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