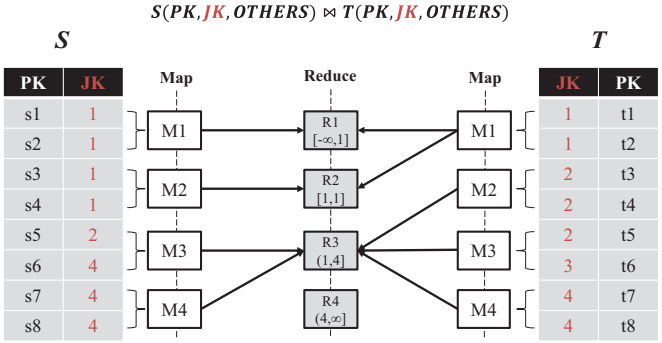
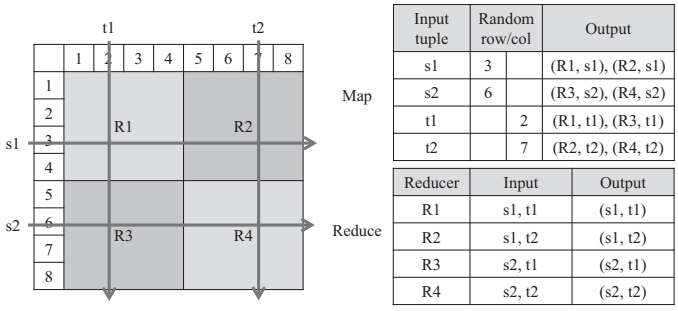


Fig. 2. Range-based partitioning method.



Input tuple	Random row/col		Output
s1	3		(R1, s1), (R2, s1)
s2	6		(R3, s2), (R4, s2)
t1		2	(R1, t1), (R3, t1)
t2		7	(R2, t2), (R4, t2)

Reducer	Input	Output
R1	s1, t1	(s1, t1)
R2	s1, t2	(s1, t2)
R3	s2, t1	(s2, t1)
R4	s2, t2	(s2, t2)

Fig. 3. Randomized partitioning approach.

2.2. Range-based method

The range-based partitioning method is a simple, but effective, solution to the skew handling problem (Atta, 2011; DeWitt, 1992). In this method, one first takes a pilot sample of both relations S and T. By counting the number of repeated samples in each, the most skewed relation can be determined (e.g., in Fig. 2, the most skewed relation is S due to '1' values). Then, we can create a partitioning vector and evaluate the join operation as shown in the following example:

Example 2 (Range partitioning)**.** Let S' be a sorted list of samples from a relation S. In this example, for simplicity reasons, let us assume $S' = S$. Then, we have $S' \cdot jk = \{1, 1, 1, 1, 2, 4, 4, 4\}$. Since there are $k = 4$ reducers, we can select $k - 1$ 'splitting values' which form the partitioning vector. The $\lfloor |S'|/(k-1) \rfloor$th elements are usual selections. In our example, as $\lfloor 8/3 \rfloor = 2$, the 2nd, 4th and 6th elements {1, 1, 4} are selected. The partitioning vector {1, 1, 4} creates 4 sub-ranges: $[-\infty, 1]$, [1, 1], (1, 4] and $(4, \infty]$. Sub-ranges are assigned to different reducers. This contributes to balance the input workloads.

When the join begins, we can use the fragment-replicate technique (Epstein, 1978) . Since $R1$ and $R2$ have sub-ranges $[-\infty, 1]$ and [1, 1], input tuples, where $s \cdot jk = 1$ or $t \cdot jk = 1$, should be fragmented or replicated. To reduce the communication cost, a reasonable heuristic is to fragment the most skewed relation S and replicate the other relation T. Then, {$s1$, $s2$, $s3$, $s4$} are divided into two reducers, and {$t1$, $t2$} are duplicated. As a result, we have 2 reducers that produce 4 output tuples respectively instead of 1 reducer that produces 8 output tuples. Compared to the hash partitioning, the range partitioning allows an input tuple to be assigned two or more reducers. With the fragment-replicate technique, we can divide the expected output workloads.

Our question about the range-based method is that why do we discard samples from the less skewed relation, i.e. T. This can be a reason of serious imbalances. In Example 2, $R3$ has a sub-range (1, 4]. Then, all data tuples, where $jk \neq 1$, are delivered to $R3$. This shows two possible problems: First, input tuples in the less skewed relation cannot be divided into the same size across all reducers; Second, join product skew cannot be even detected during the partitioning phase.

Actually, DeWitt (1992) also proposed a technique, called virtual processor partitioning, to alleviate the join product skew problem. This technique assigns many virtual processors for each actual reducer so that we can create a longer partitioning vector than that of the original range partitioning. Since the longer partitioning vector indicates small size sub-ranges, highly sophisticated controls are enabled. However, a practical question still remains: how many virtual processors are needed? It is difficult to answer the question if we do not know the join selectivity. Obviously, we argue that exploiting samples from both relations can solve these problems.

2.3. Randomized method

A recent research (Okcan & Riedewald, 2011) has shown that a randomized algorithm is effective to handle data skew. In addition, the algorithm works equally well for any join condition that belongs to $\{<, \leq, =, \geq, >, \neq\}$. We briefly review the randomized algorithm with Fig. 3 and the following example:

Example 3 (Random partitioning)**.** We first create an 8 by 8 matrix that represents the cross-product space between two relations S and T. The matrix can be covered by $k = 4$ reducers. To balance reducer's workloads, reducers should cover the same number of cells. Since $|S| = |T| = 8$, the best case is that a relation is divided to $\sqrt{k} = 2$ sub-ranges respectively. Therefore, the optimal partition is shown in Fig. 3. Each reducer covers 16 out of 64 cells in total.

It is relatively easy to use the matrix in a join algorithm. For an input tuple s from S, the Map function finds all reducers intersecting the row corresponding to s in the matrix. A tuple t can be processed in a similar way. A reducer, that intersects with both s and t, is able to produce a join result. Since all reducers have the same number of cells, the number of input tuples is the same. However, the number of output tuples is likely different according to join selectivities for each reducer. This is the reason why the randomized method is proposed.

In this method, a random row s' is selected for a given input row s and pretends as if s corresponds to the s' row. As shown in Fig. 3, the $s1$ tuple is mapped to the row 3. Since the row 3 intersects with $R1$ and $R2$, the Map function creates two intermediate pairs ($R1$, $s1$) and ($R2$, $s1$). The input $s2$ randomly selects the row 6, and the Map creates ($R3$, $s2$) and ($R4$, $s2$). The random rows are represented as horizontal lines. They intersect with a number of vertical lines, each of which represents an input tuple t. Therefore, all combinations between s and t can be evaluated exactly once across all reducers. This completes the join algorithm.

Due to the random selection, there are some random rows and columns that are selected for multiple input tuples, while others are not selected at all. However, the large data size prevents significant variations among rows and columns. According to Okcan and Riedewald (2011), the randomized algorithm practically guarantees that a reducer does not receive more than 1.1 times of its optimal input, and its output is not exceed 1.21 times its optimal size. However, the randomized method cannot avoid high input duplication. Every rows and columns from S and T are duplicated $\sqrt{k}$ times. As the size of an input relation becomes larger, the amount of duplication will also increase significantly. In addition, a large k also leads the high input duplication. This duplication always happens regardless of join conditions and a distribution of join attribute values.

Conceptually, in a cross-product space, some cells can be filtered out according to the join condition (e.g. lower-left and upper-right corners in equi-join cases). We argue that our method can remove non-candidate cells before the actual join begins because we use deterministic sub-ranges. To reduce the input duplication, the M-Bucket-Theta algorithm has already been proposed by Okcan and Riedewald (2011). However, the algorithm requires two MapReduce jobs in order to create an equi-depth histogram. The overhead, caused by the pre-precessing step, can be a burden when input relations are not skewed. Moreover, the algorithm still has a problem to handle join product skew as the algorithm creates a one-dimensional histogram from two different relations.

3. Multi-dimensional range partitioning

We call our new skew handling method as multi-dimensional range partitioning (MDRP). The method is stochastic like the range-based algorithm and the M-Bucket algorithm. We add some new ideas to extend previous stochastic methods and show the effectiveness of our method. In this section, we describe our MDR partitioning method.

3.1. Multi-dimensional range partitioning

3.1.1. Creation of a partitioning matrix

We first consider an equi-join of two relations S and T on a join attribute jk. We have k reducers, and the input relations can be partitioned into k sub-ranges: $\{S_1, S_2, \ldots, S_k\}$ and $\{T_1, T_2, \ldots, T_k\}$. For a relation S, the sub-ranges cover the *domain* of jk from α to β, $\alpha < S \cdot jk \leq \beta$. Two special cases S_1 and S_k cover sub-ranges $[-\infty, \beta]$ and $(\alpha, \infty]$ respectively. The sub-ranges are sorted by their join attribute values. In other words, for all i and j, if $i > j$ then $S_i \cdot \alpha \geq S_j \cdot \alpha$ and $S_i \cdot \beta \geq S_j \cdot \beta$. Boundaries of sub-ranges can be determined by a partitioning vector as in the range partitioning. For example, let us consider two samples S' and T' from S and T relations: $S' = \{1, 1, 1, 1, 2, 4, 4, 4\}$ and $T' = \{1, 1, 2, 2, 2, 3, 4, 4\}$. As we have shown in Example 2, we can select $k - 1 = 3$ splitting values $\{1, 1, 4\}$ and $\{1, 2, 3\}$, resulting $k = 4$ sub-ranges for each relation. As a result, we can create a partitioning matrix M as shown in Fig. 4.

In the partitioning matrix M, ith row represents S_i and jth column is mapped to T_j. A cell (S_i, T_j) is classified into either a candidate cell or a non-candidate cell. Considering the given join condition, non-candidate cells will not produce any join results. In Fig. 4, non-candidate (shaded) cells show the area of not satisfying the equi-join condition. For instance, (S_1, T_2) is mapped to sub-ranges $S : [-\infty, 1]$ and T: (1, 2] which are not overlapped. Therefore, reducers do not have to cover non-candidate cells.

A candidate cell (S_i, T_j) has a value denoted by $M(i, j)$. The value indicates workloads of the cell that has to be processed. In our approach, we determine the workload $M(i, j) = |S'_i \bowtie T'_j|$ where S'_i and T'_j are samples in the sub-ranges. For example, $M(1, 1) = 8$ because we have 4 and 2 samples in S' and T' whose jk attribute values are '1'. Similarly, $M(3, 2) = 3$ because we have $\{s5\}$ and $\{t3, t4, t5\}$. The workloads of the other candidate cells can be computed in the same way.

	T1 [∞-1]	T2 (1-2]	T3 (2-3]	T4 (3-∞]
S1 [∞-1]	8			
S2 (1-1]				
S3 (1-4]		3	0	6
S4 (4-∞]				0

Fig. 4. An example of a partitioning matrix.

Then, the partitioning matrix can be used for a load balancing algorithm. Before we start a MapReduce job, we can create a mapping between candidate cells and reducers. Reducers should be assigned the same number of cells and workloads as much as possible. When the Map function receives an input tuple s, we find candidate cells that contain $s \cdot jk$ in its sub-ranges, and we obtain a list of reducers according to the mappings between cells and reducers. The load balancing algorithm has a responsibility to balance workloads across reducers.

An interesting aspect of the workload computation is that we do not consider the size of input tuples. The reason of this is every cell has almost the same size of input tuples. When we create a partitioning vector, we select splitting values from a *sorted* list of samples. Therefore, if the number of samples is large enough, every sub-range S_i has almost the same number of input tuples. We will discuss about the enough number of samples in Section 4.1.

It is also notable that $M(i, j) = 0$ should not be ignored for the actual join evaluation. Since we only exploit samples, some join results may not be produced when we create the partitioning matrix. For correctness of join results, all candidate cells have to be assigned to at least one reducer. In our example, reducers have to cover (S_3, T_3) and (S_4, T_4) cells.

3.1.2. Identifying and chopping of heavy cells

In the matrix M, each cell has different workloads. A load balancing algorithm can be used for the same amount of workloads across reducers, but it is sometimes impossible to obtain an optimal mapping between cells and reducers. We now define a heavy cell as follows:

Definition (Heavy cell). A heavy cell in the partitioning matrix M is a cell (S_i, T_j) where

$$\frac{M(i, j)}{\sum_{s=1}^{k} \sum_{t=1}^{k} M(s, t)} \geq \frac{1}{k}$$

is satisfied.

Intuitively, the ratio of heavy cell's workload to the total workloads is greater than or equal to $1/k$. Since, k is the number of reducers, $1/k$ indicates the optimal workload ratio for each reducer. Therefore, if there is a heavy cell, it is impossible to balance the workloads.

For example, in Fig. 4, we have two heavy cells (S_1, T_1) and (S_3, T_4). The total workload is 17, and the optimal workload ratio is $1/4 = 0.25$. Since $M(1, 1) = 8$, the (S_1, T_1) cell is a heavy cell ($8/17 \simeq 0.47 \geq 0.25$). Similarly, (S_3, T_4) is also a heavy cell ($6/17 \simeq 0.35 \geq 0.25$).

For the purpose of load balancing, the heavy cells have to be chopped. We now define ω as the optimal workload, i.e. $\omega = (\sum_{r=1}^{k} \sum_{s=1}^{k} M(r, s))/k$. In our example, $\omega = 17/4 = 4.25$. Then, heavy cells are divided into d cells until $M(i, j)/d \leq \omega$. For instance, (S_1, T_1) and (S_3, T_4) are chopped into 2 non-heavy cells because 8/2 and 6/2 are less than ω. By chopping a heavy cell, we can assign the heavy cell to two or more different reducers. This enables us to use the fragment-replicate technique. In Fig. 5, we can see that heavy cells are divided into several non-heavy cells and assigned to different reducers.

3.1.3. Assigning cells to reducers

We now have a set of non-heavy candidate cells $C = \{c_1, c_2, \ldots, c_{|C|}\}$ to be assigned to reducers. It should be clarified

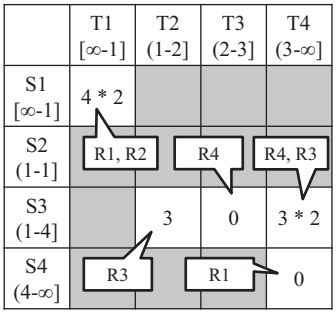


Fig. 5. Mapping between cells and reducers.

that c_l may differ from a cell (S_i, T_j) in the matrix M. The original (S_i, T_j) cell can be divided into several cells, and the C denotes a set of those non-heavy (chopped) cells. Thus, two or more cells c_l and c_m may refer the same cell (S_i, T_j). A cell $c_l \in C$ consists of sub-ranges S_i, T_j and its workload $w(c_l)$. In Example 4, we use $(S_i, T_j, w(c_l))$ in order to refer a cell c_l.

A load balancing algorithm assigns the cells to a set of reducers $R = \{R_1, R_2, \ldots, R_k\}$. Let C_i be a disjoint subset of C assigned to ith reducer R_i. The number of cells in C_i is denoted $|C_i|$. We also use W_i in order to specify the total workloads assigned to the reducer, i.e. $W_i = \Sigma_{c_l \in C_i} w(c_l)$.

In our problem, the load balancing algorithm should satisfy the following constraints: First, all reducers are assigned the same number of cells as much as possible, i.e. $\lambda_{in} = (\max(|C_i|)/\text{avg}(|C_i|)) \simeq 1$; Second, the total workloads for each reducer should be the same as much as possible, i.e. $\lambda_{out} = (\max(W_i)/\text{avg}(W_i)) \simeq 1$. Note that we have two criteria λ_{in} and λ_{out}. The first λ_{in} is about the number of input tuples, and the second λ_{out} is about the number of output tuples. Therefore, if we can satisfy both $\lambda_{in} = 1$ and $\lambda_{out} = 1$, the load balancing algorithm is the optimal algorithm. However, many load balancing algorithms, such as the greedy algorithm and the LPT algorithms (Graham, 1969), only consider a single constraint. Hence, we design a new load balancing algorithm that balances the input and output skew.

In Algorithms 1 and 2, we show our load balancing algorithm. The algorithm is based on a greedy selection of reducers. We first

Algorithm 1 Assign.

Input: $C = \{c_1, c_2, \ldots, c_{|C|}\}$, a set of non-heavy cells
Output: $\{C_1, C_2, \ldots, C_k\}$, mappings (cells and reducers)
1: **for** each cell c_l in C in decreasing order of $w(c_l)$ **do**
2: R_i = getNextReducer()
3: $C_i = C_i \cup \{c_l\}$
4: **end for**
5: **return** $C_1, C_2, \ldots, C_k$

sort cells in C according to their workload $w(c_l)$. For each cell c_l, we select a reducer R_i that has the minimum number of cells and workloads. Once R_i is determined by the `getNextReducer()` function, the cell c_l is added to C_i. We explain more details with Fig. 5:

Example 4 (Load balancing algorithm). We have a set of non-heavy candidate cells $C = \{(S_1, T_1, 4), (S_1, T_1, 4), (S_3, T_2, 3), (S_3, T_4, 3), (S_3, T_4, 3), (S_3, T_3, 0), (S_4, T_4, 0)\}$. Note that we have two $(S_1, T_1, 4)$ and $(S_3, T_4, 3)$ cells because the original cells are chopped into two cells respectively. In addition, cells are sorted according to their workload $w(c_l)$. Since, we have 4 reducers $\{R_1, R_2, R_3, R_4\}$. The set C will be divided into 4 subsets C_1, C_2, C_3, C_4. Initially, all $|C_i| = 0$ and $W_i = 0$. First, we take $(S_1, T_1, 4)$ and a reducer R_1.

Algorithm 2 getNextReducer.

Output: R_{idx}, a reducer to be assigned a cell
1: $minCell = \infty, minLoad = \infty, idx = 0$
2: **for** each reducer R_i **do**
3: **if** $minCell > |C_i|$ **then**
4: $minCell = |C_i|$
5: **end if**
6: **end for**
7: **for** each reducer R_i **do**
8: **if** $minCell = |C_i|$ and $minLoad < W_i$ **then**
9: $idx = i$
10: $minLoad = W_i$
11: **end if**
12: **end for**
13: **return** R_{idx}

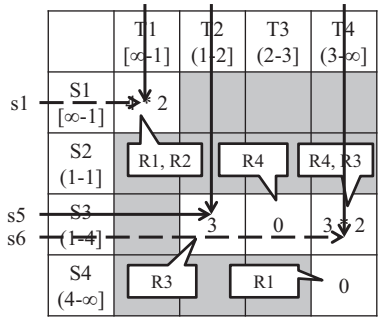


Fig. 6. Join processing with a partitioning matrix.

Then, $C_1 = \{(S_1, T_1, 4)\}$, $|C_1| = 1$ and $W_1 = 4$. Next, we receive another $(S_1, T_1, 4)$, and a reducer R_2 is selected by Algorithm 2. Then, $C_2 = \{(S_1, T_1, 4)\}$, $|C_2| = 1$ and $W_2 = 4$. Similar process is applied to every $c_l \in C$. Finally, $C_1 = \{(S_1, T_1, 4), (S_4, T_4, 0)\}$, $C_2 = \{(S_1, T_1, 4)\}$, $C_3 = \{(S_3, T_2, 3), (S_3, T_4, 3)\}$, and $C_4 = \{(S_3, T_4, 3), (S_3, T_3, 0)\}$. The $\lambda_{in} = (\max(|C_i|)/\text{avg}(|C_i|)) = 2/1.75 = 1.14$ where $\max(|C_1|) = 2$, and $\lambda_{out} = (\max(W_i)/\text{avg}(W_i)) = 6/4.25 = 1.41$ where $\max(W_3) = 6$.

Although the proposed algorithm is a hybrid for both λ_{in} and λ_{out}, we can also consider two extremes for λ_{in} and λ_{out} separately. The first extreme case is to only count the number of cells assigned to each reducer. We can simply implement this first extreme case without changing of Algorithms 1 and 2 by assigning the same workload to $w(c_l)$ for all l cells. This is exactly the same with the traditional greedy algorithm (Graham, 1969). The greedy algorithm guarantees 2-*approximation* compared to the optimal case. However, the first extreme may contain significant output skew. Likewise, the (output-oriented) second extreme can also cause significant input skew. Especially, cells with $w(c_l) = 0$ will be assigned to the same reducer. Therefore, a hybrid method for both input and output is essential.

3.1.4. Join processing using the partitioning matrix

Like the original range-based method, we use the fragment-replicate technique for actual join processing. The mappings between cells and reducers $\{C_1, C_2, \ldots C_k\}$ are distributed and shared to all Map functions. Fig. 6 shows how we use the mappings. Suppose that a Map function receives an input tuple $s1$ from S. Since $s1 \cdot jk = 1$, the input tuple belongs to a cell (S_1, T_1). The cell is mapped to two reducers $R1$ and $R2$. In other words, $(S_1, T_1, 4) \in C_1$ and $(S_1, T_1, 4) \in C_2$. As the tuple belongs to two different reducers, we now determine whether the input tuple should be fragmented or replicated. By counting the number of samples in S_1 and T_1 where $jk = 1$, we can know S has more

repeated values than T. Therefore, we fragment the $s1$ input tuple; one of reducers is randomly selected and the tuple is delivered to the reducer. On the other hand, input tuples from T are replicated to both reducers. In Fig. 6, dashed lines show the fragmentation of input data whereas solid lines represent the replication.

Note that, compared to the original range-based method, our approach provides sophisticated fragment-replicate controls. The range-based method determines an entire relation to be fragmented or replicated. However, our approach makes a decision *cell-by-cell*. For a cell, S can be fragmented, while in the other cell S can be replicated. Generally, S and T have different distributions on the join attribute. Therefore, our approach can handle data skew in both relations.

Another aspect to be mentioned is that we do not duplicate an input tuple for all reducers intersecting with a row in the partitioning matrix. For example, the $s5$ and $s6$ tuples belong to a cell (S_3, T_2) and (S_3, T_4) respectively. We know that the other cells cannot make a join result with given input tuples. Therefore, we only consider reducers mapped to the cell. This enables us to reduce unnecessary communication costs, compared to the randomized method. The M-Bucket algorithm also has similar ideas in terms of reducing the communication costs. However, our method does not require entire scan of input relations and considers both input and output skew.

3.2. Theoretical analysis

The λ_{in} and λ_{out} show the effectiveness of a load balancing algorithm in terms of input and output skew. As shown in Example 4, the values are close to 1. In fact, it is impossible to exactly know the imbalance across reducers because our load balancing algorithm works with samples from entire data sets. However, assuming that we know the exact statistics about input relations, it is possible to roughly guess the upper bound. We analyze the worst case of our algorithm in terms of input and output skew.

Theorem 1 (Input imbalance)**.** *In the worst case, the input imbalance λ_{in} is less or equal to 2.*

Proof. We have $|C|$ cells to be assigned to different reducers. Since we always select a reducer which is the minimum number of cells assigned (greedy algorithm in Algorithms 1 and 2), a reducer is assigned either $\lfloor |C|/k \rfloor$ or $\lceil |C|/k \rceil$ cells. The difference between the maximum and the minimum number of cells is 1. Therefore, the worst case is that a reducer is assigned 2 cells whereas the others are assigned a single cell, i.e. $|C| = k+1$, $\max(|C_i|) = 2$ and $\text{avg}(|C_i|) = (k+1)/k$. Then, the λ_{in} in the worst case is:

$$\lambda_{in} = \frac{\max(|C_i|)}{\text{avg}(|C_i|)} = \frac{2}{(k+1)/k}$$

Since k is a positive integer, $\lambda_{in} \leq 2$. □

Intuitively, in the worst case, the input imbalance does not exceed twice than the optimal (average) case. Note that, the upper bound is still useful even if we do not know the exact statistics of input relations. The unit of load balancing is a cell (sub-range). Therefore, if we partition an input relation into equal sized partitions, the upper bound of load balancing is still valid. Next, we can analyze the output skew, λ_{out}:

Theorem 2 (Output imbalance)**.** *In the worst case, the output imbalance λ_{out} is less or equal to 2.*

Proof. Since we divided a heavy cell, the maximum ratio of a cell does not exceed $1/k$. We now consider a cell $c_1 \in C$ which has the largest $w(c_l)$ among non-heavy cells. Suppose that the workload $w(c_1)$ is $(1/k) - \epsilon$, where ϵ is a small positive real number close to 0. Then, the worst case is that there was a heavy cell which was already chopped into k cells, i.e. $c_2, \ldots c_{k+1}$. The workload of the heavy cell was $\eta = 1 - w(c_1) = (1 - (1/k)) + \epsilon$. Then, $w(c_2) = \ldots = w(c_{k+1}) = \eta/k$. While the optimal (average) workload is $1/k$, the maximum workload for a reducer $max(W_i)$ is:

$$max(W_i) = w(c_1) + w(c_2) = \left(\frac{1}{k} - \epsilon\right) + \left(\frac{(1 - (1/k)) + \epsilon}{k}\right)$$
$$\simeq \left(\frac{1}{k}\right) + \left(\frac{(1 - (1/k))}{k}\right) = \frac{2k-1}{k^2}$$

Therefore, in the worst case, the output skew λ_{out} is:

$$\lambda_{out} = \frac{\max(Wi)}{\text{avg}(Wi)} = \frac{(2k-1)/k^2}{1/k} = \frac{2k-1}{k}$$

Since k is a positive integer, $\lambda_{out} \leq 2$. □

In summary, the essence of our approach is as follows: First, compared with the range partitioning, we create a partitioning matrix instead of a partitioning vector. This enables us to consider the size of join results; second, compared to the randomized algorithm, our approach exploits the partitioning matrix in order to filter out non-candidate (not satisfying given join conditions) cells before the actual join processing. This reduces unnecessary input duplication, leading lower communication cost.

3.3. Complex join conditions

In Section 3.1, we only demonstrated a two-way equi-join example. However, our method is also applicable to other join types, such as theta-joins and multi-way joins. In this subsection, we extend the proposed method to other join types.

Let us consider a general theta-join between two relations. Any theta-join result is a subset of the cross-product. Since the partitioning matrix represents the cross-product space, any join condition can be considered. According to a given join condition θ, candidate cells and non-candidate cells are classified. Using samples from both relations, we can determine workloads of each candidate cell. The rest of processing steps are similar with the equi-join example.

In a theta-join case, an input tuple can belong to two or more cells. Therefore, input duplication can be increased compared to equi-join cases. If the number of duplications is greater than $\sqrt{k}$, the randomized method is more effective than our method. However, in practice, many theta-join queries measure the distance of two tuples. For example, in a spatial database, we usually find nearest neighbors from a query point. In this case, the join condition θ is very selective and almost the same with equi-join queries. Therefore, input duplication is likely smaller than $\sqrt{k}$ in the most cases.

We now consider a special case of multi-way joins, i.e. single-key join, which is a join operation among several relations that have the same join attribute. Actually, the MapReduce programming model is known as a good choice for processing single-key joins. This is because the framework has the key-equality based data flow. Regardless of the number of input relations, input tuples that have the same join attribute value will be delivered to the same reducer. Then, we can use any single machine join algorithm to produce the join results. In our method, we now consider a join among three relations *R, S, T*. They have the same join attribute *JK*. The partitioning matrix is then extended to a three-dimensional partitioning cube. A cell in the cube represents a sub-range considering three relations. Then, we can find heavy cells in the cube, and finally we can process the join operation similarly. For multi-way joins that have several join attributes, we have to perform iterative join processing step by step. Although this iterative approach requires more MapReduce jobs than Afrati and Ullman (2010) and

Zhang (2012), it can reduce the amount of input duplication. Thus, there is a trade-off between the iteration and the duplication.

4. Implementation details

In this section, we describe some of the details of the implementation of our method within the Hadoop MapReduce framework. We begin by explaining how we sampled the relations, and then consider the memory-awareness of the implementation. Details on the Map and the Reduce functions are also provided.

4.1. Sampling

To balance the workloads across reducers, we need to create equal sized cells. Since we determine sub-ranges via sampling, we need a sampling method that guarantees the size of a sub-range. Specifically, we approximate quantiles of given join attribute values which make each sub-range equal sized.

The Kolmogorov statistic (Gibbons, 1997) provides us a probabilistic guarantee for sampling quantiles. Suppose that we want to estimate the median value (at the 50% position in the sorted list of join attribute values). Let α be the proportion of tuples in a relation that have smaller join attribute values than the median value. Let β be the proportion of tuples in the sample that satisfy the same condition. The Kolmogorov's statistic tells us that $|\alpha - \beta| \leq d$ with probability $\geq p$ if the sample size is at least n. d is called the precision and p is the confidence. Given the values of p and d, n can be found using standard tables. Table 1 shows some representative cases, reproduced from Muralikrishna and DeWitt (1988). For example, if we take 26,575 samples and select a value appeared at the 50% position, then with 99% confidence the value appears between 49% and 51% in the original relation. The Kolmogorov statistic works equally well for any distribution and used in many literatures (DeWitt, 1991; Muralikrishna & DeWitt, 1988). In our experiment, we use $p = 0.99$, $d = 0.01$ and $n = 26,575$.

In our implementation, we assume that a relation is stored in HDFS with a random order on a join attribute. The HDFS partitions a relation into multiple splits according to a given block size (64 MB). If q split needs to take n samples, each split takes n/q samples. Using HDFS's API, we call getSplits() method to obtain a list of splits. After that, we take n/q samples from the top of each split. Since we assume that the relation is sorted in a random order on the join attribute, this sampling method is sufficient for our purposes.

4.2. Memory-awareness

Given k reducers, our method initially creates k^2 cells in the partitioning matrix. Since the MapReduce framework is supposed to handle very large data sets, the size of a cell is sometimes too big to fit in memory. We can avoid this situation by making the implementation 'memory-aware'. Suppose that we have a memory limit m, the maximum number of input tuples fit in the memory. Two input relations S and T use the memory $m/2$ respectively. Given the maximum size of a relation $\max(|S|, |T|)$, we can simply select the number of sub-ranges for each relation, i.e. $p = \lceil \max(|S|, |T|)/(m/2) \rceil$. Intuitively, p indicates the maximum number of input tuples from a relation. Then, the number of sub-ranges increases from k to p, and the partitioning matrix contains p^2 cells which fit in the memory.

Table 1
Required sample size.

d: precision/p: confidence	0.90	0.95	0.98	0.99
0.10	149	185	234	266
0.05	596	740	937	1,063
0.01	14,900	18,500	23,425	26,575

To determine p, we have to know the size of input relations $|S|$ and $|T|$. One possible option is to maintain data statistics as shown in the random algorithm (Okcan & Riedewald, 2011). However, in Hadoop, we can effectively estimate the size with small samples. When we take samples as described in Section 4.1, we use the RecordReader class (HDFS's API) which is connected with input data. The class provides the getProgress() function which informs the progress of data loading. For example, once we read n samples from a relation S, the function returns $n/|S|$. Let $n/|S| = \gamma$. Then, $|S|$ becomes n/γ where n and γ are known values.

4.3. The map and the reduce functions

Before we start a MapReduce job, we create a partitioning matrix via sampling as described in above subsections. When the MapReduce job begins, we copy the partitioning matrix into all mappers and reducers. The DistributedCache (White, 2009) mechanism is employed in order to share the partitioning matrix. Then, we can load the partitioning matrix from the setup() method of each mapper and reducer.

We now explain the implementation of the Map and the Reduce functions. Algorithm 3 shows the Map function. For simplicity

Algorithm 3 Map.

```
Input: an input tuple s ∈ S
Input: a partitioning matrix M
1:  listCell = M.listCell(s.jk, S)
2:  for each cell c in listCell do
3:    R = c.listReducer()
4:    if M.fragment(s.jk, S) == true then
5:      i = random() % |R|
6:      output (R[i], (s, S))
7:    else
8:      for i=1 to |R| do
9:        output (R[i], (s, S))
10:     end for
11:   end if
12: end for
```

Algorithm 4 Reduce.

```
Input: (reducerId, [(s1, S),(s2, S),...,(t1, T),(t2, T),...])
1: tupleS = {}
2: for each (si,S) in input list do
3:   tupleS = tupleS ∪ {si}
4: end for
5: for each (ti,T) in input list do
6:   for each sj in tupleS do
7:     output si ⋈ tj
8:   end for
9: end for
```

reasons, the Map function receives an input tuple from S. T is processed similarly. The Map function accesses the partitioning matrix which is loaded during the initialization of the mapper. M is a p by p partitioning matrix that contains candidate cells. In the $M \cdot listCell$ function, we examine the input tuple $s \cdot jk$ whether it satisfies the join condition with regard to cell's sub-range. Satisfied cells are returned by the *listCell* function (line 1). Since we already mapped cells and reducers, we can easily obtain a list of reducers for each cell c in *listCell* (lines 2–3). The next step is to determine

the tuple to be fragmented or replicated (line 4). As discussed in Section 3.1.4, it depends on the existence of more repeated values. In other words, S or T can either be a build input or a probe input. If an input tuple s would be fragmented, we can select a random reducer and deliver the input tuple to the reducer (lines 5–6). Otherwise, the input tuple is copied to all reducers that are assigned to a cell (lines 8–10).

The Reduce function receives a list of input tuples from both relations S and T. Input tuples are sorted by the input relation, i.e. tuples from S are followed by tuples from T. This is enabled by the 'secondary sort' feature of the Hadoop MapReduce framework. We use the setGroupingComparatorClass() method in order to sort input tuples in the Reduce function. The input tuples from S are used as a build input whereas the input tuples from T are used as a probe input. Thus, we first create an empty set for the S relation and then add all S tuples into the set *tupleS* (lines 1–4). After that, for each combination of input s and t, we evaluate the join condition and produce an output tuple (lines 5–9).

5. Experiments

To verify the effectiveness of our algorithm, we conduct experiments on a 13-machine Hadoop-1.1.2 cluster with both real and synthetic data sets. One machine served as the master node (Namenode and JobTracker), while the other 12 were the worker nodes (Datanode and TaskTracker). Every node has a i7-3820 (quad core) processor, 10 MB cache, 32 GB RAM, a 3.5 TB hard disk. In total, the test bed has 48 cores with 8 GB memory per core available for Map and Reduce tasks. Each core is configured to run one map and one reduce task concurrently. All machines are directly connected to the same Gigabit network switch, and the distributed file system block size is set to 64 MB. We run each experiment three times and report the average execution time.

In our experiments, we present results for following data sets:

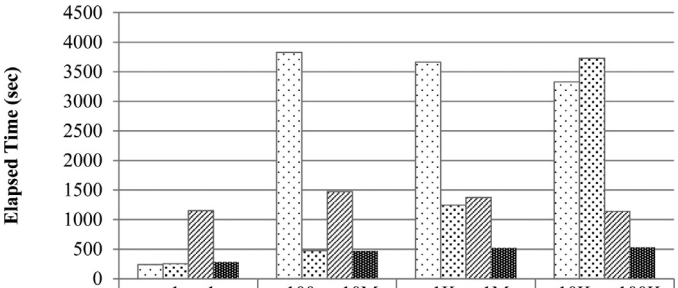
Scalar skew (xα): For a fixed α, we create a 100M tuple relation. The α is the number of tuples in which the value '1' appears in the join attribute (these tuples are chosen at random). The other tuples have a join attribute value chosen randomly from 2 to 100M. The data schema is very simple, which contains three attribute fields (*pk, jk, others*). The size of each tuple is 200 bytes. Thus, each relation occupies approximately 20GB (200 bytes × 100M) of disk space. The term 'scalar skew' is used in DeWitt (1992), and this data set helps us to understand exactly what experiment is being performed. This is also used in many skew handling related researches Atta (2011), DeWitt (1992), Harada and Kitsuregawa (1995) and Walton (1991).

Zipf's Skew (zβ): For a 100M tuple relation R with a domain of D distinct values, the ith distinct join attribute value, for $1 \le i \le D$, has the number of tuples given by the expression

$$|D_i| = \frac{|R|}{i^{\beta} \times \sum_{j=1}^{D} 1/j^{\beta}}$$

where β is the skew factor. When $\beta = 0$, the distribution becomes *uniform*. With $\beta = 1$, it corresponds to the pure Zipf distribution (Lu & Tan, 1994).

Cloud: This is a real data set containing extended cloud reports from ships and land stations (Hahn & Warren, 1999). There are 382 million records, each with 28 attributes, resulting in a total data size of 28.8 GB. This data set is used by Okcan and Riedewald (2011) in order to demonstrate the performance of the randomized algorithm. Thus, we selected this data set for a comparison purpose.



	x1⋈x1	x100⋈x10M	x1K⋈x1M	x10K⋈x100K
HASH	240	3826	3663	3325
RANGE	252	472	1243	3727
RANDOM	1151	1470	1373	1139
MDR	281	468	521	532

Fig. 7. Performance on scalar skew data sets.

5.1. Scalar skew experiments

Fig. 7 shows results for computing an equi-join on scalar skew data sets. We compare four partitioning approaches (HASH, RANGE, RANDOM and MDRP) with four different cases. For a fair comparison, we considered the followings: (1) The RANGE algorithm is an implementation of the virtual processor range partitioning (DeWitt, 1992) which is an improvement of the original range partitioning for handling join product skew. However, there is no further researches on the number of virtual processors. Therefore, we use 60 virtual processors for each reducers, which follows the best parameter in DeWitt (1992). (2) The RANDOM algorithm is an implementation of the 1-Bucket-Theta algorithm proposed in Okcan and Riedewald (2011). The algorithm is the state-of-the-art theta-join algorithm in MapReduce. However, it requires a square matrix for an optimal mapping. As a result, in this experiment, we only use 36 reducers for all algorithms even though we have 48 cores in total.

The first case (x1 ⋈ x1) shows the zero skew case. We see that HASH shows the best performance among all algorithms. A main reason of this is that HASH does not incur the overhead of sampling, creating a partitioning matrix, and assigning cells to reducers. In addition, the Map function only needs to compute a hash function in order to determine a destination reducer, while the other skew handling algorithms have to search a matrix for appropriate cells. However, compared to the HASH algorithm, RANGE and MDRP also show the similar elapsed time. Thus, we can see that the overhead of sampling is not significant. Considering gains from skewed test cases, this difference seems to be acceptable. Finally, the RANDOM algorithm is the worst choice because of its high input duplication.

The other cases (x100 ⋈ x10M, x1K ⋈ x1M, x10K ⋈ x100K) show results when join product skew exists. The number of output tuples is greater than 1 billion, and the output size is over 400 GB. In this experiment, we vary the degree of skew with keeping the total number of output tuples. Obviously, detecting join product skew from samples becomes more difficult from the second case to the last case. The HASH algorithm is a baseline which does not handle data skew.

In the second case, RANGE and MDRP algorithms are effective to handle data skew similarly. The RANGE algorithm uses samples from the x10M data set which leads a number of '1's in its partitioning vector. In the third case, the RANGE algorithm collect samples from x1M data set, which means less number of reducers produce output tuples whose join attribute values are '1'. As a result, longer elapsed time is required. This trend continues to the last case. In contrast, we can see that the MDRP algorithm shows

Table 2
Load Imbalance in Reducers (MAX/AVG).

	HASH			RANGE			RANDOM			MDRP		
	In	Out	Time	In	Out	Time	In	Out	Time	In	Out	Time
x1 ⋈ x1	1.00	1.00	1.04	1.25	1.25	1.00	1.00	1.00	1.08	1.81	1.96	1.18
x100 ⋈ x10M	2.75	33.11	11.37	1.34	1.18	1.22	1.00	1.13	1.03	1.93	1.08	1.15
x1K ⋈ x1M	1.17	32.85	12.15	1.08	4.82	2.52	1.00	1.05	1.05	1.63	1.25	1.16
x10K ⋈ x100K	1.04	32.83	11.03	2.02	32.81	11.29	1.00	1.02	1.10	1.57	1.09	1.08

Table 3
Data size – # of records (GB).

		x1 ⋈ x1	x100 ⋈ x10M	x1K ⋈ x1M	x10K ⋈ x100K
M-IN	All	200 (42)	200 (42)	200 (42)	200 (42)
M-OUT	HASH	200 (43)	200 (43)	200 (43)	200 (43)
	RANGE	200 (44)	200 (44)	200 (44)	200 (44)
	RANDOM	1200 (263)	1200 (263)	1200 (263)	1200 (263)
	MDRP	200 (44)	200 (44)	238 (52)	261 (57)
R-OUT	All	100 (41)	1090 (449)	1098 (452)	1100 (453)

consistent performance regardless of the degree of skew. It is notable that the RANDOM algorithm is also robust in the presence of data skew. However, its overhead of input duplication is significant, which leads longer elapsed time than that of our approach.

Table 2 explains more details on the experiment. In the table, each algorithm has three columns: In, Out and Time. The In and Out columns correspond to the ratio between the maximum and the optimum (average) number of input/output tuples respectively. The Time column represents the same ratio in terms of the processing time of reducers.

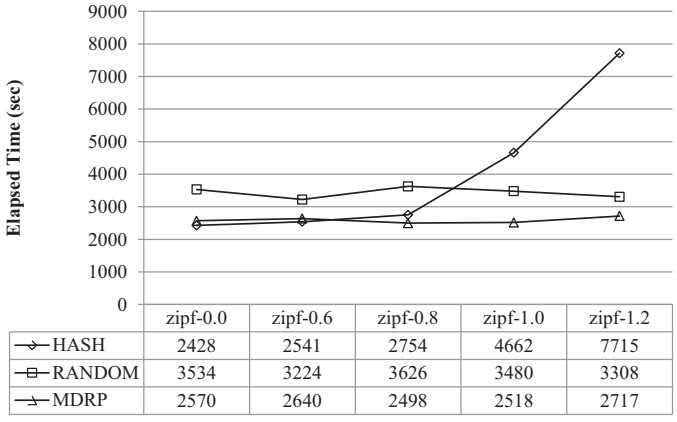
The Out column is most distinguishable than the other columns. We can see that HASH's poor performance is a result of the output skew. The maximum number of output tuples is greater over 30 times than the optimal number of output tuples. The RANGE algorithm detects and handles the output skew in the x100 ⋈ x10M case, but data skew in the other cases are not detected. However, the RANDOM and MDRP algorithms prevent load imbalances in all cases.

In Table 2, it should be mentioned that the RANDOM algorithm is better than MDRP in terms of load balancing. All ratios are close to 1 which is the optimal ratio. However, as shown in Fig. 7, the processing time of MDRP is better than that of RANDOM. This is because the RANDOM algorithm has to process more input tuples than MDRP. In this experiment, each relation contains 100M tuples, and the RANDOM algorithm duplicates a tuple $\sqrt{k} = \sqrt{36} = 6$ times. Therefore, the number of input tuples for reducers is 1.2 billion (600M for each relation). However, MDRP only produced 261M input tuples in total, which is about 1/4 times, compared to the RANDOM algorithm.

Table 3 shows details of data size. M-IN represent the number of records (or bytes) of input data. M-OUT means the total number of output tuples from all mappers. The output tuples are fed to reducers. Finally, R-OUT records are stored into the distributed file system. With these results, we can see that the RANDOM algorithm generates many M-OUT tuples compared to the others.

5.2. Zipf's distribution

In this experiment, we create a pair of data sets (one for each join input). For one data set, the join attribute values are drawn uniformly at random from the range 1–1M. For other data set, we set the skew factor β from 0 (uniform) to 1.2. The skew factor is set to 1.0 in common, but we conduct more experiments in order to see a clear trend in results. Regardless of the skew factor, the number of output tuples is about 10 billion which occupies 4TB of disk space. As the number of tuples in a relation is 100M, each join attribute value has 100 tuples in a relation when the skew factor is 0.



	zipf-0.0	zipf-0.6	zipf-0.8	zipf-1.0	zipf-1.2
HASH	2428	2541	2754	4662	7715
RANDOM	3534	3224	3626	3480	3308
MDRP	2570	2640	2498	2518	2717

Fig. 8. Performance on Zipf's distribution.

In Fig. 8, we change the skew factor of a data set, while the other data set has the same skew factor zipf-0.0. As predicted, we obtained a similar result with the scalar skew experiment. The HASH algorithm is vulnerable to the presence of data skew. The performance sharply degrades when the skew factor is greater than 0.8. This kind of data is very common in real world such as word frequency, citation of papers, Web hits and so on (Newman, 2005). The need for skew handling algorithms is reconfirmed. On the other hand, RANDOM and MDRP show consistent performance regardless of the skew factor. In the elapsed time, the MDRP algorithm outperforms the other algorithms. We do not present results of the RANGE algorithm because the results are similar with that of the MDRP algorithm. Since we only change skew factors of a data set, this result is reasonable.

5.3. Non-equijoin experiments

We now compare our algorithm with the state-of-the-art theta-join algorithm. In Okcan and Riedewald (2011), the Cloud data set is used for evaluation of theta-join queries. Especially, two types of theta-join queries are examined: input-size dominated joins and output-size dominated joins. We employed the same queries for the comparison purpose. The input-size dominated test query is as follow:

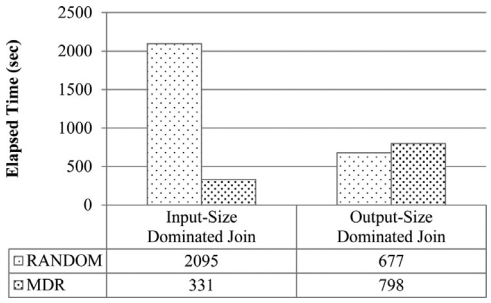


Fig. 9. Performance onnon-equijoin queries.

```
SELECT R.date, R.longitude, R.latitude, S.latitude
FROM Cloud AS R, Cloud AS S
WHERE R.date = S.date AND R.longitude = S.longitude
      AND ABS(S.latitude - T.latitude) <= 10
```

This join produces 390 million output tuples, a much smaller set than the total of 382 × 2 million input tuples. On the other hand, the output-size dominated test query is:

```
SELECT R.latitude, S.latitude
FROM Cloud-5-1 AS R, Cloud-5-2 AS S
WHERE ABS(S.latitude - T.latitude) <= 2
```

For the output dominated query, we take two independent random samples of 5 million records each from Cloud (Cloud-5-1 and Cloud-5-2). The result contains 20 billion output tuples, a much larger set than the total 10 million input tuples. Again, we adopted all data sets and test queries from Okcan and Riedewald (2011).

Fig. 9 shows results of two test queries. We only report results of RANDOM and MDRP algorithms because HASH and RANGE do not deal with non-equijoin queries. In the input-size dominated query, it is clear that the MDRP algorithm outperforms the RANDOM algorithm. Since two algorithms produce the same join results, the difference in elapsed time can be seen the result of input duplications. If the input size grows, the performance gap will be larger and larger. However, in the output dominated query, RANDOM shows the better performance than ours. This is a reasonable result due to the large data size. In practice, the RANDOM algorithm actually provides near optimal solution in terms of load balancing. Our load balancing algorithm though also achieves the similar result.

In a single machine, the cross-product operation requires comparisons of n^2 pairs where n is the input size. In a shared-nothing system, the n^2 comparisons are distributed into multiple machines, but duplication of input tuples cannot be avoided. The MDRP algorithm filters non-candidate cells out so that we can reduce actual comparisons of n^2 pairs and makes smaller duplication compared to the RANDOM algorithm.

5.4. Scalability experiments

Finally, we conduct speed-up and scale-up experiments. The speed-up means that as we add more machines by a certain factor, the time taken to compute a join operation should be decreased by the same factor. To measure the speed-up, we compute a join x1 ⋈ x1 and x10K ⋈ x100K that represent zero skew and skew cases respectively. We increase the number of cores (for mappers and reducers) from 1 to 32. From 1 to 4, we use the vertical scaling technique, i.e. we use a single machine that has multiple cores. From 4 to 32, we use the horizontal scaling technique, i.e. we use 8 machines each of which has 4 cores.

The results are shown in Fig. 10 and Table 4. The 'ideal' line shows the linear speed-up, and we can see that our algorithm

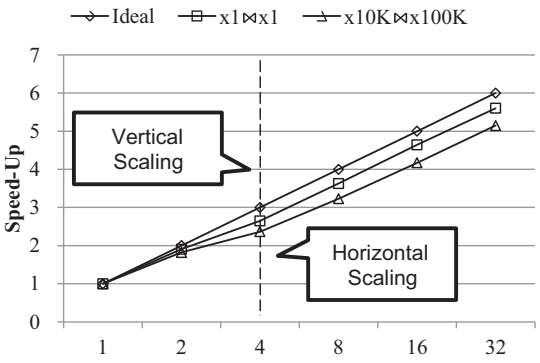


Fig. 10. Speed-up experiments.

Table 4
Speed-up details.

Core	1	2	4	8	16	32
x1 ⋈ x1	8837	4716	2816	1429	707	364
x10K ⋈ x100K	13,863	7845	5380	2956	1537	782

Table 5
Scale-up details.

# of machines	1	2	4	8
Input-size dominated join	2816	3320	3378	4479
Output-size dominated join	5380	3972	3119	2883

also follows the linear speed-up. This trend does not depend on the existence of data skew. However, it is notable that there is a difference between results of vertical and horizontal scaling techniques. In vertical scaling, the performance does not follow the linear speed-up. This indicates that there was an intervention between cores in a single node. In contrast, the performance in horizontal scaling shows the linear speed-up. This means that the input data is small so that the network overhead does not affect the performance.

The scale-up measures that as we add more machines, the size of a task that can be executed in a given time should be increased by the same factor. In our experiment, we add a machine that contains 4 cores, i.e. horizontal scaling. Since we deal with the join operation, we conduct two scale-up tests for input-size dominated joins and output-size dominated joins. For input-size dominated joins, we compute x1 ⋈ x1 where the number of input tuples varies according to the number of machines (from 100M to 800M). In other words, for input-size dominated joins, we assume that each relation has the uniform distribution on the join attribute. On the other hand, in output-size dominated joins, we compute x10K ⋈ xL where L varies from 100K to 800K. Therefore, the number of output tuples is greater than that of input tuples. The input size is fixed to 100M.

The results are shown in Fig. 11 and Table 5. In input-sized dominated join queries, the scale-up ratio is 0.6 when machines are increased from 1 to 8. In output-size dominated join queries, we found out an interesting aspect. The total elapsed time 'MDRP' shows better scale-up than the linear scale-up 'ideal'. This is because we only change the number of output tuples, while the input size is fixed. More mappers can process the same input relations faster. We can see that the 'MDRP' line increases linearly, which means that mappers also show the almost linear scale-up. The 'MDRP-R' line shows the average time for reduce tasks, and it almost follows the linear scale-up. We guess a reason of this is the low network cost.

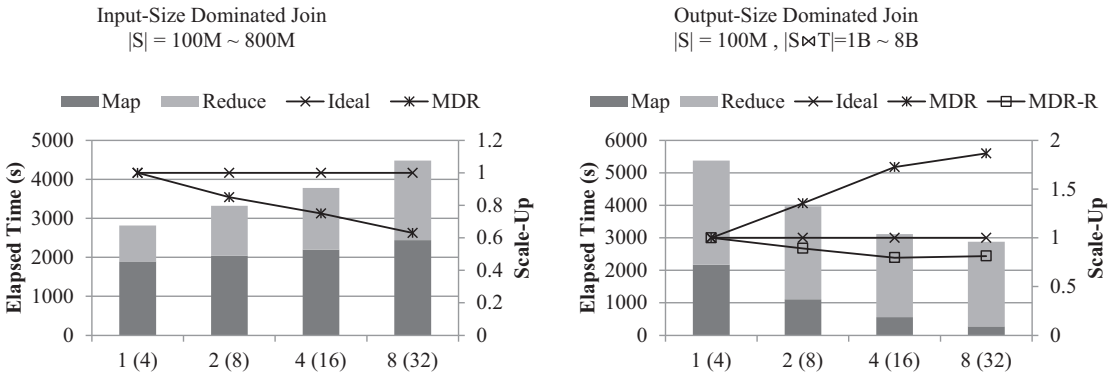


Fig. 11. Scale-up experiments.

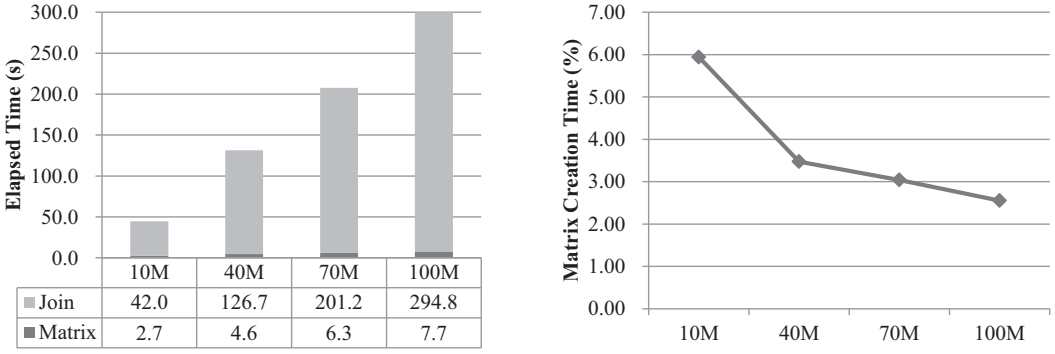


Fig. 12. Elapsed time for the sampling.

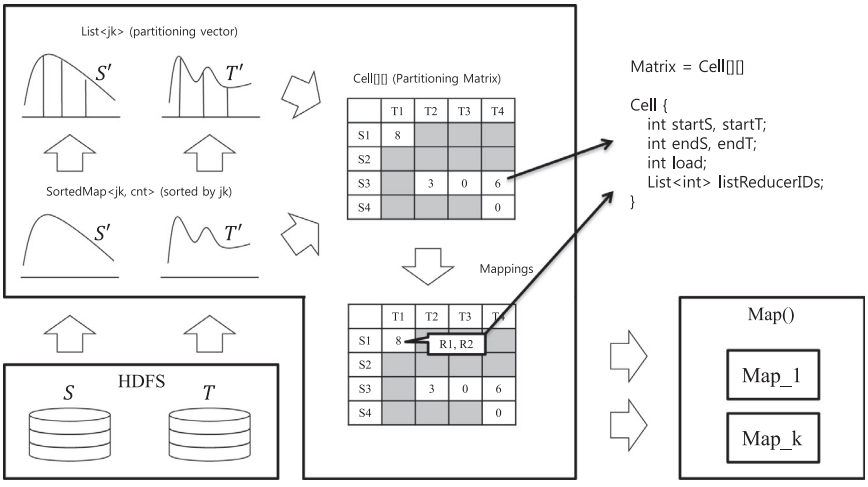


Fig. 13. Processing steps for the sampling.

5.5. Collection of statistics

An important issue in the sampling process is the elapsed time for constructing the partitioning matrix. For this reason, we conduct an experiment to measure the matrix creation time. We compute x1 ⋈ x1 and measure the ratio of the matrix creation time compared to the total elapsed time. The input data sets are not skewed because if there is a skewed value, the ratio of the matrix creation time would be smaller. The results are shown in Fig. 12. We increase the size of input data set from 10M to 100M. Then, the ratio decreases from 6 to 3 percent. We think this ratio is affordable because gains from skewed input data is relatively bigger than the sampling cost.

To understand the sampling process, Fig. 13 shows the processing steps of the partitioning matrix creation. We take samples from relations stored in the HDFS (sampling), and we create the partitioning matrix using the samples (creation). Finally, we distribute the partitioning matrix to all mappers (distribution). Therefore, we can analyze the sampling cost step by step.

Fig. 14 shows processing times for sampling processes. For different input data size $|S|$, the sampling time is increased. For different number of sample size $|S'|$, the sampling time and the creation time is increased. However, the amount of increasing time is relatively small, so is seems to be possible to sample more records in order to create the partitioning matrix. Finally, the number of processing units k is important because the processing

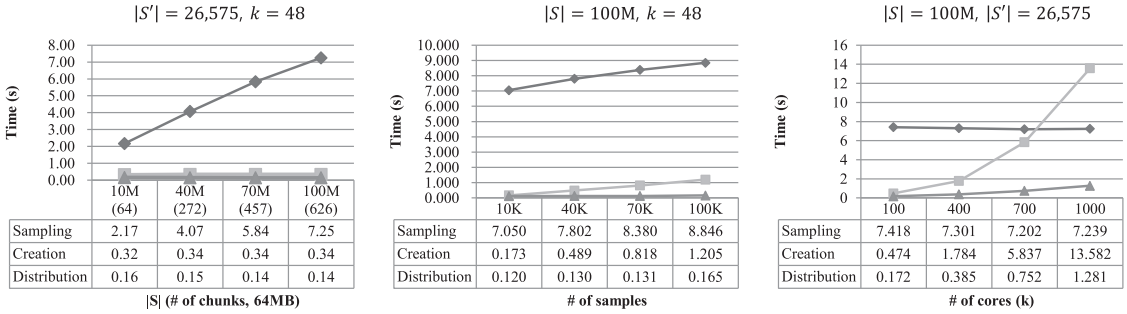


Fig. 14. Processing times for sampling processes.

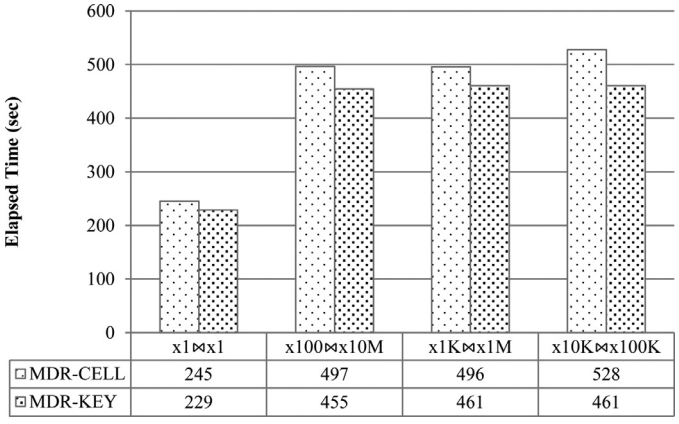


Fig. 15. Exploiting histograms.

time is increased exponentially. When $k = 1000$, it requires almost 15 s to create the partitioning matrix. If input size is small, this can be an overhead.

In MDRP, the sampling process is required in order to create the partitioning matrix. However, if we maintain histograms of input data, we do not have to perform the sampling process. This reduces the processing time of overall join evaluation. Moreover, exploiting histograms is helpful for load balancing. A number of studies have proposed algorithms to create and maintain an equi-depth histogram. In the equi-depth histogram, the number of data records in a bucket is similar with each other. Therefore, the data buckets are similar with a cell in a partitioning matrix, and by assigning buckets to the reducers, we can expect the optimal balance of workloads.

Exploiting histograms provides fine-grained balancing of workloads. Without histogram, we only are able to use a range instead of join attribute value itself. This yields unnecessary data replication as we have shown in the previous subsection. To measure benefits from exploiting histograms, we compare the elapsed times as shown in Fig. 15. MDRP-CELL indicates the original algorithm without exploiting histograms. It is notable that in this experiment, MDRP-CELL does not contain sampling time because we want to measure the effectiveness of fine-grained balancing. MDRP-KEY shows the elapsed time when we exploit existing histogram. Compared to MDRP-CELL we can see that MDRP-KEY shows significant improvements. For example, in x10K ⋈ x100K data sets, the improvement is $(1 - (461/528)) \times 100 = 12.69\%$.

The experimental results show that exploiting of histogram is helpful for join processing. However, building and maintaining a histogram is a difficult problem. In this case, spatial indexing structures can be used as a histogram. For example, the grid file system (Nievergelt, 1984) provides an overview of underlying data. The grid file system creates the grid directory that represents and maintains the dynamic correspondence between record space and data buckets. The grid directory consists of two parts: a dynamic k-dimensional array which contains pointers to data buckets and k one-dimensional arrays which define a partition of a domain. Therefore, we can obtain the exact number of records in a bucket (or a cell) by counting the number of pointers.

Since the grid file maintains sub-ranges of attribute values, it seems to be seen similar with our approach. However, there are many different points: (1) a dimension of the grid file represents an attribute in a relation. In our approach, different dimensions correspond to different relations. (2) The main purpose of our approach is processing of parallel joins instead of creating of persistent indexes. Therefore, our partitioning matrix is created on-the-fly whenever a join operation is evaluated. (3) We introduce the notion of heavy cells and propose a technique to chop the heavy cells. Our heavy cell definition is based on the join selectivity (result size) which is a feature that only exists in the join operation. On the other hand, the spatial indexing structure creates partitions using their sub-ranges. Different sub-ranges can have different width. Hence, the heavy cells can be viewed as a wide sub-range. In join operations, this difference can cause input imbalance across reducers.

6. Related work

6.1. Data skew in parallel joins

Traditional shared-nothing systems already suffered from the skew handling problem. Walton (1991) describes four types of data skew: tuple placement skew (TPS), selectivity skew (SS), redistribution skew (RS) and join product skew (JPS). TPS occurs when different amounts of input tuples are stored across the machines, and we can see SS when the selectivity of selection predicates varies across the machines. RS is seen when there is a mismatch between a distribution of a join attribute and a redistribution mechanism (e.g. poorly designed hash functions). JPS is related to the size of join results. In MapReduce, TPS and SS are not serious problems. For TPS, the Namenode coordinates the entire data sets so that tuples are evenly spread across the Datanodes. For SS, the selection of input tuples is performed by mappers which receive entire input data. This means that SS becomes a problem only when it causes RS or JPS. RS and JPS are problems that have to be considered, and we have shown that our approach is useful for RS and JPS.

There are several recent works in this category. Vitorovic (2015) proposes a load balancing algorithm which considers the properties of input and output data to address RS and JPS. The algorithm assigns the portions of the join keys to machines based on a matrix which contains the statistics on both input and output tuple distributions. Beame (2014) studies the algorithm for processing full conjunctive query(multi-way join query) when the input data have

a skew in a parallel database environment. They use HYPERCUBE algorithm (Beame, 2013) for processing the query, and prove the upper and lower bounds of communication cost when the number of tuples is very much greater than the number of servers. They show upper and lower bounds of the cost in massively parallel communication model and MapReduce model. Polychroniou (2014) introduces a network-optimal join algorithm. The algorithm makes an optimal transfer schedule for each distinct join key based on the information about the join key distribution and the number of tuples to be transferred and computed over the network.

6.2. Skew handling in parallel DBMSs

Before the range-based method and the randomized method, there have already been a number of skew handling algorithms. For example, Kitsuregawa and Ogawa (1990) and Hua and Lee (1991) proposed efficient parallel hash join algorithms. However, a limitation of their algorithms is that they require the relations to be joined are completely scanned before the join begins. Thus, the algorithms may perform much worse than the basic repartition join when the relations are not skewed. The M-Bucket algorithm (Okcan & Riedewald, 2011) also falls into this category. This is the reason why the range-based and randomized methods are widely used in practice.

Although most of the skew handling algorithms estimate the workloads before the actual join operation, there are some algorithms that handle data skew dynamically (Harada & Kitsuregawa, 1995; Shatdal & Naughton, 1993). In other words, they monitor workloads of each machine at run time. However, the monitoring task can also be a burden to systems in zero skew relations. In a recent work (Bruno, 2014), the algorithm also combines multiple partitioning methods including range-based and randomized-based methods into a join strategy which can consider a skewed data for the cloud-scale computation environment. Our method is static but effective regardless of the presence of data skew.

6.3. Skew handling in MapReduce

Handling data skew is an important issue in MapReduce. TopCluster (Gufler, 2012) and SkewTune (Kwon, 2012) are representative researches, but they only consider a single input data set, which is not applicable to join operations. Therefore, a MapReduce-based implementation (Atta, 2011) of the range-based method can be seen as the first skew handling join algorithm in MapReduce. Around the same time, Okcan and Riedewald (2011) proposed the randomized method which is applicable to any join conditions. They also have proposed the M-Bucket algorithm. Zhang (2012) extended the randomized method to multi-way theta-join queries.

Recent studies on this category suggest considering other important issues such as network and data locality. First, some works handle data skew focused on general MapReduce tasks. Slagter (2014) propose an efficient reducer assignment algorithm which exploits distance between nodes and racks for multi-way join. The algorithm performs a join between two largest relations first, and the reducers which contain intermediate results accept required tuples directly from other reducers. Hassan (2014) proposes 2-phase MapReduce for join processing with skewed data. In the first phase, the MapReduce job computes distributed histogram of frequency of join keys. In the second phase, the large join keys are partitioned by using hash and randomized methods, and join results are generated in reduce step. The performance of the algorithm may degrade considerably when the input data is not skewed because the algorithm should read the whole data before join. Gu (2014) categorizes MapReduce join algorithms into Common, Map/Broadcast, Bucket Map, and Sort-Merge-Bucket. They propose selection strategy which chooses a join algorithm with least cost according to data and system environment. Xu (2014) focuses on partitioning intermediate keys to balance the workload of reducers when the keys are skewed in general MapReduce task. They propose two partitioning schemes, cluster combination and cluster split combination. Both schemes use sampling technique to compute the key distribution, and adjust distribution of intermediate keys of reducers. Chen (2014) assigns data cluster to MapReduce nodes according to data locality, so that it makes mapper and reducer access the same data. Its partitioning algorithm is similar to Xu's (2014) approach. Chen (2015) presents the data skew problem, and proposes an effective data cluster split strategy for heterogeneous environment where each node may have different computational power.

7. Conclusion

Handling data skew is essential for efficient join algorithms using MapReduce. The range-based and randomized partitioning methods have been widely used so far, but they have some limitations. In this paper, we proposed a new skew handling method that outperforms traditional algorithms. The proposed method is better than the range-based method when join product skew exists in underlying data. This is because we consider samples from all relations to be joined, while the range-based method only considers samples from the most skewed relation. In addition, our method outperforms the randomized method when the size of input relation becomes large. The reason for this is that the randomized method creates multiple copies of input data regardless of given join conditions.

The proposed method is based on the original Hadoop MapReduce framework. We assume that the framework does not contain a histogram information on a join-key attribute. Having histogram in the framework can surely help the performance as it is shown in our experiments. At the same time creating a histogram by employing extra Map-Reduce job requires additional cost. The trade-off between them should be considered for the practical use of histogram. We also assume that the framework does not perform a dynamic load balancing once a join operation starts. The dynamic load balancing approach requires considerable modification of the original MapReduce framework.

Some future works based on the proposed algorithm can be suggested as follows. Selection of a join-key attribute among multiple join conditions is an important problem in practice. A possible option is to select the least skewed attribute as a join-key attribute. With a small number of samples from all attributes, a candidate join-key attribute can be determined. Efficient maintenance of a histogram is another practical problem in that it can remove our current sampling phase. A separate MapReduce job to compute the histogram on demand seems not practical because joins are common operations in an enterprise computing environment. Therefore, incremental maintenance should be considered in order to exploit the histogram in a join operation.

Acknowledgment

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (R0992-15-1023).

References

Afrati, F. N., & Ullman, J. D. (2010). Optimizing joins in a map-reduce environment. In *Proceedings of the 13th international conference on extending database technology* (pp. 99–110).

Atta, F. (2011). SAND join—A skew handling join algorithm for google's mapreduce framework . In *2011 IEEE 14th international multitopic conference (INMIC)* (pp. 170–175).

Beame, P. (2013). Communication steps for parallel query processing. In *Proceedings of the 32nd symposium on principles of database systems*.
Beame, P. (2014). Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems* (pp. 212–223).
Bruno, N. (2014). Advanced join strategies for large-scale distributed computation. In *Proceedings of the VLDB endowment: vol. 7* (pp. 1484–1495).
Chen, Q. (2015). LIBRA: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on Parallel and Distributed Systems, 26*(9), 2520–2533.
Chen, Y. (2014). *Algorithms and Architectures for Parallel Processing Volume 8630 of the series Lecture Notes in Computer Science*, 229–241.
Dean, J., & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communications of ACM, 51*, 107–113.
DeWitt, J. (1992). Practical skew handling in parallel joins. In *Proceedings of the 18th international conference on very large data bases* (pp. 27–40).
DeWitt, J. (1991). An evaluation of non-equijoin algorithms. In *Proceedings of the 17th international conference on very large data bases* (pp. 443–452).
Doulkeridis, C., & Norvag, K. (2014). A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal, 23*(3), 355–380.
Epstein (1978). Distributed query processing in a relational data base system. In *Proceedings of the 1978 ACM SIGMOD international conference on management of data* (pp. 169–180).
Gibbons, J. D. (1997). *Nonparametric methods for quantitative analysis* (3rd ed.). American Sciences Press.
Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics, 17*(2), 416–429.
Gu, J. (2014). Cost-based join algorithm selection in hadoop. *Web Information Systems Engineering*, 246–261.
Gufler, B. (2012). Load balancing in mapreduce based on scalable cardinality estimates. In Proceedings of the 2012 IEEE 28th international conference on data engineering, (pp. 522–533).
Hahn, C. J., & Warren, S. G. (1999). *Extended edited synoptic cloud reports from ships and land stations over the globe, 1952–1996*. Environmental Sciences Division, Office of Biological and Environmental Research, U.S. Department of Energy.
Harada, L., & Kitsuregawa, M. (1995). Dynamic join product skew handling for hash-joins in shared-nothing database systems. In *Proceedings of the 4th international conference on database systems for advanced applications* (pp. 246–255).
Hassan, M. A. (2014). Handling data-skew effects in join operations using mapreduce. *Procedia Computer Science, 29*, 145–158.
Hua, K. A., & Lee, C. (1991). Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the 17th international conference on very large data bases* (pp. 525–535).
Kitsuregawa, M., & Ogawa, Y. (1990). Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC). In *Proceedings of the 16th international conference on very large data bases* (pp. 210–221).
Kwon, Y. (2012). Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data* (pp. 25–36).
Lee (2012). Parallel data processing with mapreduce: A survey. *SIGMOD Record, 40*(4), 11–20.
Lu, H. J., & Tan, K. L. (1994). Load-balanced join processing in shared-nothing systems. *Journal of Parallel and Distributed Computing, 23*(3), 382–398.
Muralikrishna, M., & DeWitt, D. J. (1988). Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD international conference on management of data* (pp. 28–36).
Newman, M. (2005). Power laws, pareto distributions and zipf's law. *Contemporary Physics, 46*, 323–351.
Nievergelt, J. (1984). The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems, 9*(1), 38–71.
Okcan, A., & Riedewald, M. (2011). Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD international conference on management of data* (pp. 949–960).
Polychroniou, O. (2014). Track join: Distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD international conference on management of data* (pp. 1483–1494).
Shatdal, A., & Naughton, J. F. (1993). Using shared virtual memory for parallel join processing. In *Proceedings of the 1993 ACM SIGMOD international conference on management of data* (pp. 119–128).
Slagter, K. (2014). Smartjoin: A network-aware multiway join for mapreduce. *Cluster Computing, 17*(3), 629–641.
Vitorovic, A. (2015). Load balancing and skew resilience for parallel joins: EPFL report.
Walton, C. B. (1991). A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th international conference on very large data bases* (pp. 537–548).
White, T. (2009). *Hadoop: The definitive guide*. O'Reilly Media, Inc.
Xu, Y. (2014). Balancing reducer workload for skewed data using sampling-based partitioning. *Computers & Electrical Engineering, 40*(2), 675–687.
Zhang (2012). Efficient multi-way theta-join processing using mapreduce. *VLDB Endowment, 5*(11), 1184–1195.