Information Systems 77 (2018) 129–150

Contents lists available at ScienceDirect

Information Systems

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

SharesSkew: An algorithm to handle skew for joins in MapReduce

Foto N. Afrati [a,*], Nikos Stasinopoulos [a], Jeffrey D. Ullman [b], Angelos Vassilakopoulos [a]

[a] National Technical University of Athens, Greece
[b] Stanford University, California, USA

ARTICLE INFO

Article history:
Received 22 August 2016
Revised 13 February 2018
Accepted 4 June 2018
Available online 14 June 2018

Keywords:
MapReduce
Data Skew
Parallel Join Processing

ABSTRACT

In this paper we offer an algorithm which computes the multiway join efficiently in MapReduce even when the data is skewed. Handling skew is one of the major challenges in query processing and computing joins is both important and costly. When data is huge distributed computational platforms must be used. The algorithm *Shares* for computing multiway joins in MapReduce has been shown to be efficient in various scenarios. It optimizes on the communication cost which is the amount of data that is transferred from the mappers to the reducers. However it does not handle skew. Our algorithm distributes Heavy Hitter (HH) valued records separately by using an adaptation of the Shares algorithm to achieve minimum communication cost. HH values of an attribute is decided by our algorithm and depends on the sizes of the relations (or the part of the relations with HH) and how these sizes interrelate with each other. Unlike other recent algorithms for computing multiway joins in MapReduce, which put a constraint on the number of reducers used, our algorithm puts a constraint on the size (number of tuples) of each reducer. We argue that this choice results in even distribution of the data to the reducers. Furthermore, we investigate a family of multiway joins for which a simpler variant of our algorithm can handle skew. We offer closed forms for computing the parameters of our algorithm for chain and symmetric joins.

1. Introduction

We study data skew that occurs when we want to compute a multiway join in a single MapReduce round. When the map phase produces key-value pairs, some keys may receive a significant overload when many tuples having the same value on a specific attribute (*Heavy Hitter*) are present in the data. On the other hand, it is well recognized that in MapReduce, the shuffle phase may add significant overhead if we are not careful with how we distribute the inputs even in the case when all keys receive almost the same amount of inputs [1–3]. The overhead of the shuffle phase depends on the *communication cost* which is the amount of data transferred from the mappers to the reducers. In this paper we develop an algorithm which handles skew in a way that minimizes the communication cost. We will discuss this algorithm in some extend (and with examples) in this introductory section before we conclude with listing all our contributions here.

The algorithm assumes a preliminary round that identifies the Heavy Hitters (HH), e.g., as in [4] or prior knowledge of database statistics [5]. Then it decomposes the given join in a set of *residual* joins, each of which is a version of the original join with two key differences:

- First, it is applied on a different piece of the data. I.e., each residual join is applied on a) the tuples that contain the specific HH values that identify the residual join and b) other non-HH tuples that possibly join with them.
- Furthermore, the map function is slightly different because in each residual join we need to minimize the communication cost under different constraints. These constraints differ because the size of the piece of the data that this residual join is performed on is different.

For each residual join, we use the Shares algorithm ([6]) to compute and minimize the communication cost.

The performance of our algorithm is proven to have the following good properties:

- The overhead that it introduces depends on the number of residual joins (see Section 5.1) and the algorithm is robust to different levels of skewness (see Section 8). This means that the quality of the HH (i.e., the number of tuples that contain each HH) does not affect the performance of the algorithm.

We illustrate the key ideas of the algorithm SharesSkew and how it differs for used practices in systems, in the following examples.

* Corresponding author.
E-mail address: afrati@gmail.com (F.N. Afrati).

https://doi.org/10.1016/j.is.2018.06.005
0306-4379/

Algorithm SharesSkew on 2-way Join

Suppose we have the join $R(A, B) \bowtie S(B, C)$. Systems such as Pig or Hive that implement SQL or relational algebra over MapReduce have mechanisms to deal with joins where there is significant skew (see, e.g., [7–9]).

These systems use a two-round algorithm, where the first round identifies the Heavy Hitters. In the second round, tuples that do not have a Heavy Hitter for the join attribute(s) are handled normally. That is, there is one reducer[1] for each key, which is associated with a value of the join attribute.

Since the key is not a Heavy Hitter, this reducer handles only a small fraction of the tuples, and thus will not cause a problem of skew. For tuples with Heavy Hitters, new keys are created that are handled along with the other keys (normal or those for other Heavy Hitters) in a single MapReduce job. The new keys in these systems are created with a simple technique as in the following example:

Example 1. We have to compute the join $R(A, B) \bowtie S(B, C)$ using a given number, k, of reducers. Suppose value b for attribute B is identified as a Heavy Hitter and that there are r tuples of R with $B = b$ and s tuples of S with $B = b$. Suppose also for convenience that $r > s$.

The distribution to k buckets/reducers is done in earlier approaches by partitioning the data of one of the relations in k buckets (one bucket for each reducer) and sending the data of the other relation to all reducers. Of course since $r > s$, it makes sense to choose relation R to partition. Thus values of attribute A are hashed to k buckets, using a hash function h, and each tuple of relation R with $B = b$ is sent to one reducer – the one that corresponds to the bucket to which the value of the first argument of the tuple was hashed. The tuples of S are sent to all the k reducers. Thus the number of tuples transferred from mappers to reducers is $r + ks$.

The approach described above appears not only in Pig and Hive, but dates back to [10]. The latter work, which looked at a conventional parallel implementation of join, rather than a MapReduce implementation, uses the same (non-optimal) strategy of choosing one side to partition and the other side to replicate.

In Example 2 we show how we can do significantly better than the standard technique of Example 1 and, thus, illustrating our technique.

Example 2. We take again the join $R(A, B) \bowtie S(B, C)$. Remember that $B = b$ is our HH value and all other values for B are non-HHs.

- For $B = b$, we adopt the following strategy: We partition the tuples of R with $B = b$ into x groups and we also partition the tuples of S with $B = b$ into y groups, where $xy = k$. We use one of the k reducers for each pair (i, j) for a group i from R and for a group j from S. Now we are going to partition tuples from both R and S, and we use hash functions h_r and h_s to do the partitioning. We send each tuple (a, b) of R to y reducers using the key (i, q), where $i = h_r(a)$ and $q = 0$ to $y - 1$ (i.e., q ranges over all y groups). Similarly, we send each tuple (b, c) of S to x reducers using the key (q, j), where $j = h_s(c)$ and $q = 0$ to $x - 1$ (i.e., q ranges over all x groups).
  Hence the communication cost for the HH $B = b$ is $ry + sx$. We can show that by minimizing $ry + sx$ under the constraint $xy = k$ we achieve communication cost equal to $2\sqrt{krs}$, which is always less than what we found in Example 1, which was $r + ks$.
- As for the non-HH values, i.e. when $B \neq b$, the join is computed as in a hash join.

[1] In this paper, we use the term *reducer* to denote the application of the Reduce function to a key and its associated list of values. It should not be confused with a Reduce task, which typically executes the Reduce function on many keys and their associated values.

Thus, we have decomposed the original join to two residual joins, one involving the HH value ($B = b$) and the second any other non-HH value for attribute B.

In Fig. 1, we give a graphic representation of the technique for this particular example and show how a few input data are distributed to the reducers. For this graphic example, we use modulo hash functions to partition the data.

Note, in this example, that the value of attribute B, $b_1 = 3$ appears in 3 out of 4 tuples of relation R and in 2 out of 4 tuples of relation S. Hence, it appears in 5 out of 8 tuples so it is a HH value and its skewness level is $5/8 = 0.625$ in this dataset.

The contributions in this paper are:

- We present a MapReduce algorithm (SharesSkew) to handle skew for multiway joins. The algorithm minimizes communication cost and is tested with experiments which testify that the wall-clock time is significantly affected by the communication cost.
- We initiate an investigation to find whether there are families of multiway joins where SharesSkew does not perform significantly better than Shares. Thus, we investigated the performance of SharesSkew algorithm in more detail as concerns how it performs in special classes of multiway joins. We have shown that a) there exist multiway joins that Shares does almost as well as SharesSkew does, even in the presence of skew and b) there exists a class of multiway joins where, Shares has very high communication cost even for random data.

In what follows, when we refer to an *input* we mean a tuple of one of the relations, when we refer to the *size* of a reducer we mean the maximum number of inputs (tuples) that hash to a reducer. Furthermore, we assume a constraint that sets an upper bound q on the size of each reducer. We use the Shares algorithm to compute the shares for each attribute as a function of the total number of reducers, k. After that we bound the number of inputs that are sent to each reducer by q and we compute how many reducers we must use for each residual join. We argue that since the communication cost is increasing with the number of reducers, this is the best strategy compared to distributing all tuples for all residual joins to all reducers. In the rest of the paper we will focus on minimizing the communication cost as a function of the number of reducers. Since the Shares algorithm distributes tuples evenly to the reducers (the hash function we use sees to that), it is straightforward to enforce the constraint that puts an upper bound on the size of each reducer, and thus compute the appropriate number of reducers needed.

The paper is structured as follows:

In the rest of this section we explain the SharesSkew algorithm on the 2-way join and, in the end of the section, we explain our formal setting. In Section 2, related work can be found. In Section 3, an overview of the Shares algorithm from [6] is presented. In Section 4, we give an overview of SharesSkew algorithm and we relate the reducer size (which is a parameter that sets a bound on the number of inputs a reducer can receive and controls the degree of parallelization in the algorithm) and the number of reducers. In Section 5, we present the algorithm SharesSkew, along with its pseudocode in Section 6, and in Section 7 we give an extended example of how to apply this algorithm. Then, in Section 8, we give closed forms for shares (see Section 3 for their definition) and calculate the communication cost for chain joins and symmetric joins. We, thus, show the robustness of our algorithm to different levels of skewness and also we compare the performance of SharesSkew and Shares algorithms. Section 9 explains how the efficiency of SharesSkew is a consequence of the efficiency of Shares algorithm. In Section 10 we discuss the benefits of our algorithm. In Section 11 we provide an extensive experimental evaluation on

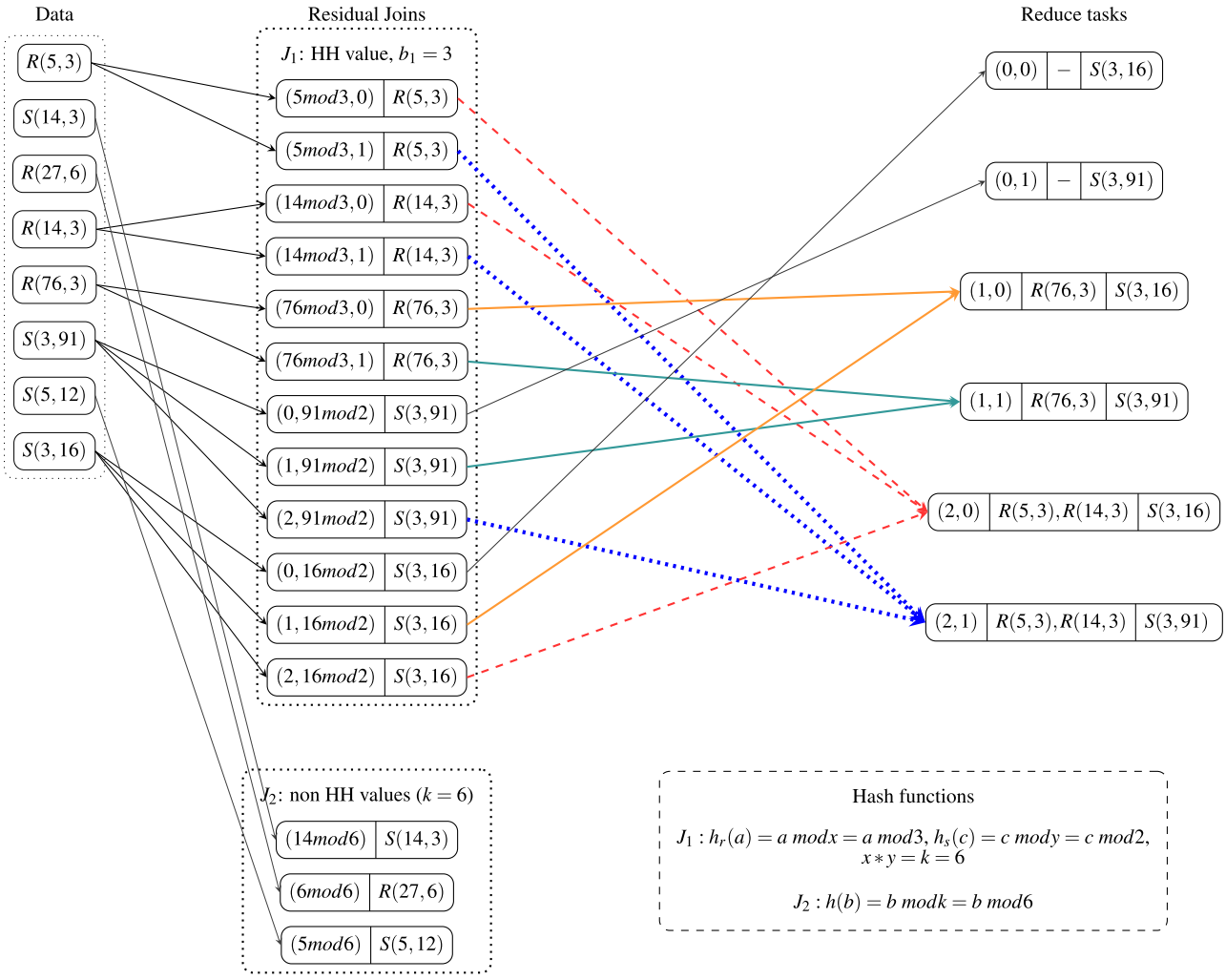

Fig. 1. (Example 2) Applying the SharesSkew algorithm on the join $R(A, B) \bowtie S(B, C)$ with one HH $B = b_1 = 3$.

cluster size effect, parallelization and skew resilience. We conclude the paper in Section 12.

2. Related work

Work that is based on Shares algorithm and addresses data skew issues appears in [4,5,11]. In these works residual joins are formed and the number of reducers is fixed. The algorithms find the shares (that are needed in the Shares algorithm) from a linear program that takes as parameters the sizes of the relations. In [5] statistical information is taken into account whereas in [4,11] the communication cost achieved by the algorithms is worst-case optimal. Our algorithm fixes an upper bound on the number of tuples a reducer is allowed to hold and the communication cost achieved is instance optimal. The works in [4,5,11] find also worst-case lower bounds.

[12] offers an empirical evaluation of the Shares algorithm and also discusses optimizations which are based on a refined technique of how to compute the shares in order to mitigate the effect of non-integer shares. It combines various known techniques (including those in [5]) to investigate how various queries over datasets (including Twitter and Freebase datasets) can be efficiently computed.

The work cited so far is the only work that investigates skew when computing multiway joins in MapReduce. In the rest of this section, we review work either on 2-way joins with skew considerations or on multiway joins without skew considerations. We also review work on the Shares algorithm and recent developments on optimal serial algorithms for computing multiway joins.

A theoretical basis for a family of techniques (including the Shares algorithm) to cope with skew by relating them to geometry is described in [13]. In [2] it is proven that with high probability the Shares algorithm[2] distributes tuples evenly on uniform databases (these are defined precisely in [2] to be databases which resemble the case of random data). This class of databases include databases where all relations have the same size and there is no skew.

It was only a few years ago that an optimal serial algorithm to compute multiway joins was discovered [14]. Previous serial algorithms could be significantly suboptimal when there was significant skew and/or there were many *dangling tuples* (tuples that do not contribute to the result).

In [15] a new algorithm is described that is able to satisfy stronger runtime guarantees than previous join algorithms for data in indexed search trees.

There is a large amount of (recent or not so recent) work on 2-way joins for MapReduce but here we will review only some of it since this work is not closely related to our paper. In our paper the challenge is how to handle multiple relations in a multiway join

[2] called HyperCube algorithm in this paper

efficiently. The case of the 2-way join is used in Section 1 only as an introductory example.

Handling skew in MapReduce joins is considered also in the following papers. In [16,17], Skewtune is introduced as a system to mitigate skewness in real applications on Hadoop.

In [18] and [19], multi-round MapReduce algorithms are considered with careful load balancing techniques.

A comparison of join algorithms for Log processing in MapReduce is provided in [20], where skew is also discussed.

In [21], the authors significantly improve cluster resource utilization and runtime performance of Hive by developing a highly optimized query planner and a highly efficient query execution engine that handles skew as well. Work in [21] is done to facilitate users (as was Hive's original purpose) to pose SQL queries on distributed computation frameworks by hiding from them the details of query execution. This is different from the goal of our paper or the work in [5,12].

The presence of skew has been in focus of the parallel databases systems domain (a detailed taxonomy of approaches can found in [22]). [23] proposes a redistribution and duplication strategy in which data are first moved in appropriate pools on which then computation occurs.

Multiway join in MapReduce without skew considerations Multiway join in MapReduce without skew considerations is examined in [24], where a query optimization scheme is presented for MapReduce computational environments. The query optimizer which is designed to generate an efficient query plan is based on multiway join algorithms. Another system implemented in MapReduce and based on multiway join is presented in [25].

There are a few papers on theta-joins in MapReduce (some addressing skew), including [26] and [27] which focus on optimal distribution of data to the reducers for any theta function.

Efficient multi-round MapReduce algorithms for acyclic multiway joins are also developed recently [28,29]. In [30], a comprehensive survey is provided for large scale data processing mechanisms based on MapReduce.

3. Shares algorithm

The algorithm is based on a schema according to which we distribute the data to a given number of k reducers. Each reducer is defined by a vector, where each component of the vector corresponds to an attribute. The algorithm uses a number of independently chosen random hash functions h_i (where $i \in \{A, B, C\}$, i.e. h_A, h_B, h_C); one for each attribute $X_i \in \{A, B, C\}$. Each tuple is sent to a number of reducers depending on the value of h_i for the specific attribute X_i in this tuple. If X_i is not present in the tuple, then the tuple is sent to all reducers for all h_i values. For an example, suppose we have the 3-way join $R(A, B) \bowtie S(B, C) \bowtie T(C, A)$. In this example each reducer is defined by a vector (a, b, c). A tuple (a, b) of R is sent to a number of reducers and specifically to reducers $(h_A(a), h_B(b), i)$ for all $i = 0$ to $c - 1$. I.e., this tuple needs to be replicated a number of times, and specifically in as many reducers as is the number of buckets into which h_C hashes the values of attribute C.

When the hash function h_i hashes the values of attribute X_i to x_i buckets, we say that the *share* of X_i is x_i. The communication cost is calculated to be, for each relation, the size of the relation times the replication that is needed for each tuple of this relation. This replication can be calculated to be the product of the shares of all the attributes that do not appear in the relation. In order to keep the number of reducers equal to k, we need to calculate the shares so that their product is equal to k.

Thus, in our example, the communication cost is $rc + sa + tb$ and we must have $abc = k$. (We denote the size of a relation R by the lowercase r.) Using the Lagrangean method ([6]), we find the values that minimize the cost expression: $a = (krt/s^2)^{\frac{1}{3}}$, $b = (krs/t^2)^{\frac{1}{3}}$ and $c = (kst/r^2)^{\frac{1}{3}}$, and thus the minimum communication is $3(krst)^{\frac{1}{3}}$. We give a more detailed example of the Lagrangean method in Section 3.1.

3.1. Dominance relation

An attribute A is *dominated* by an attribute B in the join if B appears in all relations where A appears. It is shown in [6] that if an attribute is dominated, then it does not get a share, or, in other words, its share is equal to 1. We will extend the notion of dominance in the context of SharesSkew in Section 5.2.

Now we give an example to illustrate the original Shares algorithm and the dominance relation.

Example 3. We repeat from [6] the example about a 3-way join. So, let $R(A, B)$, $S(B, C)$ and $T(C, D)$ be three binary relations whose join we want to compute, and let r, s and t be their sizes respectively. First we observe that attribute A is dominated by B and D is dominated by C, hence we do not include them in the communication cost expression since each gets share equal to 1. Thus, the communication cost expression that we want to minimize is $ry + s + tx$, where x is the number of shares for attribute B and y is the number of shares for attribute C.

We use the method of Lagrange multipliers to solve and find the x and y that minimize the cost expression $ry + s + tx$ under the constraint $xy = k$. We begin with the equation $ry + s + tx - \lambda(xy - k)$, take partial derivatives with respect to the two variables x and y, and set the results equal to zero. We thus get that: (1) $r = \lambda x$, which implies $ry = \lambda xy = \lambda k$, and (2) $t = \lambda y$, which implies $tx = \lambda xy = \lambda k$.

If we multiply (1) and (2) we get $rtxy = rtk = \lambda^2 k^2$, which implies $\lambda = \sqrt{rt/k}$. From (1) we get $x = \sqrt{kr/t}$ and from (2) we get $y = \sqrt{kt/r}$. Thus the communication cost, which is $ry + tx$, is equal to $\sqrt{2krt}$ when we substitute for x and y. Hence, we proved that $\sqrt{2krt}$ is the optimal communication cost.

Now how do we hash values to the reducers? We were given k reducers.

A tuple $\tau = (u, v)$ of relation S is sent only to one reducer, the reducer $(h_B(u), h_C(v))$ where h_B and h_C are the hash functions that partition the values of attribute B into x buckets and the values of attribute C into y buckets respectively. Now, a tuple $\tau = (u, v)$ of relation R is sent to y reducers, i.e., to all reducers with first component of their vector equal to $h_B(v)$. Similarly a tuple $\tau = (u, v)$ of relation T is sent to x reducers, i.e., to all reducers with second component of their vector equal to $h_C(u)$.

In order to make the example more concrete, suppose we have the following data:

$$R(1,2), R(3,2), R(1,3), R(3,3), R(2,4), R(3,4), R(3,5), R(6,5)$$

$$S(2,2), S(3,2), S(4,4), S(5,4)$$

$$T(2,3), T(4,5)$$

Then the sizes of the relations are: $r = 8, s = 4, t = 2$. Suppose we use 4 reducers. Then we can compute the shares: $y = \sqrt{kt/r} = \sqrt{4 \times 2/8} = 1$ and $x = \sqrt{kr/t} = \sqrt{4 \times 8/2} = 4$. This gives communication cost $ry + s + tx = 8 + 4 + 8 = 20$. This means that the optimal distribution is to use all four reducers to distribute the tuples of relation R, two tuples of R in each reducer and then one S tuple in each reducer accordingly but we send the T tuples to all reducers. Thus each reducer gets $20/4 = 5$ tuples two R tuples, one S tuple and the two T tuples. In more detail, we hash the values of B

to buckets 2,3,4 and 5 (they are the same as their values) and all values of C in one bucket.

In order to give a more interesting numerical calculation, suppose we have $k = 64$ reducers and sizes $r = 400, s = 250, t = 100$. Then we calculate the shares in the optimal solution: $x = 16, y = 4$. Thus the values of the B attribute are hashed in 16 buckets and the values of the C attribute into 4 buckets. The communication cost is $400 \times 4 + 250 + 100 \times 16 = 1600 + 250 + 1600$. The 1600 that appears twice is no coincidence, it is that the optimal solutions appears when all the terms in the cost expression become equal.

4. Overview of SharesSkew algorithm

The Shares algorithm fixes the number of reducers and optimizes under this constraint. In this paper, we do not fix the number of reducers and try to apportion them among what could be an exponential number of different joins as in the Shares algorithm. Instead we fix the reducer size q (i.e., the number of inputs allowed in each reducer) and find how many reducers we should use for each residual join under this constraint.

Thus, q will determine the number of reducers k and the shares for each attribute that participates in the key (see Section 3) for any special case involving HHs (In other words, make the shares big enough that the tuples from each relation are distributed so no reducer gets more than q tuples in total).

4.1. Partitioning Relations

So after identifying HHs for each attribute that is not dominated, we partition each relation into a possibly exponential (in the number of attributes) number of pieces, depending on whether a relation has no HH or which particular HH it has in each of its nondominated attributes (thus we have the *types* that are explained in detail in Section 5.1). Then, we consider in turn each combination of choices for all those nondominated attributes i.e., for each attribute we assign a type which is either a non-HH or a particular HH. Each such combination defines one residual join.

Consider the example $R(A, B) \bowtie S(B, E, C) \bowtie T(C, D)$. Suppose B has HHs b_1 and b_2, while C has HHs c_1, c_2, and c_3. Then there are 12 combinations, depending on whether B is b_1, b_2, or something else, and on whether C is c_1, c_2, c_3, or something else. R is then partitioned into three pieces, depending on whether B is b_1, b_2, or something else. T is partitioned into 4 pieces, and S is partitioned into 12 pieces, one for each of the 12 cases mentioned above (see more details in Examples 5 and 6).

So for each combination of choices, we will join the parts of each relation that agree with that choice. How we do this partitioning and how we distribute tuples to reducers accordingly is described in detail in Section 5.3.

Suppose we have 4 reducers. If the communication cost is 9, this means that 9 tuples are sent from the mappers to the reducers (e.g., tuple t_1 is sent to 4 reducers, tuple t_2 is sent to 2 reducers and tuple t_3 is sent to 3 reducers – this is total $4 + 2 + 3 = 9$ because tuple t_1 is sent in four copies, etc.). Thus the average number of tuples each reducer gets is 9/4.

4.2. Number of Reducers

For each combination of choices, solving by Lagrange multipliers, we get optimal shares as a function of k, the total number of reducers for this particular combination of choices. But we want to pick k so no reducer gets more than q tuples. We compute the number of tuples that are expected to wind up at a reducer by dividing the communication cost (given as a function of the number of reducers, k) by the number of reducers k. This way, we get k for each combination and get the shares for each attribute accordingly.

Once we do this for each combination of choices (i.e., for each residual join), and we get the proper k as a function of q for each, we can add those k's for each residual join to get the total number of reducers we need, as a function of q.

5. SharesSkew algorithm

This section contains the detailed description of our algorithm after we formally define residual joins. In the end of the section we have two simple examples and in Section 7, we give a more elaborate and complete example.

5.1. Definition of Residual Joins

As explained in Section 4, the SharesSkew algorithm defines a number of residual joins and then applies the Shares algorithm in each residual join seeing first that the attributes with HH in this residual join do not get any shares in the cost expression. Here we will show how we define each residual join. This is done independently of the data using only the knowledge about the HH values. Thus the overhead in this stage is independent of the size of the input. Each residual join takes as input a piece of the original data. These pieces may not be disjoint. Moreover, the complexity of the SharesSkew is that of the Shares algorithm multiplied by the number of residual joins. However, in practice these data pieces often do not have large overlaps so that each residual join handles an amount of data which is much smaller than the total amount.

In order to define residual joins we need first to define attribute types depending on how many HH an attribute has. If an attribute has p HH then it has $p + 1$ types. In particular, for each attribute X we define a set L_X of *types*:

1. If X has no Heavy Hitter values, then L_X comprises of only one type, $T_{\_}$, called the *ordinary type*.[3]
2. If X has p values that are Heavy Hitters, then L_X comprises of $p + 1$ types: one type T_b for each Heavy Hitter, b, of X, and one ordinary type $T_{\_}$.
   - A *combination of types*, C_T, is an element of the Cartesian product of the sets L_{X_i}, over all attributes $X_i, i = 1, 2, \ldots,$ and defines a *residual join*.

We say that a tuple of relation R is *relevant* to combination C_T if it satisfies the constraints of C_T. Given a combination of types C_T, another combination C'_T is *subsumed* by C_T if whenever C_T and C'_T disagree on a position (say one that corresponds to attribute B), the following are true; a) the type of B in C_T is ordinary, b) the type of B in C'_T is non-ordinary and, c) either B gets no shares in C_T or the following happens: for each relation R, let r be the relevant size with respect to C_T and b_h be the relevant size with respect to C'_T. Let b be the share of B with respect to C_T, then we require that b is less than r/b_h. We define the set of *maximal combinations* that are considered by the algorithm to be the maximal set such that no combination in the set is subsumed by another combination in the set. The test of subsumption may not be complete but we need to mention here this possibility for optimization. This means that it may not remove all subsumed residual joins.

Example 4. Suppose again we have the 2-way join $R(A, B) \bowtie S(B, C)$ but with all attributes A, B, C having one HH each, a, b, c respectively. In this case we can form 8 residual joins because each attribute has two types. But we still consider two residual joins the same as in Example 2, because the rest of the residual joins are subsumed by these two. I.e., we consider one residual join for type

[3] Ordinary type represents all other values of attribute X, the ones that are not Heavy Hitters.

combination $C_{T_1} = \{A : T_-, B : T_-, C : T_-\}$ (without HH) and one for type combination $C_{T_2} = \{A : T_-, B : T_b, C : T_-\}$ (with HH).

In more detail, suppose we have a HH for attribute A, which is $A = a$. Then it seems we need to take also the residual joins for $C_{T_3} = \{A : T_a, B : T_-, C : T_-\}$ and $C_{T_4} = \{A : T_a, B : T_b, C : T_-\}$. However C_{T_3} is subsumed by C_{T_1} and C_{T_4} is subsumed by C_{T_2} because in both cases, in either combination C_{T_1} or C_{T_2}, attribute A gets no shares. To gain more intuition, observe: a) In the first case the attribute A gets no shares (i.e., its share is equal to 1); here the tuples that contain the HH: $A = a$ have a different value on their B attribute hence they are distributed across several reducers and they do not create a problem. b) In the second case, there is only one tuple with $A = a$ in the relevant data i.e., the tuple (a, b); the other tuples $(a, -)$ or $(-, b)$do not participate in this residual join.

We want to show that $C_{T_2} = \{A : T_-, B : T_b, C : T_-\}$ may not always be subsumed by $C_{T_1} = \{A : T_-, B : T_-, C : T_-\}$. Suppose that we have 1000 tuples of relation $R(A, B)$ and 1000 tuples in relation $S(B, C)$ that do not have a HH in the B attribute; we also have 400 tuples in each relation with a HH value b. The number of relevant tuples in $C_{T_1} = \{A : T_-, B : T_-, C : T_-\}$ for each relation is equal to $r = 1000$. The number of relevant tuples in $C_{T_2} = \{A : T_-, B : T_b, C : T_-\}$ for each relation is equal to $b_h = 400$.

Then, share of B in C_{T_2} is equal to the number of reducers, that is $b = k$.

We take two cases:

1. Let $k = 2$. Now, since $r = 1000$ and $b_h = 400$, and $k = 2$, then $b_h < r/k = 1000/2 = 500$, hence C_{T_1} subsumes C_{T_2}.
2. Let $k = 10$. If $k = 10$, then $b_h > r/k = 1000/10 = 100$, meaning that C_{T_2} is not subsumed by C_{T_1}.

Each C_T defines a *residual join* which is the join computed only on a subset of the data. Specifically, if an attribute X has ordinary type in the current C_T we exclude the tuples for which $X = HH$.

If attribute X is of type T_b then we exclude (from all relations) the tuples with value $X \neq b$.

5.2. The dominance rule revisited

We now extend the notion of dominance in the context of the SharesSkew algorithm. We apply the dominance rule on each residual join but a with small modification: An attribute A is only dominated by an attribute B if B does not contain a HH.

5.3. Description of the SharesSkew Algorithm

We defined residual joins. Now we need to define how to hash on k reducers the relevant tuples for each residual join. As in the original Shares algorithm, we write the communication cost expression in terms of the share variables for the attributes and then we minimize this expression under the constraint that the product of the shares is equal to k. However, for each residual join we have a different cost expression.

The SharesSkew Algorithm works in 2 stages (namely, PreMap and Map):

Stage 1 First, from the original join, we decompose into the residual joins. Then, for each residual join J, we construct a set of keys by computing shares that minimize the cost expression for this J. The cost expression for each residual join is a version of the *generic* cost expression of the original join where we have omitted the share variables for attributes with HH – in other words each such share is equal to 1. Moreover, the size of each relation in the cost expression is now equal to the number of tuples that satisfy the constraints of the specific residual join.

Stage 2 During the map phase, we distribute tuples according to the set K_J of keys we constructed for each residual join J, specifically: A tuple t is hashed according to the set of keys K_J if t contains as HH values of attributes of J the value that actually defines J.

6. Pseudocode for the SharesSkew algorithm

Below, we give the pseudocode for the SharesSkew algorithm, both PreMap and Map Stages:

6.1. PreMap - Stage 1

The *PreMap* is executed before the main *Map stage*. Note that the PreMap stage, that is the calculation of shares for all residual joins, is performed only once before the MapReduce job takes place. The calculated shares can be computed either on one machine and then distributed to the mappers as parameters or once per mapper before the Map phase. In either case, this computation time is minuscule and has no impact on total execution time.

The input of the PreMap stage, *HH* is a list of sets, with each set containing (i) all of HHs of the i-th attribute and (ii) the number of tuples for every relation in which each HH value appears. For instance, attribute B may have a single HH value $b_1 = 5$ which occurs in $r_{b_1} = 1000$ tuples of relation R and $s_{b_1} = 500$ tuples of relation S.

First, the HeavyHitters information is mapped to the set of all possible types for each attribute, both ordinary (T_-) and HH (T_{b_1}), and their respective occurrences in relations. Next, we compute a cartesian product on all attribute types for every HH, thus forming all possible residual joins J. Finally, the Shares algorithm is applied, for all residual joins j_i. Hence, each residual join j_i, is associated with different shares that minimize the communication cost for j_i.

Details on how to apply the dominance rule on a residual join query schema are presented in Algorithm 1. We also give a detailed account of how the algorithm works in the case of the join query $R(A, B) \bowtie S(B, C, E) \bowtie T(C, D)$ in Appendix A.

Here we update twice an initialized (line 2) dominance matrix which will hold information about dominance between attributes. First, we identify when it is the case that two attributes are *both* part of at least one relation by looking through the schema (lines 3-10). After that, if there exists at least one more relation in which one attribute is present and the other is not, then the attributes form a (*dominating, dominated*) pair inside the dominance matrix (lines 11-18).

In lines 23-33, we translate the newly constructed dominance matrix into a list of *(dominating, dominated)* attribute pairs for the particular residual join j. In addition, since an attribute with a HH gets a share of 1 at all times, we place HHs along the dominated attributes inside the dominance matrix (lines 34-37).

6.2. Map - Stage 2

In the Map Step (Algorithm 2) we are given one tuple at a time and the, previously computed, shares for the set J of all residual joins. Then for each residual join j_i:

- We first check if this join is compatible with the input tuple in hand (line 3). To this end, we use Algorithm 3. Depending on the result, we continue on to construct the keys for the tuple replication.
- If the tuple is compatible with the residual join j_i, then for every attribute value, if it is a HH we hash the values according to the previously calculated shares. (line 4-7)
- If the attribute value is not a HH value then the tuple should be replicated as many times as the share for this attribute (lines 9-11)

Algorithm 1: Apply the Dominance Rule to a single residual join.

```
   input  : schemaBinaryMatrix, r: relationSizes, HH: HeavyHitters
   output: (dominated, dominating) attributes
 1 /* preMap.constructDominanceMatrix(schemaBinaryMatrix)          */
 2 /* initialize a matrix of −1s                                   */
 3 for elem in dominanceMatrix do elem ← -1;
 4 /* Scan schemaBinaryMatrix for co-appearance                    */
 5 foreach relation k in schemaBinaryMatrix do
 6 |   foreach pair(i,j) in dominanceMatrix do
 7 |   |   if schemaBinaryMatrix(k,i) == 1 & schemaBinaryMatrix(k,j) == 1 then
 8 |   |   |   dominanceMatrix(i,j) ← 0
 9 |   |   end
10 |   end
11 end
12 /* Scan dominanceMatrix for co-appearance and schemaMatrix for
      dominance                                                    */
13 foreach relation k in schemaBinaryMatrix do
14 |   foreach pair(i,j) in dominanceMatrix do
15 |   |   if schemaBinaryMatrix(k,i) == 1 & schemaBinaryMatrix(k,j) == 0 &
   |   |   dominanceMatrix(j,i) == 0 then
16 |   |   |   dominanceMatrix(i,j) ← 1
17 |   |   end
18 |   end
19 end
20 /* If a value is HH, set the appropriate column in
      dominanceMatrix to zero                                      */
21 foreach HeavyHitter h in HH do
22 |   dominanceMatrix(h, :) ← 0
23 end
24 /* Infer Dominating attributes from the Dominance Matrix        */
25 foreach cell(i, j) in dominanceMatrix do
26 |   if dominanceMatrix(i,j) == 1 & dominanceMatrix(j,i) == 0 then
27 |   |   // i dominates j
28 |   |   Add i-th attribute to dominatingAttributes ;
29 |   |   Add j-th attribute to dominatedAttributes ;
30 |   else if dominanceMatrix(i,j) == 0 & dominanceMatrix(j,i) == 1 then
31 |   |   // j dominates i
32 |   |   Add i-th attribute to dominatedAttributes ;
33 |   |   Add j-th attribute to dominatingAttributes ;
34 end
35 /* If a value is HH, then it gets a share of 1                  */
36 foreach HeavyHitter h in HH do
37 |   Add h-th attribute to dominatedAttributes ;
38 end
39 /* Construct (dominated, dominating) pairs of attributes        */
```

- Afterwards, we compute all the possible combinations of keys where a) for the HH dimensions we fix the dimensions with the hashed values and b) for the replicated key dimensions we span the key to the share size (lines 14-15)
  For example, for tuple $R(a = 5, b = 35)$ where $b = 35$ is a HH value with a share of 13, since $35 \ mod 13 = 9$, we would create keys (0, 9), (1, 9), (2, 9), (3, 9), (4, 9).
- Finally, in lines 19-21 the map task emits the set of key-value pairs for the particular residual join; this is what we denoted as K_J in the description of Stage 2 in Section 5.3.

Observe that since the HH attributes do not get shares, the following is true:

- Each tuple is hashed to reducers according to the values of the non-HH attributes in this tuple.
- If all attributes in the tuple are HHs then, this means that, in the current residual join, there is only this tuple that participates (i.e., is relevant) from its relation, and it is hashed to all reducers.

Comparison of Shares and SharesSkew algorithms In Shares the PreMap Step is trivial because we only consider one combination, the combination of all attributes having ordinary type. Then the Map Step only needs lines 13 through 16 to create keys for this single combination.

Algorithm 2: Map Step (Distribute input tuples to the reducers).

```
input  : t: tuple, HHs list, Shares_{j_i}: shares for residual join j_i
output: key-value pairs
for each residual join j_i do
    /* Decide if tuple is relevant to current residual join */
    if map.decideResidualJoin(tuple t, residual join j_i, HHs list) then
        for each attribute value val in t do
            if val is a HH then
                /* Hash in this dimension                    */
                replication = val mod Shares_{j_i}[val]
            end
            else
            end
            /* Replicate across this dimension               */
            replication = Shares_{j_i}[val]  /* Construct bucket
               addresses by fixing (hash) or spanning
               (replication) dimensions                      */
            for key in CartesianProduct(replication) do
                keylist.append(key)
            end
        end
    end
    foreach key in keylist do
        emit(< key; value >) ;
    end
end
```

6.3. SharesSkew Algorithm for 2-way Join

Essentially what we suggested in Example 2 can be summarized as follows. We decompose the 2-way join $R(A, B) \bowtie S(B, C)$ into two residual joins. They both compute the same query but on different data:

1. The first residual join computes the join on all tuples that do not contain the HH value. The Shares algorithm for this join is trivial, we do not have replication of the tuples; hence the communication cost is equal to the sum of sizes of the two relations (counting only the tuples without HH).
2. The second residual join computes the join on only the tuples that contain the HH value of B.

Let us see how the SharesSkew algorithm applies to this 2-way join. We first assume that all three attributes of the join are hashed according to their values. Then we write the generic communication cost expression which is $ryz + sxz$. For the second residual join, we put the share variable for attribute B (here this is z) equal to 1. Hence we need to minimize $ry + sx$ under the constraint that $xy = k$. This is what we did intuitively in Example 2.

6.4. Example: Combining Heavy Hitters in SharesSkew algorithm

The following example (Example 5) explains what we do if a single attribute has more than one Heavy Hitter. Furthermore, we devote the next section (Section 7) to presenting an elaborate example when there are several attributes with Heavy Hitters (Example 6).

Example 5. Suppose that in the 2-way join $J = R(A, B) \bowtie S(B, C)$ we have two Heavy Hitters for attribute B (say b_1 and b_2). Then we have three residual joins: one without HHs, one with only the tuples with $B = b_1$ and a third one with only the tuples with $B = b_2$.

7. Applying the SharesSkew algorithm on an example

In this section we include an elaborate and detailed example for SharesSkew. Example 6 below deals with the calculation of shares for each residual join (presented in Fig. 2) and Example 7 continues on to showcase how input tuples are distributed to the reducers for every residual join (presented at in Fig. 3).

Example 6. We want to compute the 3-way join: $J = R(A, B) \bowtie S(B, E, C) \bowtie T(C, D)$.

Suppose attribute B has two HHs, $B = b_1$ and $B = b_2$ and attribute C has one HH, $C = c_1$. Thus attribute B has three types, T_{-}, T_{b_1} and T_{b_2}, attribute C has two types, T_{-} and T_{c_1} and the rest of the attributes have a single type, T_{-}. Thus we have $3 \times 2 = 6$ residual joins, one for each combination. By r, s, t we denote the sizes of the relations that are *relevant* in each residual join, i.e., the number of tuples from each relation that contribute in the particular residual join. We list the residual joins J_1 to J_6 along with some data tuples to illustrate how SharesSkew works. In our data, we assume that the HH are: $b_1 = 3, b_2 = 9$ and $c_1 = 30$.

J_1: All attributes of type T_{-}. Here r is the number of only those tuples of relation R for which $B \neq b_1$ and $B \neq b_2$, s is the number of only those tuples of relation S for which $B \neq b_1$ and $B \neq b_2$ and $C \neq c_1$, and t is the number of those tuples in relation T for which $C \neq c_1$. Here we have tuples $R(10, 1)$, $R(11, 2)$ and $S(2, 21, 31)$, and $T(31, 41)$, $T(32, 42)$, $T(33, 43)$. Thus $r = 2, s = 1$ and $t = 3$.

J_2: All attributes of type T_{-}, except B of type T_{b_1}. In this case r is the number of only those tuples in relation R for which $B = b_1$, s is the number of only those tuples in relation S for which $B = b_1$ and $C \neq c_1$, and t is the number of those tuples in relation T for which $C \neq c_1$. Here we have the tuples: $R(10, 3)$, $R(12, 3)$, $R(14, 3)$, $R(15, 3)$, and, $S(3, 26, 31)$, $S(3, 26, 32)$,

Algorithm 3: Decide Residual Join.

input : t: tuple, j_i: residual join, HHs: Heavy Hitters
output: True or False: Whether tuple t should be sent to j_i

```
1  /* decisionFlag is the return value                                  */
2  /* initialize                                                        */
3  decisionFlag = True
4  for tupleValue in t, residualValue in j_i do
5      /* If flag is still active                                       */
6      if residualValue is not HH then
7          if tupleValue is HH then
8              /* residualValue is not a HH, but tupleValue is HH     */
9              decisionFlag=False
10             return decisionFlag
11         end
12     end
13     else
14         if tupleValue==residualValue then
15             decisionFlag=True
16             continue
17         end
18         else
19             if tupleValue is HH then
20                 /* residualValue and tupleValue are HH, but
                      tupleValue does not match the residualValue     */
21                 decisionFlag=False
22                 return decisionFlag
23             end
24         end
25     end
26 end
27 return decisionFlag
```

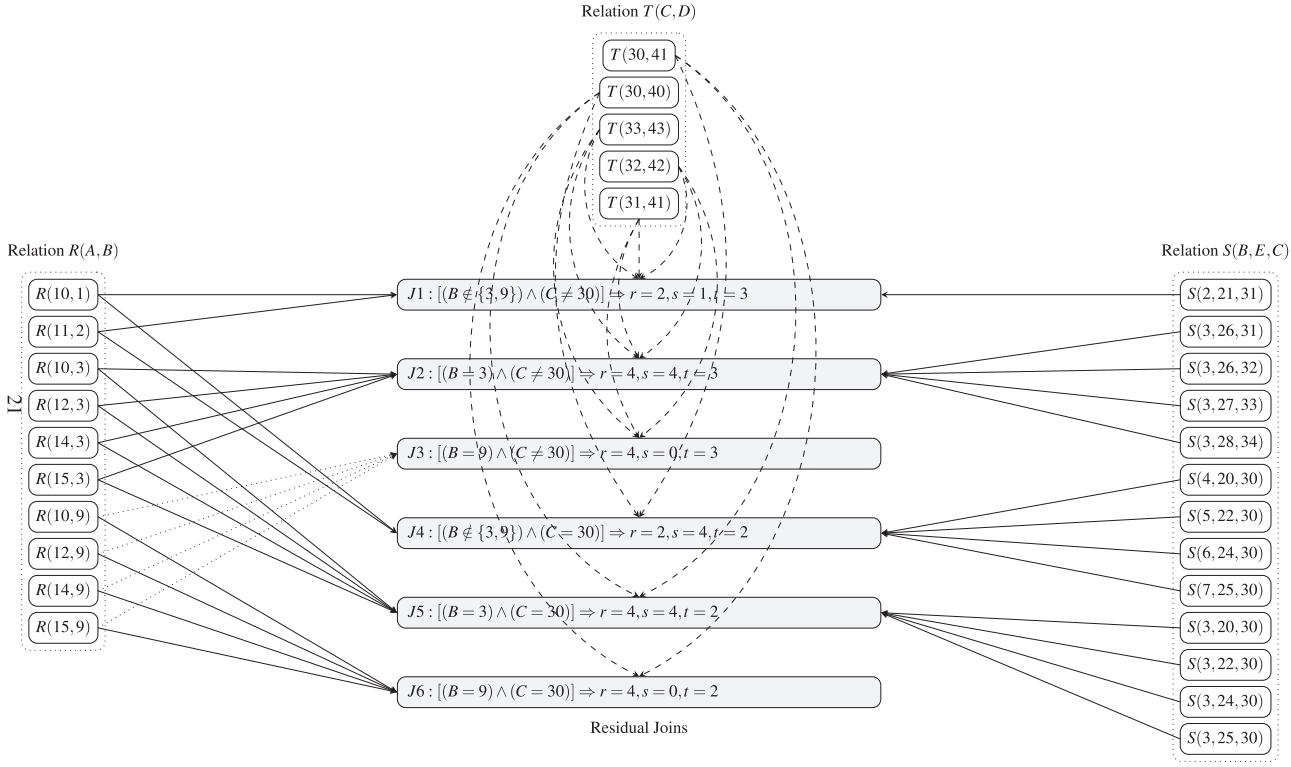

Fig. 2. (Example 6) Calculating the shares for the residual joins J_1 to J_6.

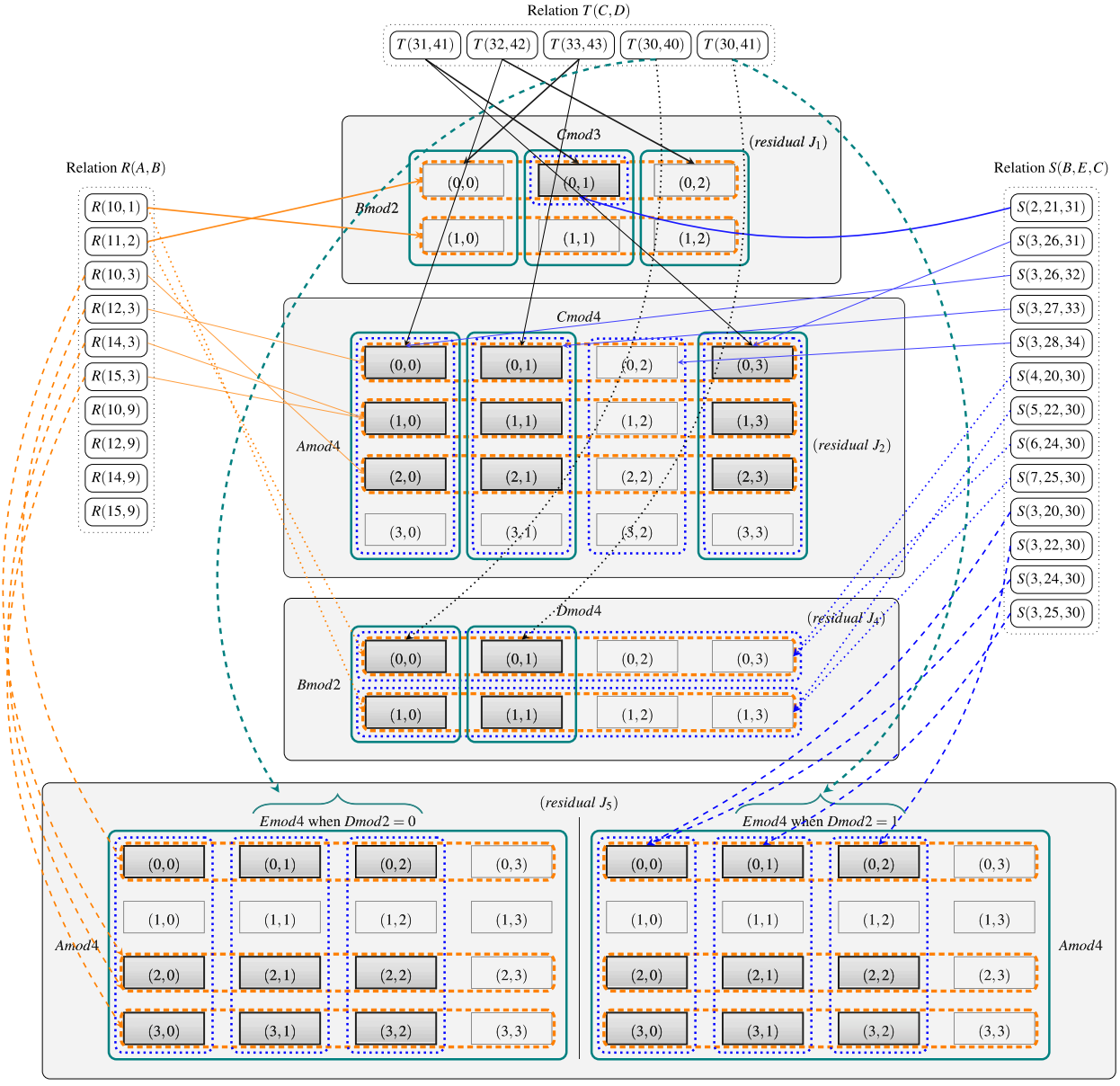

Fig. 3. (Examples 6–8) Distribution of tuples to the reducers for the residual joins J_1, J_2, J_4, J_5.

S(3, 27, 33), S(3, 28, 34), and T(31, 41), T(32, 42), T(33, 43). Thus $r = 4, s = 4$ and $t = 3$.

J_3: All attributes of type T_-, except B of type T_{b_2}. The analysis is almost the same as the case above with the only difference that we have a different HH for B, hence different sizes for relations R and S. Here we have the tuples: R(10, 9), R(12, 9), R(14, 9), R(15, 9) and T(31, 41), T(32, 42), T(33, 43) while S is empty. Thus $r = 4, s = 0$ and $t = 3$.

J_4: All attributes of type T_-, except C of type T_{c_1}. The treatment is similar as in (2) above for B. Here we have tuples R(10, 1), R(11, 2), and, T(30, 40), T(30, 41), and S(4, 20, 30), S(5, 22, 30), S(6, 24, 30), S(7, 25, 30). Thus $r = 2, s = 4$ and $t = 2$.

J_5: All attributes of type T_-, except B of type T_{b_1} and C of type T_{c_1}. In this case r is the number of only those tuples of relation R for which $B = b_1$, s is the number of only those tuples of relation S for which $B = b_1$ and $C = c_1$, and t is the number of those tuples in relation T for which $C = c_1$. Here we have the tuples: R(10, 3), R(12, 3), R(14, 3), R(15, 3), and, S(3, 20, 30), S(3, 22, 30), S(3, 24, 30), S(3, 25, 30), and T(30, 40), T(30, 41). Thus $r = 4, s = 4$ and $t = 2$.

J_6: All attributes of type T_-, except B of type T_{b_2} and C of type T_{c_1}. The analysis is analogous to the case (5) above. Here we have the tuples: R(10, 9), R(12,9), R(14, 9), R(15, 9), S is empty, and T(30, 40), T(30, 41). Thus $r = 4, s = 0$ and $t = 2$.

Each residual join is treated by the Shares algorithm as a separate join and a set of keys is defined that will be used to hash each tuple as follows: A tuple t of relation R_j is sent to reducers of combination C_T only if the values of the tuple satisfy the constraints of C_T as concerns values of HH. We give an example:

Example 7. We continue from Example 6. Each tuple is sent to a number of reducers according to the keys created for each residual join. E.g., a tuple t from relation R is sent to reducers as follows:

1. If t has $B = b_1 = 3$ then it is sent to reducers created in items (2) and (5) in Example 6. Indeed we have listed the following tuples in these two items above: R(10, 3), R(12, 3), R(14, 3), R(15, 3)

2. If t has $B \neq b_1$ and $B \neq b_2$ then it is sent to reducers created in items (1) and (4) in Example 6. Indeed we have listed the following tuples in these two items above: $R(10, 1)$, $R(11, 2)$.
3. If t has $B = b_2 = 9$ then it is sent to reducers created in items (3) and (6) in Example 6. Indeed we have listed the following tuples in these two items above: $R(10, 9)$, $R(12, 9)$, $R(14, 9)$, $R(15, 9)$.

Thus we have covered all the tuples in relation R as to which reducers each should be sent.

In Example 6 we showed how to construct the residual joins and in Example 7 we showed how to distribute tuples. Now we are going to show how to write the cost expression for each residual join and compute the shares.

Example 8. We continue from Example 6 for the same HH as there. Remember by *a, b, c, d, e* we denote the shares for each attribute *A, B, C, D, E* respectively and by *r, s, t* we denote the sizes of the relations that are *relevant* in each residual join, i.e., the number of tuples from each relation that contribute in the particular residual join. We always start with the generic cost expression for the original join, $rcde + sad + tabe$, and then simplify accordingly. We list the cost expression for every residual join (and in the same order as) in Example 6. We will proceed with assigning reducers to each residual join J_1 to J_6, so we assume that $q = 3$, i.e., we want each reducer to get at most three tuples.

J_1: Here all attributes are ordinary, so we simplify the relation by observing that A is dominated by B, D is dominated by C and E is dominated by C, hence $a = 1$, $d = 1$ and $e = 1$ and the expression is:
$rc + s + tb$. Since we have $r = 2, s = 1$ and $t = 3$, the cost expression is $2c + 1 + 3b$. Here we set up 6 reducers call them (i, j), $i = 1, 2$, $j = 31, 32, 33$ (the hashing is done according to the values in attributes B, D for simplicity). As described in 4.2 we solve the Langragean that minimizes the per reducer cost $(2c + 1 + 3b)/k$ s.t. $bc = k$ for k i.e. the number of reducers. Thus, $k = 6 = bc$ and $b = 2$ and $c = 3$ from all combinations minimize the cost expression. The two tuples of relation R go to 3 reducers each as follows: tuple $R(10, 1)$ goes to reducers (1,31), (1,32), (1,33) and tuple $R(11, 2)$ goes to reducers (2,31), (2,32), (2,33). The single tuple of relation S goes to one reducer the (2,31). The three tuples of relation T go to two reducers each as follows: $T(31, 41)$ to (1,31) and (2,31), $T(32, 42)$ to (1,32) and (2,32), $T(33, 43)$ to (1,33) and (2,33). This means $c = 3$, $b = 2$ are the shares for attributes C and B. The communication cost is $2 \times 3 + 1 + 3 \times 2 = 6 + 1 + 6 = 13$. The occurrence of 6 twice is no coincidence because the Lagrangean method finds the minimum which occurs when the terms in the cost expression become equal (the term 1 does not count since it is not involved in the optimization procedure, it is a constant).

J_2: Here only B is a non-ordinary attribute, hence $b = 1$ and then, from the remaining attributes D and E are dominated by C, hence $d = 1$ and $e = 1$, and the expression is: $rc + sa + ta$. Since we have $r = 4, s = 4$ and $t = 3$, the cost expression is $4c + 4a + 3a$. Here we set up 16 reducers arranged in 4×4, i.e., four buckets for the values of attribute C and four buckets for the values of attribute A. Each tuple in R goes to four reducers according to the value in its A attribute. Each tuple in S and each tuple in T goes to four reducers according to the value in their C attribute. Here the communication cost is $4 \times 4 + 4 \times 7 = 44$.

J_3: All attributes are of type T_-, except attribute B which is of type T_{b_2}. The analysis almost same as the case above with the only difference that we have a different HH for B, hence different sizes for relevant relations. Thus, the expression is $rc + sa + ta$, i.e., same as above, only the sizes of the relations will be different. Since we have $r = 4, s = 0$ and $t = 3$, the cost expression is $4c + 0 + 3a$. Here we know we do not have any data in relation S to form a join, so we do not set up any reducers.

J_4: All attributes are of type T_-, except attribute C which is of type T_{c_1}. Hence $c = 1$. From the rest of the attributes, A and E are dominated by B. Thus, the expression is $rd + sd + tb$. Since we have $r = 2, s = 4$ and $t = 2$, the cost expression is $2d + 4d + 2b$. We set up 8 reducers arranged in 2×4 for attributes B and D.

J_5: Here we set both $b = 1$ and $c = 1$ and this gives us $rde + sad + tae$.
Since we have $r = 4, s = 4$ and $t = 2$, the cost expression is $4de + 4ad + 2ae$. We set up 32 reducers arranged in $4 \times 4 \times 2$ buckets for the values of attributes A, D, E. That is the shares are $a = 4, d = 2, e = 4$.

J_6: The expression is $rde + sad + tae$, i.e., same as above, only the sizes of the relations will be different. Since we have $r = 4, s = 4$ and $t = 2$, the cost expression is $4de + 4ad + 2ae$ (it is the same as in the item just above, but it is only a coincidence). The shares are the same as above.

8. SharesSkew on important joins

In this section we derive closed forms for the attribute shares and the communication cost for chain joins for the SharesSkew algorithm. The closed forms can be also used to demonstrate how the algorithm is robust in different levels of skewness by observing that the optimal communication cost that is calculated is a continuous function of the skewness levels (i.e., the number of skewed tuples in each relation). Then, in Section 8.3 we give an answer to the question: Does SharesSkew always perform significantly better than Shares in the presence of HHs? We introduce a family of joins (symmetric joins) and argue that a subfamily of it has the property to defy HHs even for Shares. Actually, we show something even stronger which is that the communication cost of Shares for this subfamily is almost optimal. On this last point, it is worth noting that there is already formal (probabilistic) evidence [2] that the Shares algorithm behaves almost optimally for certain joins even in the presence of skew.

8.1. Chain Joins: all relations of equal size

We begin with chain joins which are joins of the form:

$$R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie \ldots \bowtie R_n(A_{n-1}, A_n)$$

We assume all relations in a specific residual join are of equal size r. When some attributes are HH in a specific residual join, then the cost expression as analyzed in [6] changes because (as we have already explained) the HH attributes take a share equal to 1. For example, when the chain join is $R(A, B) \bowtie S(B, C) \bowtie T(C, D) \bowtie U(D, E)$ the cost expression is $rcd + sd + tb + ubc$.

However, we make an observation here that leads to a trick that allows us to get closed forms for the shares even for this (different) cost expression.

Suppose attribute A_j is one that contains a HH with value equal to a_j. Consider the type combination C_T that includes type $A_j : T_{A_j}$. We say that attribute A_j appears in C_T with a HH. Now, we want to work on the cost expression of type combination C_T where at least one of the attributes appears with a HH. We make the observation that each attribute in the chain that appears in C_T with a HH may be seen as dividing the chain in two parts (if there is only one such attribute in C_T) because in the cost expression the share of the attribute that appears in C_T with a HH is equal to 1; so this expression can be viewed as the sum of two expressions each

for one subchain. We can thus find the shares that minimize each subexpression (by solving with the Lagrangean method as is done in detail in Section 4.4 of [6]) given that it uses k_i, $i = 1, 2$ reducers. Now we need to minimize taking into account that $k_1 k_2 = k$.

In the general case, where we assume we have m subchains (because we have $m-1$ HHs) the constraint is $\Pi_1^m k_i = k$. Each subchain i corresponds to a subexpression in the expression of the communication cost. We can view each subchain as a separate chain and again use the results from [6] to compute the minimum of the corresponding subexpression i for specific k_i. This is the following:

$$rn_i k_i^{(n_i-2)/n_i}$$

where r is the size of each relation and n_i is the number of relations in the i-th subchain. Now we want to minimize the:

$$\sum_i rn_i k_i^{(n_i-2)/n_i}$$

under the constraint $\Pi_1^m k_i = k$. By taking the Lagrangean method, we find that

$$(n_1-2)k_1^{(n_1-2)/n_1} = (n_i-2)k_i^{(n_i-2)/n_i}$$

Hence,

$$k_i = \left(\frac{n_1-2}{n_i-2}\right)^{n_i/(n_i-2)} k_1^{n_i(n_1-2)/n_1(n_i-2)}$$

Hence, by multiplying all, and since $k_1 k_2 \ldots k_m = k$, we can get the k_1 and from it all the k_i's.

These calculations hold when all subchains have even length. For odd length, the case is similar but a little more tedious.

8.2. Chain Joins: Arbitrary relation sizes

First we compute the optimal communication cost for chain joins without HH (it is not included in [6]). In this case, the cost expression from [6] is shown to be a sum of terms where $\tau_1 = r_1 k/a_1$, $\tau_n = r_n k/a_{n-1}$ and for all other τ's we have $\tau_i = r_i k/a_{i-1}a_i$, where a_i is the share for attribute A_i. Solving the Lagrangean obtains that all odd terms are equal to each other and the same goes for all even terms. In particular, the equalities for the even-n case have the form

$$\frac{r_1}{a_1} = \frac{r_3}{a_2 a_3} = \frac{r_5}{a_4 a_5} = \cdots = \frac{r_{n-1}}{a_{n-2}a_{n-1}}$$
$$\frac{r_2}{a_1 a_2} = \frac{r_4}{a_3 a_4} = \cdots = \frac{r_{n-2}}{a_{n-3}a_{n-2}} = \frac{r_n}{a_{n-1}}$$

Thus, setting $\tau_i = r_i k/a_{i-1}a_i = \lambda_1$ for odd terms and $\tau_i = r_i k/a_{i-1}a_i = \lambda_2$ for even terms, and multiplying all odd terms to get $\lambda_1^{n/2}$ and all even terms to get $\lambda_2^{n/2}$ we observe that: the denominator of the left hand side (after this multiplication) is the product of all a_i's, hence it is equal to k. Thus we get that

$$\lambda_1 = k^{1-2/n}(r_1 r_3 r_5 \ldots)^{2/n}$$

$$\lambda_2 = k^{1-2/n}(r_2 r_4 r_6 \ldots)^{2/n}$$

Thus the communication cost is:

$$\text{cost} = n/2(\lambda_1 + \lambda_2) =$$

$$n/2 \times k^{(n-2)/n}((r_1 r_3 r_5 \ldots)^{2/n} + (r_2 r_4 r_6 \ldots)^{2/n})$$

Now, we start calculating the shares and communication cost of the SharesSkew algorithm by following the same lines as we did in Section 8.1. Thus, for finding the optimal shares for chains with HH, we do exactly the same calculations as in Section 8.1 with the difference that: instead of multiplying the $k_i^{(n_i-2)/n_i}$ by rn_i in the cost expression of each subchain i we multiply it by

$$n_i/2 \times ((r_1 r_3 r_5 \ldots)^{2/n_i} + (r_2 r_4 r_6 \ldots)^{2/n_i})$$

where the $r_1, r_2, \ldots$ refer to the sizes of the relation for subchain i. Thus we can conveniently put

$$\hat{r}_i = 1/2 \times ((r_1 r_3 r_5 \ldots)^{2/n_i} + (r_2 r_4 r_6 \ldots)^{2/n_i})$$

and thus, having to minimize the following cost expression:

$$\sum_i \hat{r}_i n_i k_i^{(n_i-2)/n_i}$$

By taking the Lagrangean method, now we find that

$$\hat{r}_1(n_1-2)k_1^{(n_1-2)/n_1} = \hat{r}_i(n_i-2)k_i^{(n_i-2)/n_i}$$

Then we proceed in similar way again to find that the optimal communication cost is the sum over all subexpressions i of:

$$\sum \hat{r}_i n_i k_i^{(n_i-2)/n_i}$$

where

$$k_i = \left(\frac{\hat{r}_1(n_1-2)}{\hat{r}_i(n_i-2)}\right)^{n_i/(n_i-2)} k_1^{n_i(n_1-2)/n_1(n_i-2)}$$

Now from these calculations, the robustness of the algorithm for different levels of skewness can been seen since the communication cost depends on the number of skewed tuples (these are the corner r_j's of the subexpressions) in a continuous way – i.e., without having a breaking point where it may jump. This was expected from the SharesSkew algorithm which distributes tuples with skew in an "even" manner and hence it has the same behavior for any multiway join.

Table 1
Adjacency matrix for a symmetric join with $d = 2$.

	R_1	R_2	R_3	R_4	R_5	...	R_m
A_1	1	1	1	0	0	0	0
A_2	0	1	1	1	0	0	0
A_3	0	0	1	1	1	0	0
A_4	0	0	0	1	1	...	0
A_5	0	0	0	0	1	...	0
...			...			...	...
A_{m+2}	1	1	0	0	0	...	1

8.3. Symmetric joins

We define *symmetric* joins to be the joins with an adjacency matrix (see Table 1) which is as follows: a) in the i-th row, the i-th entry through $i+d$-th entry (for a given d for this join) is 1 and all other entries have 0 (where m is the length of a row and $i+d$ is computed mod-m) and b) contains $m+d$ rows where m is the number of columns.

Thus properties of symmetric joins include:

- All relations have the same arity, d, (i.e. every column of the matrix has d ones).
- Each attribute appears in exactly d relations (i.e. every row of the matrix has d ones).

Our goal is to give a closed-form expression for the communication cost for symmetric joins.

We shall first analyze the case where all relations are of equal size. That case involves considerably simpler algebraic expressions, yet serves to introduce the calculations used in the general case, without obscuring the idea behind the algebra.

With techniques very similar to chain joins, as in [6], it is easy to get the Lagrangean equations for this case:

$$\tau_{d+1} + \tau_{d+2} + \cdots + \tau_n = \lambda k$$
$$\tau_1 + \tau_{d+2} + \cdots + \tau_n = \lambda k$$
$$\tau_1 + \tau_2 + \tau_{d+3} + \cdots + \tau_n = \lambda k$$
$$\tau_1 + \tau_2 + \tau_3 + \tau_{d+4} + \cdots + \tau_n = \lambda k$$
$$\ldots$$

Where $\tau_1 = r_1 k/a_1$, $\tau_n = r_n k/a_{n-1}$ and for all other τ's we have $\tau_i = r_i k/(a_{i-1}a_i)$, with r_i being the size of relation R_i and a_i the share of attribute A_i. The fact that k is the product of the a_i's justifies this rewriting. These equations imply the following d groups of equalities:

$$\tau_1 = \tau_{d+1} = \tau_{2d+1} = \cdots$$
$$\tau_2 = \tau_{d+2} = \tau_{2d+2} = \cdots$$
$$\tau_3 = \tau_{d+3} = \tau_{2d+3} = \cdots$$
$$\ldots$$
$$\tau_d = \tau_{2d} = \tau_{3d} = \cdots$$

We shall henceforth look only for values of the share variables (and give closed form solution to the communication cost) that satisfy the d groups of equalities. We take into account that τ_i/τ_{i+1} is equal to a_{i+d-1}/a_{i-1}. Since we can calculate the τ's, we can derive a simple linear system of equations which we can solve to find the a's. After we do the math we get the theorem:

Theorem 1. *For each join in the class of symmetric joins with n relations and d attributes in each relation, the Shares algorithm has communication cost equal to*

$$n_d k^{1-\frac{d}{n}} \sum_{all\ S} \left(\prod_{r_i \in S} r_i \right)^{1/n_d}$$

where n_d is the smallest integer such that n divides dn_d, and each S is the subset of relations $\{R_j, R_{j+d}, R_{j+2d}, \ldots, R_{j+dn_d}\}$, $j = 1, 2, \ldots$.

The communication cost stated in the above theorem is very simple when all relations have the same size r:

$$nrk^{1-\frac{d}{n}}$$

Remember that nr is the size of the input. Thus the minimum possible communication cost is nr. In the above expression this is multiplied by $k^{1-\frac{d}{n}}$. This means that, if d is close to n then this cost is almost optimal, e.g., for $d = n - 3$ this cost is proportional to $k^{\frac{3}{n}}$ which is close to 1 for large n. For such symmetrical joins, using the SharesSkew instead of the Shares algorithm (remember the above communication cost is calculated for the Shares algorithm) will not result in a significant improvement of performance. It is worth noting that chain joins are among those joins where the Shares algorithm has high communication cost – since it is proportional to $k^{\frac{n-2}{n}}$. Thus, multiround MapReduce algorithms are worth investigating in this case.

8.4. Multi rounds of MapReduce

After the above analysis, it does not come as a surprise that two rounds (or more) of MapReduce can have significantly lower communication cost than one round. Imagine the following scenario: There are two parts in the multiway join and they loosely joined with one another (meaning that they share very few attributes, say only one attribute is shared): a) a part with HH and also symmetric which falls in the subclass of previous section with resiliency to skew and b) another part such as a chain join without HH. Now, suppose that the communication cost of the chain join (this is the second part of our multiway join here) is high compared to the the cost of the first part of our multiway join (this is possible as we explained in the previous section). If we use one round, then we will have to distribute, for each residual join, the same tuples that are input to the relations of the second part of our join.

However, if we join the two parts separately first and then have another round for joining them together, we may have the following benefits: a) the first part will run with low communication cost by using Shares algorithm, b) the final join of the outputs of the two parts can be done with minimum communication (assuming that the sizes of the intermediate relations are not high), thus c) the important overhead to the communication cost will come only from the second part of the join (the chain subjoin), which does not have any HH and thus we can save here by using Shares algorithm and not SharesSkew.

8.5. Discussion

The SharesSkew algorithm performs well in high levels of skewness. Imagine the 2-way join where the join attribute B has a value that appears in half of the tuples whereas the rest of its values appear, each, in a small number of tuples. Then, the residual join that will handle the HH value will distribute the tuples as if we had a uniform distribution because it hashes only on the values of the non-join attributes. Note also, that a high level of skewness may appear when we have nulls. On the other end, if the data has a heavy tail, that is if there are many different values that appear less frequently in the dataset, we can modify the algorithm so that: the heavy tail is viewed as random data whereas data around the peak of the normal (or any similar) distribution are treated as HH. In order to do that, one efficient method is to group values together and declare HH groups of values. This will keep the number of residual joins low while taking advantage of the good performance of the SharesSkew algorithm for each residual join.

9. Efficiency of SharesSkew

In this section, we provide an additional explanation about why the SharesSkew algorithm is expected to perform in data with skew as efficiently as the Shares algorithm performs in similar data without skew. In SharesSkew, for each residual join, we designated arbitrarily no shares to the HH attributes (i.e., their share equals 1) and then run Shares algorithm to compute the remaining shares. In this section we will show that by doing this, it is as if we run the Shares algorithm on random (non-skewed) data.[4]

We first begin with the 2-way join and then we explain for any multiway join.

9.1. The 2-way join

To see that the method we presented in Example 2 is actually based on the Shares algorithm (and to be able to extend it for more than one HH), we think as follows:

We take Example 2. Imagine we had the following dataset instead of the one given in the example with the HH for attribute B: We replace each tuple of relation R (remember, in our example, for all tuples we had the value for attribute $B = b$) with a tuple where B has distinct fresh values $b_1, b_2, \ldots$ and the same for the tuples of relation S with B having distinct fresh values $b'_1, b'_2, \ldots$. This dataset has no HH, so we can apply the Shares algorithm to find the shares and distribute the tuples to reducers normally. The only problem with this plan is that the output will be empty because we have chosen the b_is and b'_is to be all distinct. This problem however has an easy solution: after we have done the distribution of the tuples

[4] Remember that all the explanations in this section are on the conceptual level, i.e., we do not suggest that we need to materialize anything that is presented in this section or change anything in SharesSkew algorithm which we explained in the previous sections.

properly (and efficiently) we turn back, in the reducer side, all b_is and b'_is to b. Thus we can compute the join correctly and produce the desired output. Thus we keep this replacement (of the b_is and b'_is) at the conceptual level, in order to create a HH-free join and be able to apply the Shares algorithm and compute the shares optimally, however when we transfer the tuples to the reducers, we transfer the original tuples. We formalize this thought for the 2-way join in the next paragraph (Section 9.2 extends for any multiway join).

First we introduce two *auxiliary attributes* B_R and B_S and an *auxiliary relation* $R_{aux}(B_R, B_S)$.[5] Now we imagine that we have the join $R(A, B_R)\bowtie R_{aux}(B_R, B_S)\bowtie S(B_S, C)$ to compute on three relations, where relation $R(A, B_R)$ contains all tuples of relation $R(A, B)$ with B being the HH having replaced the HH value with a fresh value, different for each tuple (say set B_1 consists of all these values). Similarly, for each tuple of relation $S(B, C)$ with B being the HH we have replaced the HH value with a fresh value, different for each tuple (say set B_2 consists of all these values – B_1 and B_2 are disjoint). Also let relation R_{aux} consist of the Cartesian product of B_1 and B_2 and tuples of the form $(b'b')$ for every b' in B that is not a HH. The 3-way join applied on this data is computed in an almost isomorphic way as the 2-way on the tuples with HH, i.e., the same number of pairs of tuples from R and S are joining and we have the same size of result. However, the 3-way join is a join without skew, so we are free to use the original Shares algorithm. Then the communication cost expression is $rcb_s + r_{aux}ac + sab_r$ (where c, b_s, a and b_r are the shares for attributes C, B_S, A and B_R respectively). Here the auxiliary relation is not actually transferred; hence we can drop from the cost expression the middle term. Since we dropped this term, it is as if we have a cost expression for the join $R(A, B_R)\bowtie S(B_S, C)$.

Now, according to the dominance relation for the 2-way join $R(A, B_R)\bowtie S(B_S, C)$, we have a choice: either to choose that A dominates B_R or B_R dominates A. We choose the former, hence $b_r = 1$. Similarly we take $b_s = 1$. Hence the cost expression now is $rc + sa$. Thus we have arrived at the same algorithm to compute 2-way join with skew as the one we developed intuitively in Example 2 and with the same communication cost.

9.2. Any multiway join

The conceptual structure in the general case is as follows: For each combination of types, C_T (that corresponds to residual join J), we define a HH-free residual join J' (it will be HH-free by construction of the dataset on which it is applied):

1. If attribute X has non-ordinary type in C_T then:
   –We introduce a number of auxiliary attributes, one auxiliary attribute per each relation R_j where attribute X appears. We denote the auxiliary attribute for relation R_j by X_{R_j}.
   –In the schema of each relation R_j where X appears, we replace X with attribute X_{R_j}.
2. We form the residual join J' for C_T by adding to the original join new relations as follows: one relation, R^X_{aux}, for each attribute X which is not of ordinary type. The schema of that relation consists of the attributes X_{R_j} for each j such that X is an attribute of R_j.

Now we apply this modified join J' on the following database $\mathcal{D}'$ that is constructed from the given database $\mathcal{D}$ as follows:

1. For each HH (in the current residual join) in a tuple i of relation S we do: Suppose value a is a Heavy Hitter for attribute X; then we replace a with $a.i.S$ in tuple i. We denote the set of all $a.i.S$'s by A_S.[6]
2. We form each auxiliary relation, R^X_{aux}, by populating it with a) the cartesian product of sets A_S for attribute X, one set for each relation where a is a HH in the current residual join and b) with the tuples t, for every b' in X that is not a HH in X, where t's entries are all equal to b'.
   - *Observation 1:* Database $\mathcal{D}'$ now has no Heavy Hitters.

Example 9. Thus if, in database $\mathcal{D}$, relation $R(A, B)$ is {(1, 2), (3, 2), (4, 2)} and relation $S(B, C)$ is {(2, 5), (2, 6)} then, in database $\mathcal{D}'$ we have (assuming $B = 2$ qualifies for HH): $R(A, B_R)$ is {(1, 2.1.R), (3, 2.3.R), (4, 2.4.R)}.

$S(B_S, C)$ is {(2.5.S, 5), (2.6.S, 6)}.

(I.e., we conveniently identify the tuple of R with the value of its first argument and the tuple of S with the value of its second argument.)

The auxiliary relation $R_{aux}(B_R, B_S)$ is : {(2.1.R, 2.5.S), (2.3.R, 2.5.S), (2.4.R, 2.5.S), (2.1.R, 2.6.S), (2.3.R, 2.6.S), (2.4.R, 2.6.S)}

We give below a more complicated example with two HHs

Example 10. Let us assume now that we have the relations $R(A, B)$ and $S(B, C)$ as in Example 9 and also we have relations $T(B, C, D)$ and $T'(D, E)$. Let us take database $\mathcal{D}$ that contains the same tuples in R and S as in Example 9. Relation T is {(2, 4, 5), (2, 6, 5), (2, 7, 5)} and relation T' is {(5, 1), (5, 2), (5, 3)}. Suppose $D = 5$ qualifies for HH. Then database $\mathcal{D}'$ is the same for relations R and S as in Example 9 and for relation T is

$$\{(2.4.T, 4, 5.4.T), (2.6.T, 6, 5.6.T), (2.7.T, 7, 5.7.T)\}$$

and for relation T' is

$$\{(5.1.T', 1), (5.2.T', 2), (5.3.T', 3)\}$$

Now we do not have the auxiliary relation $R_{aux}(B_R, B_S)$ because the HH for B appears in three relations, thus we should have instead $R_{aux}(B_R, B_S, B_T)$ which now has $3 \times 2 \times 3 = 18$ tuples, so we list only a few of these tuples, e.g.,

{(2.1.R, 2.5.S, 2.4.T), (2.3.R, 2.5.S, 2.4.T), (2.4.R, 2.5.S, 2.4.T), (2.1.R, 2.5.S, 2.6.T), (2.3.R, 2.5.S, 2.6.T), (2.4.R, 2.5.S, 2.6.T)}. And finally we have another auxiliary relation because of HH in attribute D. This is $R'_{aux}(D_T, D_{T'})$ and some of its tuples are {(2.7.T, 5.1.T'), (2.7.T, 5.2.T')} – in total, it has 9 tuples.

- *Observation 2:* There is a tuple $i = (a_1, a_2, \ldots, d_1, d_2, \ldots)$ in relation S in D with $d_1, d_2, \ldots$ being the HH iff there is a *corresponding* tuple $i = (a_1, a_2, \ldots, d_1.i.S, d_2.i.S, \ldots)$ in D'. Hence, in the presence of the auxiliary relations, a tuple is in the output of the residual join J applied on dataset $\mathcal{D}$ iff the corresponding tuple is in the output of join J' applied on dataset $\mathcal{D}'$.

Simplifying the Cost Expression

What is the cost expression, according to Shares, of join J' when applied on dataset $\mathcal{D}'$? First we observe that the property of the dominance relation allows us to write the cost expression for each residual join in a simple manner. We use the theorem:

Theorem 2. *The share of each auxiliary attribute is equal to 1 in the optimum solution.*

Proof. Each auxiliary attribute appears in one relation of the original join and in one auxiliary relation. Since we do not add a term in the cost expression for the auxiliary relation, we imagine that we write the cost expression for a join which is the residual join without the auxiliary relations. Hence, an auxiliary attribute appears only in one relation, hence it is dominated by an ordinary

[5] Again remember that the auxiliary attributes and relation are only used in the conceptual level.

[6] Of course, we do not include any more tuples in S.

(non-HH in this residual join) attribute. There is only one exception: when all attributes in a relation are auxiliary attributes. In this case, there is only one tuple in the relation for this particular residual join, so all attributes in the relation get a share equal to 1. This means that the term for this relation is removed from the cost expression, which agrees with the SharesSkew algorithm (according to which this tuple is sent to all reducers, hence the communication cost for this relation is equal to the number of reducers). □

Thus we established the following which shows that join J' on dataset $\mathcal{D}'$ is skew-free:

- Each tuple is hashed to reducers according to the values of the non-HH attributes in this tuple.

9.3. Lower Bound for 2-way Join

Here we prove that the solution in Example 2 is optimal because it meets the lower bound that we compute in this section.

Suppose r_q and s_q is the number of tuples in each reducer from relations R and S respectively. Suppose $r_q = \xi s_q$. Let communication cost be denoted by c. Then $c = k(r_q + s_q) = ks_q(1+\xi)$. The total output is rs and the output from all reducers is $kr_q s_q = k\xi s_q^2$. Thus we have the inequality:

$$rs \leq k\xi s_q^2$$

Hence

$\sqrt{rs/(k\xi)} \leq s_q$. Thus from the first equation above about the cost we have:

$c \geq k(1+\xi)\sqrt{rs/(k\xi)}$ or $c \geq (1+\xi)\sqrt{krs/\xi}$

But for all ξ we have $(1+\xi)/\sqrt{\xi} > 2$. Hence,

$$c \geq 2\sqrt{krs}$$

which is the exact same cost as the one we computed in Example 2.

Given a dataset that is distributed from the mappers to the reducers, we define the *distribution mapping* that maps each tuple t to a number of reducers $R_{t1}, R_{t2}, \ldots$ as pairs $(t, R_{t1}), (t, R_{t2}), \ldots$. We proved above the following theorem:

Theorem 3. *Given a join and a dataset $\mathcal{D}$, for any residual join, there is a dataset $\mathcal{D}'$ with no HH and with tuples that can be mapped one-to-one to tuple of $\mathcal{D}$ such that the following is true: when we apply SharesSkew to $\mathcal{D}$ the result is the same distribution function as when we apply Shares to $\mathcal{D}'$.*

This theorem proves that the good properties of Shares are transferred to SharesSkew.

10. Benefits of SharesSkew algorithm

When the Shares algorithm is applied on random non-skew data, then each reducer gets approximately the same amount of data because for any relation the following holds: For any of its attributes, say A, and any given value to A, say $A = a$, then the number of tuples with $A = a$ does not depend on a. There is also formal evidence of that (e.g., in [2]). Thus, when we are given a certain upper bound, q, on the size of a reducer and minimize communication cost as in Shares, we result with an even distribution of the tuples in the reducers. Now, in Section 9 we showed that, for each residual join, in terms of tuples distribution by the algorithm the data look like random data. Hence, for each residual join the relevant data is distributed evenly and each reducer gets an amount of data almost equal to q.

In conclusion, the SharesSkew algorithm retains the good properties of Shares algorithm on skewed data, i.e.:

- It distributes the tuples evenly to the reducers. Hence its performance scales with the number of reducers.
- Its performance does not depend on how much skew we have in the data.
- It is particularly useful when we have large tuples (e.g., that may contain images) where the shuffle time increases considerably.

11. Experimental evaluation

We have implemented the SharesSkew algorithm in the MapReduce framework.[7]

In this section, we report experimental results to evaluate the performance of the algorithm under a multitude of settings. The purpose of these experiments is to show the scalability of the SharesSkew algorithm and, also, how it constitutes an improvement compared with the Shares algorithm. We do not address questions about the upper data size limit of the approach, as this depends on the technology available at the time.

In particular, we examine the scalability of our approach for different inputs both in terms of input size and data skewness. The algorithm's performance is also evaluated against clusters with a varying number of available compute nodes in order to explore its suitability for different needs and setups. Furthermore, we compare the proposed approach to the Shares algorithm, the established multiway join algorithm which however does not internalize the presence of skew.

11.1. Experimental Setup

We ran the experiments on Hadoop version 2.6.0 (Cloudera CDH 5.4.2 distribution on CentOS 6.3 Linux) running on top of a 40 machines cluster (AMD Opteron 6320, 2.8 GHz, 8 cores per machine) which amounts to a total of 320 available compute nodes (vCores) in Hadoop YARN2.

We implemented SharesSkew using the `mrjob` (v4.4.4) Python library, which is a wrapper around Hadoop Streaming.

11.2. Anatomy of the evaluation

Every result presented in this section is an average of three runs. We repeat here the anatomy of a SharesSkew job together with a brief explanation of the functionality and impact of the SharesSkew distinct parts.

- Initially, the shares for the different residual joins J_i are computed. This process is performed only once and *not* under the distributed MapReduce setting in the *preMap* Stage (Section 6.1). The calculated shares of each residual join are subsequently distributed to each map task before the beginning of the MapReduce job. The preMap cost is negligible both in computation and communication terms. The calculated shares are serialized into a `cPickle` file with a file size of minuscule size ($\leq$0.5 KBytes) and sent to each cluster node as an execution parameter.
- The map tasks are entrusted with the correct distribution of the input tuples to the reduce tasks having taken into account the shares parameter for each of the residual joins J_i and their respective schemata. We have found that the total map time is tightly connected to the size of the input (see for example Fig. 7) and the same applies to a single map task for the fraction of the input assigned to it. Hence, it should be noted that processing time does not differ between map tasks.

[7] We have published both the code and the `iPython` notebooks with data to reproduce our experimental results along with all figures reported in the rest of this section at https://www.github.com/nstasino/sharesskew.

- SharesSkew aims at minimizing the communication and thereafter, reduce computation cost. We analyze the total reducer cost as the *composition* of reduce shuffle (i.e. communication) cost and actual join computation performed on each reduce node and in total. Thus, for every experimental run presented onward we purposefully analyze this composition (see for example when we discuss this in Section 11.4.2).

Evaluation metrics

We list here runtime metrics used throughout the rest of this section. (All times displayed in seconds):

- **Total Elapsed time**: End-to-end running time for the job
- **Average Map time**: Average time taken by a map task of the job
- **Average Shuffle time**: Average time attributed to the shuffle (communication) for the job
- **Average Reduce time**: Average time by a reduce task of the job

For the following metrics we report descriptive statistics (i.e. error-bars) to accentuate how the SharesSkew nodes handle their skewed input. We are especially interested in identifying lagging reducers which impede job completion.

- **Reduce Shuffle time**: The time taken for the redistribution of pertinent tuples from the mappers to the appropriate reducer. Reduce shuffle time is different, in general, from reducer to reducer and may be thus not the same as the average shuffle time.
- **Reduce Compute time**: Time taken by the reducer to compute the join on its input tuples

11.3. Dataset and queries

For the purposes of the current exposition we apply SharesSkew on two different datasets. In all of the following experiments where we changed the cluster size, the input size and/or the level of skewness, we used the first dataset. The second dataset is only used in the last experiment.

- We evaluated our running multiway example (discussed in detail in Section 4.1)

  $R(A, B) \bowtie S(B, E, C) \bowtie T(C, D)$

  using both SharesSkew and Shares as a benchmark.
  We used two sets of test data:
  1. Our first test dataset was generated as follows: For different levels of skewness, ranging from 0.1% up to 40%, we picked integer attribute values at random for the non-HH (ordinary) tuples. Heavy Hitter values were fixed integers according to skewness levels. For the particular join, there exist one HH value for attribute B, say b_1, and two HHs for attribute C, c_1 and c_2.
     The input size varies from 1 million tuples to 50 million tuples per relation. We allowed the number of concurrent reduce nodes k to stretch from 4 to 1024 in order to evaluate performance in differently sized clusters. A 20 million tuples relation approximates roughly to 1.3 GB. In our example query, the cost of joining three 1.3 GB relations is $(1.3\,GB)^3 \times p_1 \times p_2$ (where p_i is the probability that tuples from each of the two relations join). This far exceeds the capabilities of a large cluster such us ours which has a total of 2 TB of RAM and 4 GB of RAM per reduce node).
     In general, the communication cost can affect the performance of the algorithm in an independent way from the reduce time which is only affected by the reduce function. Since the communication cost depends on the size of the data, we run experiments to illustrate how huge data affects the performance of the SharesSkew on the side of communication cost and independently of the reduce function. In Fig. 4, we demonstrate that shuffle time (i.e. communication cost) takes up a large portion of the computation. Furthermore, map time is analogous to the total number of input tuples, however shuffle time evolves more aggresively for increasing input. Thus, it makes sense to employ SharesSkew to mitigate such effects.
  2. For the second dataset, we considered a Zipfian distribution of the data. More specifically, the frequency of any value for an attribute is given by the Zipf pdf:

     $$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)}$$

     where N is the dataset size, s is the characteristic skewness factor of the distribution and k is the rank of the HH value. A HH value with the highest rank (i.e. $k = 1$) is the most common in the data and its frequency is the highest. We tested against two dataset sizes $N = 100000$ and 1000000 tuples, with the skewness s factor ranging from 1.00 to 1.30. We give an indicative table of HH value frequencies with respect to the rank k and the skewness factor s in Table 2.
     The skewness level in the case of the datasets we used in the non-Zipf experiments is the percentage of HHs over the entire dataset. Hence, skewness level in a Zipfian dataset is proportional to the sum of the frequencies of all HHs (in this case, these are values with frequencies larger than a certain threshold). Although it is not accurate, we can say that, in general, for large datasets, the skewness level increases as the skewness factor decreases.

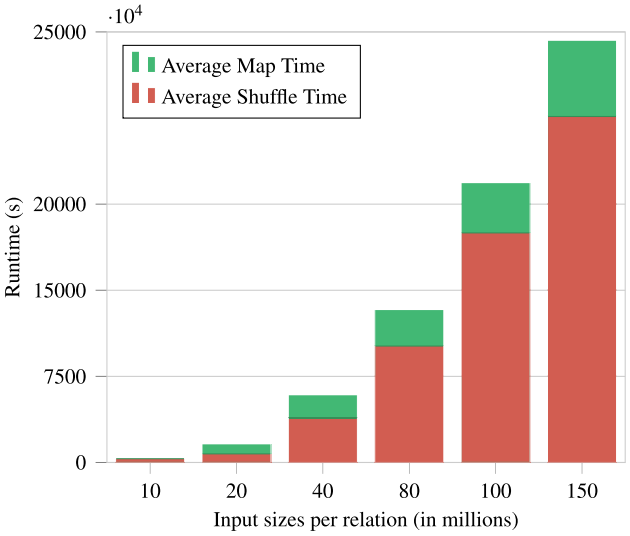

Fig. 4. Map, shuffle and reduce times for varying input 10 to 150 million tuples per relation, $k = 256$, skew 10%.

Table 2
HH value Zipfian frequency for varying skewness factor.

Skewness factor s	1.00	1.10	1.20	1.30
HH rank k				
1	0.07	0.12	0.19	0.26
2	0.03	0.06	0.08	0.10
3	0.02	0.04	0.05	0.06
4	0.02	0.03	0.04	0.04
5	0.01	0.02	0.03	0.03

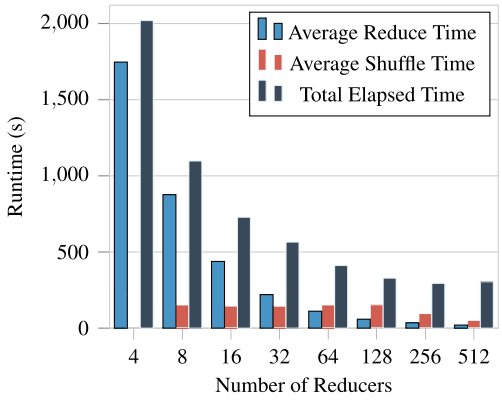

Fig. 5. Reduce times for $k = 4...512$ reduce nodes, input 30 million tuples, skew 10%.

- We have also used the TPC-DS dataset [31] to assess the performance of SharesSkew on multiway joins. To this end we run a modified `Query 6` so as it only contains the join predicates on tables Customer, Customer Address and Store Sales.

```
SELECT a.ca_state state
FROM customer_address a, customer c, store_sales s
WHERE a.ca_address_sk = c.c_current_addr_sk
    and c.c_customer_sk = s.ss_customer_sk;
```

The input sizes for the relations were 1000000, 2000000 and 57000000, a total of 60 million tuples. We notice a 5% skew on attribute *ss_customer_sk* which we identify as the main source of skewness in this experiment.

11.4. Scalability

11.4.1. Increasing the number of reducers

In this subsection we demonstrate how the algorithm behaves for a changing cluster size and give an idea of the parallelization achieved. In Fig. 5, the dataset size is 10 million tuples per relation and skewness factor is at 10%. Total elapsed, average shuffle and average reduce times for increasing number of reduce nodes are reported. As expected by the previous analysis, as we approach full utilization of the cluster's compute nodes (i.e. when $k \to 256$) total elapsed time improves. In addition note that, as the number of used nodes grows higher, total shuffle time (i.e. the realization of the communication cost) remains practically stable. This is not trivial, since the need to shuffle tuples to the reducers is usually higher the more reducers we use. SharesSkew aims at minimizing communication cost while achieving higher job parallelization; the positive effect of which is reflected on the consistently decreasing reduce times. Naturally, such gains decay when SharesSkew is instructed to use more reduce nodes that physically available. Such is the case for $k \geq 256$.

Figs. 6a and b demonstrate how shuffle (communication) and reduce (computation) times change for a different number of available reduce nodes i.e. for $k = 4$ up to 512. In Fig. 6a we see that shuffle times are fairly the same for $k = 8$ to 128 and are falling for higher numbers of reducers ($k = 256$ and 512). This is due to the overhead introduced during shuffling which is not proportionally depending on the total size of the input. Also, it must be noted that when the cluster is small ($k = 4$) paralellization is low, thus shuffle time per reducer is high and also reduce time per reduce task is also high. The beneficial effect of parallelization is thus twofold: in both the communication and computation phases and SharesSkew is designed to bring both.

As shown in Fig. 6b when the number of reducers k grows, parallelization kicks in and this leads to lower running times for the reduce tasks. In addition, as indicated by the error bars, reduce times have a small standard deviation. This is a desired result since it is an empirical metric of good reduce task load balancing.

11.4.2. Increasing the size of the input

In this section, we show how SharesSkew performs when presented with increasing input dataset sizes. Under the presence of a constant 10% skew, we let input vary from 1 million to 50 million tuples per relation. We have selected the cluster size $k = 256$ which is very close to its resource capacity.

In Fig. 7 we give a synopsis for the running times for the map, shuffle and reduce phases of SharesSkew. In particular, average map time increases as input increases, since the same number of mappers must process an increasing amount of input tuples. Furthermore, average shuffle time increases slightly with input size. Notice that input size here is per relation and we have three relations with the same number of tuples. The main take from the reduce phase obtains from the small error bars. This is a clear indication that reducers in any input size finish roughly at the same time and that there are not reducers that lag significantly, thus stalling the whole job[8]. Hence, this fact demonstrates the good property of SharesSkew that it gives good load balance at the reduce side.

11.5. Skew resilience

We examine SharesSkew behavior under different degrees of skew, input sizes and dataset types.

SharesSkew vs Shares on a dataset with skew

In the first dataset used, skew is represented by integer HH values for attributes B and C, namely b_1 and c_1, c_2. For the join in hand, $R(A, B) \bowtie S(B, E, C) \bowtie T(C, D)$, the joining attributes are B and C. Hence, having HH values on both of them is putting heavy load on the reduce side for higher levels of skewness.

In this section, we also compare our findings with the standard Shares algorithm [6] for the multiway join. We do this for three different input sizes, namely 100 thousand, 1 million and 10 million tuples per relation (and we have three of them in each case).

In Figs. 8a, 9 a, 10 a, we give map times for both SharesSkew and Shares under various skew. In all three cases, SharesSkew mappers are (marginally) slower, since they need to assign and possibly replicate each of their input tuples for more than one residual joins, which is instead the case for Shares. Furthermore, notice that map times do not change for different levels of skewness in any input size. In the cases that map time does not appear in the figures for Shares algorithm, the task could not finish or has failed due to memory limitations. These limitations, as explained before, are due to the fact that replication becomes too big for each cluster node to handle. The same rationale follows for shuffle, reduce and total figures below. SharesSkew manages to finish for higher skew than Shares.

In Figs. 8b, 9 b, 10 b, we report shuffle times. When input is small (i.e. 100k tuples per relation), shuffle is very short and we cannot draw a conclusion of whether SharesSkew shuffles better than Shares. When input increases especially to 10 million tuples per relation, shuffle time increases in general for both algorithms since now many more tuples are transferred around. However, for

[8] This phenomenon is commonly referred as *the curse of the last reducer.*

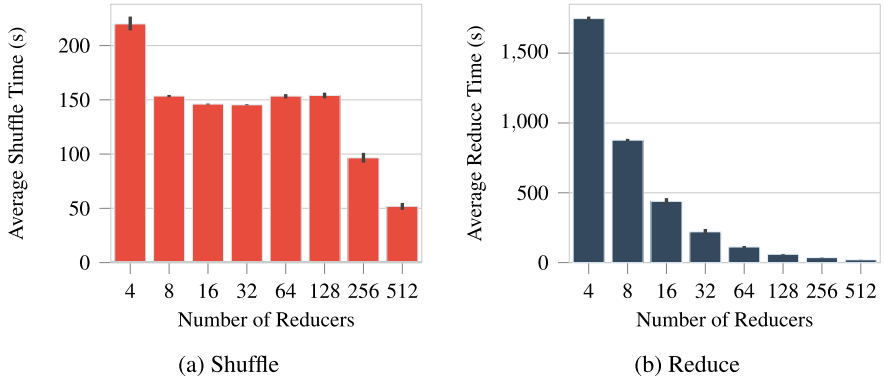

Fig. 6. Shuffle and Reduce times for k = 4...512 reduce nodes, input 30 million tuples, skew 10%.

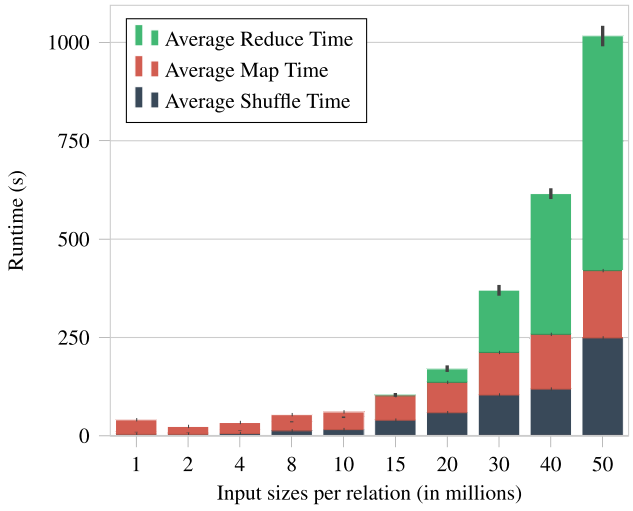

Fig. 7. Map, Shuffle and Reduce times for varying input 1 to 50 million tuples per relation, $k = 256$, skew 10%.

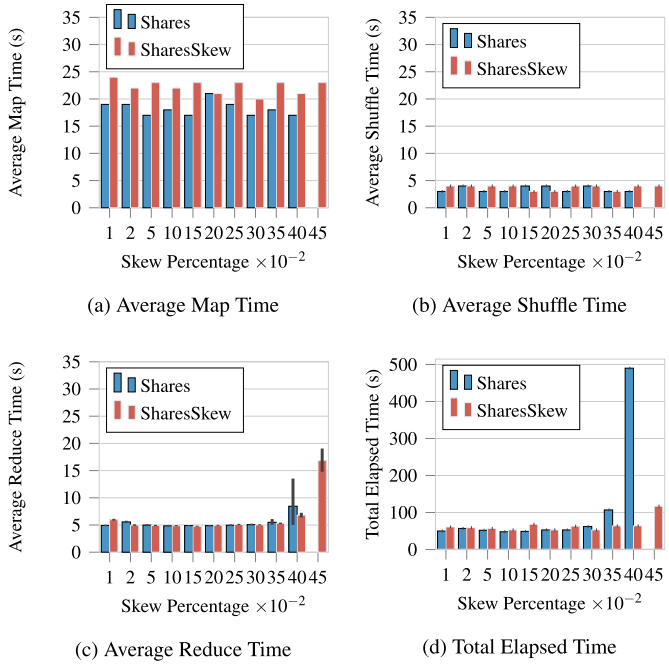

Fig. 8. Running times for varying skew 0.01 to 0.45, $k = 256$, input 100k tuples per relation.

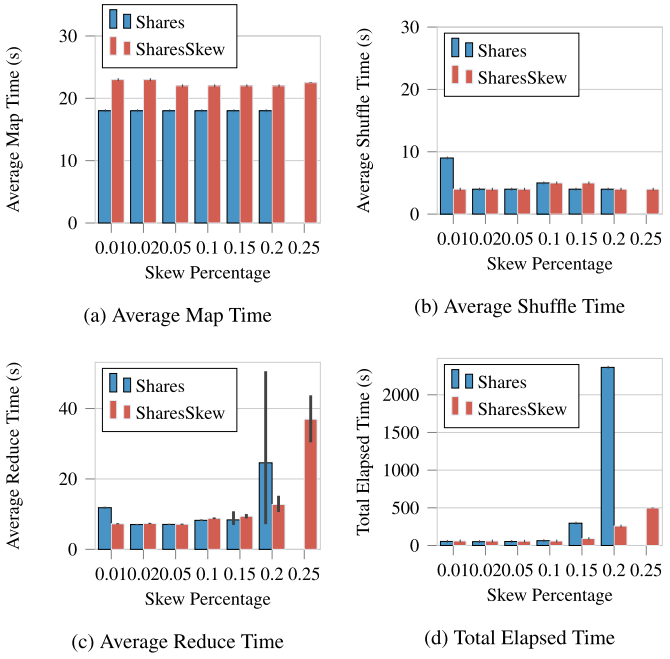

Fig. 9. Running times for varying skew 0.01 to 0.25, $k = 256$, input 1 million tuples per relation.

any skew percentage, we do not see a definite increase/decrease of shuffle time. Notice however that Shares doesn't manage to finish when skew and input both grow. Shuffle times are consistent with the main premise SharesSkew was designed; i.e. handle skew by extending the Shares idea of optimally performing multiple Shares for each residual multiway join. Hence, SharesSkew shuffle is very close to Shares. The advantage of SharesSkew over Shares is becoming more evident in Figs. 9b and 10 b where data size is higher and when skew is higher.

In Figs. 8c, 9 c, 10 c, we showcase the main advantage of the SharesSkew approach. There is a point after which for any of the three input sizes considered in this experiment when Shares reducers start to face uneven load. This occurs when skew is 0.4% for 100k, 0.2% for 1 million and, in the case of 10 million tuples per relation for skew equal to 0.03%. Shares reducers have failed due to resource (memory) limitations. This is reported with the error bars which indicate that the "curse of the last reducer" is real when there is skew. SharesSkew handles this situation much better even for large skewness levels up to 0.2% for large input sizes. Also, evident and equally important is the distribution of average reduce times which are far from uniform in Shares; SharesSkew

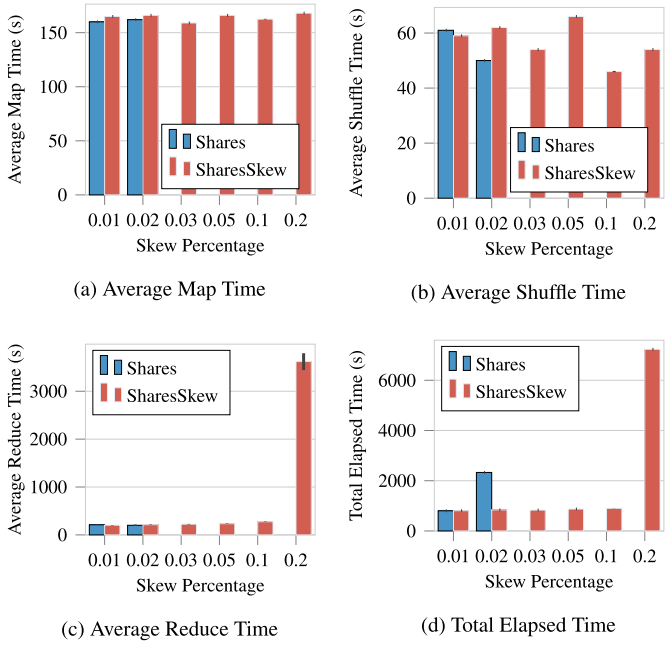

Fig. 10. Running times for varying skew 0.01 to 0.2, $k = 256$, input 10 million tuples per relation.

performs better. Since some Shares reducers finish later than others, the whole tasks has to wait for them, thus the total completion time is much higher. When we focus just to SharesSkew, performance is worsened only at higher skew levels as shown in Fig. 10c. However, due to effective reduce compute parallelization and reduce time distribution, total time is not affected by the same factor as the average reduce time.

Finally, in Figs. 8d, 9 d, 10 d, it is shown that lagging reducers make whole jobs lag in the Shares case and SharesSkew largely takes care of these problems.

We give now the rationale behind the good properties of the proposed algorithm: SharesSkew optimizes for communication cost given the q-constraint, where q is a set reducer capacity; i.e. it is equal to the number of tuples a reducer can handle. A reducer evaluates its received tuples and joins them by means of the hashjoin algorithm.

Of course, having the same input size reduce-side does not predetermine computation load. Suppose a reducer computes $R(A, B) \bowtie S(B, E, C) \bowtie T(C, D)$ for *non-HH* tuples that match with probability p. If $q = 10000$ and $q_R = q_T = 3000$, $q_S = 1000$ and $p = 0.001$, 9000 tuples will be returned. In contrast, suppose that a reducer is tasked with joining only *HHs* with identical q_R, q_S, q_T. In this case, every tuple participates in the join and 9 billion tuples are returned. The intermediate join hash tables are also much bigger in this case.

TPC-DS dataset

In Fig. 11, we present the results of evaluating a version of Query 6 containing a multiway join on three relations on the TPC−DS dataset. Pitting SharesSkew against Shares results in the same map times for both algorithms. However, SharesSkew has much lower communication cost (shuffle) and the reduce side computation is much less skewed. Not only do reducers finish quicker in SharesSkew but they tend to do it at roughly the same time, whereas there are more lagging reducers with Shares.

11.6. Zipfian distribution of the data

We have tested the Shares and SharesSkew algorithms against the second dataset that follows Zipf's law for the same join $R(A, B) \bowtie S(B, E, C) \bowtie T(C, D)$. We report map, shuffle, reduce and total elapsed times for two dataset sizes with full cluster utilization, that is when the number of reducers is $k = 256$. Notice that in the case of 1 million tuples per relation for $s \geq 1.30$ and in the case of 10 million tuples per relation for $s \geq 1.15$ the algorithm Shares fails to finish due to high skewness. However, SharesSkew manages to finish the job for all s reported in the graphs. The small dots/lines on the top center of the bars are again error bars. A wide error bar indicates that there are reduce nodes that finish much later than the average.

In Figs. 12a, 13 a map times are reported. SharesSkew is always slightly slower in the map phase than Shares, however (a) the map phase takes up a small part of the total job and (b) SharesSkew map times does not really fluctuate. Remember that SharesSkew has more work to do in the map phase, since it replicates tuples according to the appropriate residual joins.

In Figs. 12b, 13 b, shuffle times are reported. SharesSkew performs just as good as Shares in the case of 1 million tuples. In the case of 10 million tuples (Fig. 13b), the good properties of SharesSkew are again demonstrated. For low skewness (i.e. $s = 1.0$ or 1.05), there isn't a significant shuffle advantage for SharesSkew. However, things change for higher levels of skewness ($s \geq 1.1$): SharesSkew starts to shuffle better and for $s \geq 1.15$ is the only algorithm that manages to finish its job.

In Figs. 12c, 13 c, reduce times are reported. Again, for lower skewness Shares does not perform bad, but things change quantitatively and qualitatively for higher skewness levels. SharesSkew is at the same or much better on average for s up to 1.25 in the 1 million tuples case. More interestingly, the errors bars are really wider for Shares (see for example $s = 1.25$); this means that the last reducers finish much later than SharesSkew's reducers. This "struggler" nodes stall the job, since unused reducers are waiting for them to finish their individual tasks. This is evident in Fig. 12d for $s = 1.25$, where Shares performs much worse that SharesSkew. We can identify the same behavior in the case of 10 million tuples in Figs. 13c and d. Here problems for Shares arise for even lower skewness factors ($s = 1.05$ and 1.10) and Shares does not manage to finish for higher skew. SharesSkew is really at an advantage from early on. The algorithm is able to complete even for very high skewness factors.

We further discuss how the varying data distributions affect an algorithm such as SharesSkew which is designed to tackle skewed dataset distribution in the following section. One takeaway from the performance of SharesSkew on both the first and second datasets is that it is actually the first ranked HH value ($k = 1$) that really dominates the algorithm's performance.

11.7. Various distributions of data

The following example does not follow the Zipf law $n(d) = ae^{-\gamma \ln d}$ where d is the number of tuples in which a certain value appears and $n(d)$ is the number of values that appear in d tuples. Instead, it follows the distribution $n(d) = ae^{-\gamma d}$ which does not have such a heavy tail and gives better insights into SharesSkew.

Example 11. Suppose we have the two way join that joins relations $R(A, B)$ and $S(B, C)$. We will develop two scenarios. In the first scenario, attribute B has no HH in relation S but it has HH in relation R. In the second scenario, attribute B has HH in both relations.

For both scenarios, we have the following distribution: The first column gives the number of tuples in which a specific value appears and the second column gives the number of values that have this property. E.g., the first row says that we have 100,000 values that have the property that each of this values appears in 10 tuples (i.e., $d = 10$), the second row says that we have 10,000 values that

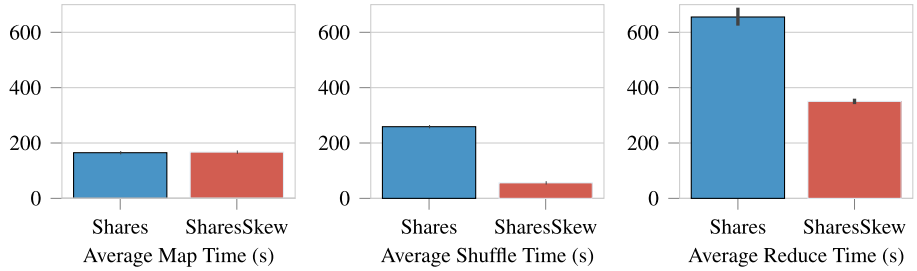

Fig. 11. Running times for the TPC-DS dataset.

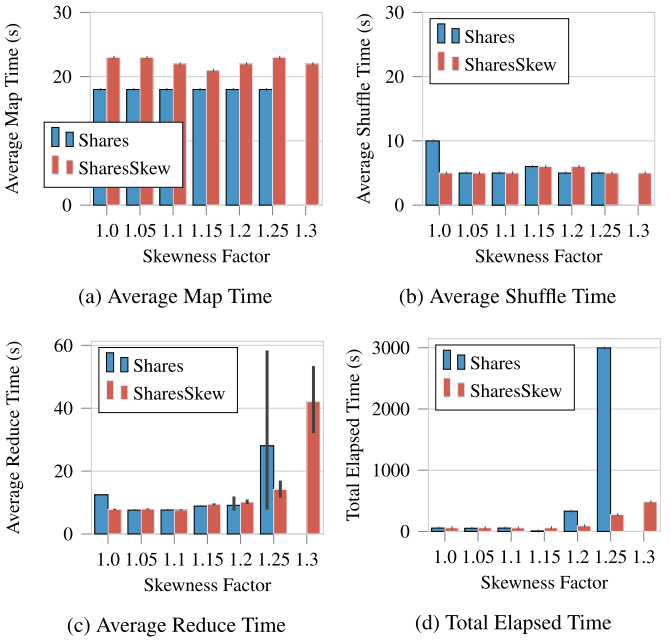

Fig. 12. Running times for varying Zipfian skewness factor 1.00 to 1.30, $k = 256$, input 1 million tuples per relation.

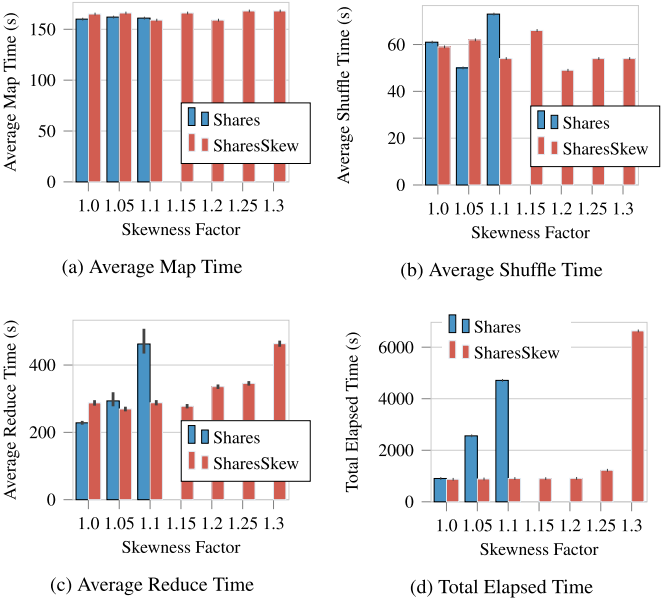

Fig. 13. Running times for varying Zipfian skewness factor 1.00 to 1.30, $k = 256$, input 10 million tuples per relation.

Table 3
Distribution of data.

d=number of tuples	number of values
10	100,000
100	10,000
1,000	1,000
10,000	100
100,000	10

have the property that each of this values appears in 100 tuples. The last row says that we have only 10 values where each of this values appears in 100,000 tuples.

First scenario. We assume that each value of B appears in 2 tuples in relation S and that for relation R, we have the distribution in Table 3.

We assign 10000 reducers to the values in the last row. Thus, each value of B is handled by 1000 reducers. Hence, each reducer has a load of 100 tuples from relation R and 2 tuples from relation S, in total 102 tuples.

We also assign 10000 reducers to the values in the row before last. Thus, each value of B is handled by 100 reducers. Hence, each reducer has a load of 100 tuples from relation R and 2 tuples from relation S, in total 102 tuples.

We assign also 10000 reducers to the values in the third row. Thus, each value of B is handled by 2 reducers. Hence, each reducer has a load of 100 tuples from relation R and 10 tuples from relation S, in total 102 tuples.

We assign also 10000 reducers to the values in the second row. Thus, each value of B is handled by 1 reducer (these values of B are not handled any more as HH since they fit in one reducer). Hence, each reducer has a load of 100 tuples from relation R and 2 tuples from relation S, in total 102 tuples.

We assign also 10000 reducers to the values in the first row. Now each reducer handles 10 values of B (similarly, these values of B are not handled any more as HH since they fit in one reducer). Now each reducer has a load of 100 tuples from relation R but, from relation S has a load of $10 \times 2 = 20$ tuples, thus, in total 120 tuples. Thus, we have used in total 50000 reducers where each reducer has a load of 102 to 120 tuples. Thus the communication cost (if we take the load of each reducer approximately to be 100 tuples) is 5,000,000.

Second scenario. Here each value of B has the same distribution in relation S as in relation R. Thus, for the last row, for each value, we have to send to each reducer 100 tuples from relation R and 100 tuples from relation S and do that 1000 x 1000 = 1,000,000 times, i.e., we need to use 1,000,000 reducers for each value. Since we have 10 values, we need to use 10,000,000 reducers with 200 tuple load for each one of them. We will not continue the analysis for this scenario, since the picture is already obvious, i.e., here the communication cost rises to more than 2,000,000,000 (in comparison to 5,000,000 for the first scenario).

Of course, it is easy to observe that for reducers that can hold up to 200,000 tuples each, the communication cost will be almost equal in both scenarios (because, simply, we have no HH) and this communication cost will be almost equal to the size of the data, i.e., approximately 5,000,000. Observe that this is the communication cost in the first scenario, and actually, this is the reason we analyzed these two scenarios. The communication cost of the first scenario is not affected by the size of each reducer because the tuples are not duplicated to multiple reducers.

Of course, these two scenarios show two extreme cases. Typically, we expect that a few values of B will behave like the second scenario and the rest will behave like in the first scenario. In such a case, the behavior of the communication cost would be smoother.

12. Conclusions

This paper presents SharesSkew, an algorithm for computing efficiently any multiway join on highly skewed data in a single round of MapReduce. SharesSkew is an extension of the Shares algorithm and is based on the idea of splitting the data into pieces, each piece either not containing any heavy hitters or containing heavy hitters in specific attributes which are known to the algorithm. This knowledge allows the algorithm to distribute the input across a number of reducers, so that each reducer does not receive more than a certain amount of data. This idea is executed by defining residual joins, each of which can be computed independently by Shares algorithm with minimum communication cost. The data for each residual join is not skewed any more, each containing data that are relevant to a combination of specific heavy hitter values.

It also starts looking on specific multiway joins which either are resilient to skewness or would perform much better on a multi-round MapReduce algorithm. These are observations that may be very useful in practice and families of important (in practical applications) multiway joins can be considered and further investigated for efficient multi-round MapReduce algorithms. A scenario where the multi-round approach will be beneficial is when data are rather irregularly distributed across relations. E.g., even when there is no skew, if there are some huge relations that are highly connected in the multiway join, whereas the rest of the relations are of smaller size and form a chain (which is a multiway join with high communication cost) then we should compute the former sub-join and the chain separately and then in a second MapReduce round, join them together with minimum communication cost.

Acknowledgements

This work was supported by the project Handling Uncertainty in Data Intensive Applications, co-financed by the European Union (European Social Fund - ESF) and Greek national funds, through the Operational Program Education and Lifelong Learning, under the Program THALES.

Appendix A. Dominance Rule in Algorithm 1

Algorithm 1 determines whether one attribute dominates another for a given residual join. We demonstrate it via the join example $R(A, B)\bowtie S(B, E, C)\bowtie T(C, D)$ where there exist two HH values for attribute B and one for attribute C.

The inputs of the algorithm are the relations sizes r, the HH values HH and the schemaBinaryMatrix, a matrix that indicates whether an attribute is present in a relation and which for the given query is given in Tables A1–A4:

We use a symmetric dominance matrix which will hold the information about dominance between attributes. We initialize it by putting with -1 in each shell (line 2):

Table A1
schemaBinaryMatrix for $R(A, B)\bowtie S(B, E, C)\bowtie T(C, D)$.

	A	B	C	D	E
R	1	1	0	0	0
S	0	1	1	0	1
T	0	0	1	1	0

Table A2
Initialized dominance matrix.

	A	B	C	D	E
A	−1	−1	−1	−1	−1
B	−1	−1	−1	−1	−1
C	−1	−1	−1	−1	−1
D	−1	−1	−1	−1	−1
E	−1	−1	−1	−1	−1

Table A3
First update to the dominance matrix.

	A	B	C	D	E
A	0	0	−1	−1	−1
B	0	0	0	−1	0
C	−1	0	0	0	0
D	−1	−1	0	0	−1
E	−1	0	0	−1	0

Table A4
Second update to the dominance matrix.

	A	B	C	D	E
A	0	0	−1	−1	−1
B	0 → 1	0	0 → 1	−1	0 → 1
C	−1	0 → 1	0	0 → 1	0 → 1
D	−1	−1	0	0	−1
E	−1	0	0	−1	0

The dominance matrix is updated twice. First, lines 4-10 mark whether two attributes (i, j) are in the same relation (k) at least once, i.e. for every row k,

$$schemaBinaryMatrix(k, i) == 1 \ \& \ schemaBinaryMatrix(k, j) == 1$$

Now, the dominance matrix has been updated to:

Second, in line 12-18 of the algorithm, we check whether (a) there is at least one relation where one attribute is there and the other is missing -

$$schemaBinaryMatrix(k, i) == 1 \ \& \ schemaBinaryMatrix(k, j) == 0$$

and at the same time (b) two attributes co-exist in at least one relation.

$$dominanceMatrix(j, i) == 0$$

The dominance matrix is now:

Note that in relation S: attribute B is there, but A is not. But we have established that A and B coexist in another relation (i.e. R) (cell (B, A) was 0). So $(B, A) \rightarrow 1$.

In lines 24-33, we check whether for any ordered pair of attributes the relevant cells in the dominance matrix are exclusively either 0 or 1. For example since the cell (A, B) is equal to 0 but (B, A) is 1, we deduce that attribute B dominates A. By the same token, B dominates E when we look at relation R. Also, C dominates D, E.

References

[1] F.N. Afrati, V.R. Borkar, M.J. Carey, N. Polyzotis, J.D. Ullman, Map-reduce extensions and recursive queries, in: EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings, 2011, pp. 1–8, doi:10.1145/1951365.1951367.

[2] P. Beame, P. Koutris, D. Suciu, Communication steps for parallel query processing, in: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013, 2013, pp. 273–284, doi:10.1145/2463664.2465224.
[3] J.D. Ullman, Designing good mapreduce algorithms, XRDS 19 (1) (2012) 30–34, doi:10.1145/2331042.2331053.
[4] M.R. Joglekar, C.M. Ré, Its all a matter of degree: Using degree information to optimize multiway joins, in: 19th International Conference on Database Theory, 2016.
[5] P. Beame, P. Koutris, D. Suciu, Skew in parallel query processing, in: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014, 2014, pp. 212–223, doi:10.1145/2594538.2594558.
[6] F.N. Afrati, J.D. Ullman, Optimizing multiway joins in a map-reduce environment, IEEE Trans. Knowl. Data Eng. 23 (9) (2011) 1282–1298, doi:10.1109/TKDE.2011.47.
[7] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, R. Murthy, Hive - a petabyte scale data warehouse using hadoop, in: Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA, 2010, pp. 996–1005, doi:10.1109/ICDE.2010.5447738.
[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, 2008, pp. 1099–1110, doi:10.1145/1376616.1376726.
[9] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, U. Srivastava, Building a highlevel dataflow system on top of mapreduce: The pig experience, PVLDB 2 (2) (2009) 1414–1425.
[10] J.L. Wolf, D.M. Dias, P.S. Yu, A parallel sort merge join algorithm for managing data skew, IEEE Trans. Parallel Distrib. Syst. 4 (1) (1993) 70–86, doi:10.1109/71.205654.
[11] P. Koutris, P. Beame, D. Suciu, Worst-case optimal algorithms for parallel query processing, in: 19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016, 2016, pp. 8:1–8:18, doi:10.4230/LIPIcs.ICDT.2016.8.
[12] S. Chu, M. Balazinska, D. Suciu, From theory to practice: Efficient join query evaluation in a parallel database system, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, 2015, pp. 63–78, doi:10.1145/2723372.2750545.
[13] H.Q. Ngo, C. Ré, A. Rudra, Skew strikes back: new developments in the theory of join algorithms, SIGMOD Record 42 (4) (2013) 5–16, doi:10.1145/2590989.2590991.
[14] H.Q. Ngo, E. Porat, C. Ré, A. Rudra, Worst-case optimal join algorithms: [extended abstract], in: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012, 2012, pp. 37–48, doi:10.1145/2213556.2213565.
[15] H.Q. Ngo, D.T. Nguyen, C. Re, A. Rudra, Beyond worst-case analysis for joins with minesweeper, in: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014, 2014, pp. 234–245, doi:10.1145/2594538.2594547.
[16] Y. Kwon, M. Balazinska, B. Howe, J.A. Rolia, Skewtune in action: Mitigating skew in mapreduce applications, PVLDB 5 (12) (2012) 1934–1937.
[17] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, Skewtune: Mitigating skew in mapreduce applications, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, in: SIGMOD '12, ACM, New York, NY, USA, 2012, pp. 25–36, doi:10.1145/2213836.2213840.
[18] S. Huang, A.W. Fu, (α, k)-minimal sorting and skew join in MPI and mapreduce, CoRR abs/1403.5381 (2014).
[19] Y. Tao, W. Lin, X. Xiao, Minimal mapreduce algorithms, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, 2013, pp. 529–540, doi:10.1145/2463676.2463719.
[20] S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, Y. Tian, A comparison of join algorithms for log processing in mapreduce, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, 2010, pp. 975–986, doi:10.1145/1807167.1807273.
[21] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E.N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, X. Zhang, Major technical advancements in apache hive, in: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, 2014, pp. 1235–1246, doi:10.1145/2588555.2595630.
[22] C.B. Walton, A.G. Dale, R.M. Jenevein, A taxonomy and performance model of data skew effects in parallel joins, in: 17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings., 1991, pp. 537–548.
[23] Y. Xu, P. Kostamaa, X. Zhou, L. Chen, Handling data skew in parallel joins in shared-nothing systems, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, 2008, pp. 1043–1052, doi:10.1145/1376616.1376720.
[24] S. Wu, F. Li, S. Mehrotra, B.C. Ooi, Query optimization for massively parallel data processing, in: ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011, 2011, p. 12, doi:10.1145/2038916.2038928.
[25] D. Jiang, A.K.H. Tung, G. Chen, MAP-JOIN-REDUCE: toward scalable and efficient data analysis on large clusters, IEEE Trans. Knowl. Data Eng. 23 (9) (2011) 1299–1311, doi:10.1109/TKDE.2010.248.
[26] A. Okcan, M. Riedewald, Processing theta-joins using mapreduce, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011, 2011, pp. 949–960, doi:10.1145/1989323.1989423.
[27] X. Zhang, L. Chen, M. Wang, Efficient multi-way theta-join processing using mapreduce, PVLDB 5 (11) (2012) 1184–1195.
[28] F.N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, J.D. Ullman, GYM: A multiround join algorithm in mapreduce, CoRR abs/1410.4156 (2014).
[29] S. Salihoglu, Let us rethink join optimization in distributed systems, CIDR, 2015.
[30] S. Sakr, A. Liu, A.G. Fayoumi, The family of mapreduce and large-scale data processing systems, ACM Comput. Surv. 46 (1) (2013) 11, doi:10.1145/2522968.2522979.
[31] R.O. Nambiar, M. Poess, The making of TPC-DS, in: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006, 2006, pp. 1049–1058.