Efficient Large Outer Joins over MapReduce

Long Cheng[1(✉)] and Spyros Kotoulas[2]

[1] cfaed, TU Dresden, Dresden, Germany
long.cheng@tu-dresden.de
[2] IBM Research, Dublin, Ireland
spyros.kotoulas@ie.ibm.com

Abstract. Big Data analytics largely rely on being able to execute large joins efficiently. Though inner join approaches have been extensively evaluated in parallel and distributed systems, there is little published work providing analysis of outer joins, especially on the extremely popular MapReduce platform. In this paper, we studied several current algorithms/techniques used in large outer joins. We find that some of them could meet performance bottlenecks in the presence of data skew, while others could be complex and incur significant coordination overheads when applied to the MapReduce framework. In this light, we propose a new algorithm, called POPI (Partial Outer join & Partial Inner join), which targets for efficient processing large outer joins, and most important, is lightweight and adapted to the processing model of MapReduce. We implement our method in Pig and evaluate its performance on a Hadoop cluster of up to 256 cores and datasets of 1 billion tuples. Experimental results show that our method is scalable, robust and outperforms current implementations, at least in the case of high skew.

1 Introduction

In light of the explosion of available data and the increasing connectivity between data systems, the infrastructure for scalable data analytics is as relevant as ever. An essential operation in this domain is the join, which facilitates the combination of records based on a common join key. Since this data-intensive operation can incur significant costs, improving the efficiency of this operation would have a significant impact on the performance of applications.

Outer Join. Although distributed inner join algorithms have been widely studied [1,2], there has been relatively little done on the topic of outer joins. In fact, outer joins are common in complex queries and widely used such as in OLAP applications. For example, in online e-commerce, customer ids are often left outer joined with a large transaction table for analyzing the purchase patterns [3]. In contrast to inner joins, outer joins do **not** discard tuples from one (or both) table(s) that do not match with any tuple in the other table. As a result, the final join results contain not only the matched part but also the non-matched part. This difference makes outer join implementations significantly different from inner joins in a distributed system and challenge current techniques [4].

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 334–346, 2016.
DOI: 10.1007/978-3-319-43659-3_25

MapReduce. As applications grow in scale, joins on multiple CPUs and/or machines is becoming important. Compared to conventional parallel DBMSs, MapReduce (over Hadoop) integrates parallelization, fault tolerance and load balancing in a simple programming framework, and can be easily run in a large computing center or cloud, making it extremely popular for large-scale data processing. In fact, most vendors (such as IBM) provide solutions, either on-premise or on the cloud, to compute on massive amount of structured, semi-structured and unstructured data for their business applications.

In this light, studying analytic techniques on this platform becomes very important. In fact, join operations are sometimes hard in MapReduce [5]. Unlike implementations in DBMS'es, complex designs for joins in MapReduce can easily lead to poor performance: the overhead of starting a communication phase between partitions is very high. Namely, we have to start a new job and re-read (part of) the data. In addition, the MapReduce paradigm is highly sensitive to the presence of data or computation skew: since coordination is infrequent and very costly, there are fewer opportunities to re-balance workloads across nodes.

In comparison to most of current studies focusing on *inner* joins over MapReduce [2,5], in this work, we focus on the design and evaluation of outer joins on this platform. We summarize our contributions as following:

- We introduce several outer join implementations which are applied in MapReduce and discuss their possible performance issues.
- We discuss the possibility to apply some advanced join strategies used in parallel DBMSs to outer joins over MapReduce. We find that, they could either meet performance issues or be complex in implementations and thus bring in high overhead in terms of the number of MapReduce jobs launched.
- We propose a new approach, called POPI (Partial Outer join & Partial Inner join), which targets efficient outer joins adapted to MapReduce.
- We implement the various approaches on Apache Pig. Our experimental results show that our method is robust and can perform better than current implementations in MapReduce, at least in the presence of high skew.

The rest of this paper is organized as follows: In Sect. 2, we shortly introduce the MapReduce framework and describe current outer join implementations over it. In Sect. 3, we discuss some advanced strategies for large data outer joins in MapReduce. We describe our new approach in Sect. 4 and present the evaluation in Sect. 5. We report on related work in Sect. 6 while we conclude the paper in Sect. 7.

2 MapReduce and Outer Joins

Overview. MapReduce [6] is designed to operate over key/value pairs. Specifically, each *Map* function receives a key/value pair and emits a set of key/value pairs. All key/value pairs produced during the map phase are grouped by their key and passed to reduce phase. During the reduce phase, a *Reduce* function is called for each unique key, processing the corresponding set of values.

Though MapReduce has various advantages on large data processing, it entails more overhead compared to traditional DBMSs during execution: This platform sacrifices per-node efficiency, for scalability [5]. Namely, performance loss on a single node can usually be compensated by simply employing more computation resources. Nevertheless, MapReduce has no way of automatically re-balancing load, and any operation that changes the distribution of data should only be performed in the context of a new job (which typically incurs a coordination overhead of tens of seconds, if not minutes). Thus, achieving good load-balancing in data/join processing is critical.

Current Methods for MapReduce. Currently, three outer join methods are commonly applied in MapReduce implementations: hash-based, replication-based and histogram-based outer joins. We focus on *left* outer joins ($\ltimes$) here since they are the most common ones and their implementations would be analogous for right outer joins. In the following, we focus on a single outer join operation between two relations R and S. We assume both R and S are $<k, v>$ pairs with $|R| < |S|$ and k is the join key. For simplicity, we also assume that R is uniformly distributed and S is skewed for all of our examples, unless otherwise specified.

Hash-Based Outer Join. Similarly to inner join implementations, this approach can be done in a single MapReduce job. In the map phase, each map task works on either R or S. To identify which relation an input record is from, each map task tags the record with its originating table, and outputs the extracted join key and the tagged record. For example, for a record $< k_1, v_1 >$ from S, the output will be $< k_1, (s, k_1, v_1) >$ pair, where s is the table tag. Then, the framework brings together records sharing the same key and eventually feed them to a reducer, based on the hash value of their keys. In the reduce phase, the reduce function separates the input records into two sets according to their table tags and then performs a cross-product between each record in these sets and output the final results.

Normally, this scheme can achieve good performance under ideal balancing conditions for distributed systems [1]. However, when the processed records has significant skew, number of records will be flushed to a small part of reducers and cause hotpots. Such issues impact system scalability which will be reduced as employing new nodes[1] cannot yield improvements - the skew records will still be distributed to the same reducers.

Replication-Based Outer Join. Compared to the replication-based *inner* joins containing only a mapreduce job, outer joins within this scheme is significantly different. It is composed by two distinct join stages in an abstract level[2]: (1) A map-side inner join between R and S. Namely, all records of the small table R is retrieved from the DFS and then each map task uses a main-memory hash table to

[1] Note that, in terms of terminology, when we talk about a node, we mean a computing unit (e.g., a Reducer in MapReduce) in this work.

[2] The detailed process about how to identity where a record comes from is the same as the hash-based approach described above, thus here we do not present it again.

join S with R, formulating the intermediate results T; and (2) A reduce-side outer join between R and T, which is done in the same way as the hash-based method described above. Namely, all the records of R and T will be grouped based on their keys, and then fed to reducers for the local outer joins.

The *replication* in this method can reduce load imbalance, as each map task has the same workloads in the first phase. Nevertheless, this operation is costly and only suitable for small-large outer joins [3]. Moreover, even if R is small, the cardinality of the intermediate results T could be large when S is highly skewed [4]. This could make tasks in the second stage very costly and consequently decreases the whole performance.

Histogram-Based Outer Join. As data skew is common in most applications, efficient approaches to handle this kind of skew becomes critical for the join performance. Apache Pig has some built-in resistance to skewed joins, a typical method is using histogram [7]. Namely, firstly, a histogram of key popularity is calculated, which can be done with a single MapReduce job. Then, the keys are re-arranged and the jobs are distributed based on that. For instance, if we have the following histogram $k_1 = 19$, $k_2 = 20$, $k_3 = 18$, $k_4 = 60$, and we have two reducers, instead of splitting the keys in a hash-based way (i.e., k_1 and k_3 go to reducer 1, k_2 and k_4 go to reducer 2), the workload will be balanced by sending k_1, k_2 and k_3 to reducer 1 and k_4 to reducer 2. However, this does not work with extreme skew. As shown in [8], if a key is overly popular, the single reducer that it will be sent to will still become a hotspot.

3 Candidate Strategies for MapReduce

In this section, we present some advanced strategies studied in parallel databases and discuss about the possibility to apply them to outer joins in MapReduce.

3.1 The PRPD Method

Xu et al. [9] propose an algorithm named PRPD (Partial Redistribution & Partial Duplication) for inner joins. In their implementation, S is partitioned into two parts: (1) a locally-retained part S_{loc}, which comprises high skew items and which is not involved in the redistribution phase, and (2) the redistributed part S_{redis} which comprises the records with low frequency of occurrence and is redistributed using a common hash-based implementation. The relation R is also divided into two parts: (1) the duplicated part R_{dup}, which contain the keys in S_{loc}, which will be broadcast to all other nodes, and (2) the redistributed part R_{redis} - the remaining part of R that is to be hash redistributed. Then, the final inner join is composed by $R_{redis} \bowtie S_{redis}$ and $R_{dup} \bowtie S_{loc}$.

This method presents an efficient way to process the high skew records (i.e. the ones with keys that are highly repetitive). All these records of S are not transferred at all, instead, a small number of records containing the same keys from R are broadcast. The results for this approach show significant speedup in the presence of data skew. Because PRPD is a hybrid method combining both

the hash and duplication-based join scheme, we can simply use outer joins to replace the corresponding inner joins in the case of MapReduce. Namely, we have

$$R \rtimes S = (R_{redis} \rtimes S_{redis}) \bigcup (R_{dup} \rtimes S_{loc}) \quad (1)$$

However, this implementation could meet the same performance issue as the duplication-based approach described above: the cardinality of the intermediate results of $R_{dup} \rtimes S_{loc}$ could be large, because S_{loc} here is highly skewed, which means that a naive PRPD algorithm cannot be applied to outer joins in MapReduce directly.

3.2 The PRPS Approach

Cheng et al. [10] propose an efficient algorithm for inner joins, named as PRPS (further refined to PRPQ in their work). They use a semijoin-alike way to handle skewed data, inspiring us to apply it to the outer joins of $R_{dup} \rtimes S_{loc}$ in Eq. 1.

In this case, we divide the detailed process into two steps: (1) The unique keys of S_{loc} are extracted and we perform an outer join with R_{dup}; and (2) The matched part of R_{dup} is joined with S_{loc} (inner join), which is union-ed with the non-matching part of R_{dup} to formulate the outputs. Namely,

$$R_{dup} \rtimes S_{loc} = [R_{dup} \rtimes \pi_k(S_{loc})]^{\top} \bowtie S_{loc}] \bigcup [R_{dup} \rtimes \pi_k(S_{loc})]_{\perp} \quad (2)$$

where the symbol $\top$ and $\perp$ means the matched and non-matched results of a outer join respectively.

We can see that this PRPS *outer* join method (referred as PRPS-O in the following) will be efficient on skew handling in MapReduce. The reason is that the large part of skewed records in S is still locally kept and just a small number of unique keys are extracted and transferred, which can be executed with two extra jobs in MapReduce. Nevertheless, as we describe later, we can use a simpler and more efficient method for the outer join implementation.

3.3 Complex Techniques

Other approaches (e.g., [4]) are also very efficient on distributed outer joins. They focus on a fine grained operation of per-node data movement (e.g., peer-to-peer communication based on requirements) to minimizing network communication during implementation. We believe that these algorithms can be coded in MapReduce, however, the number of their execution jobs could be large, more than the PRPS-O method at least. In this case, their implementations could be costly, not only because of their complex data flows, but also the overheads of MapReduce as we described. Actually, in our later evaluation, we have shown that, with two more jobs, PRPS-O takes around 80 s more on runtime, compared to our new method. Thus, we do not consider the detailed implementation and evaluation of these complex techniques in this work.

4 Our Approach

In this section, we present our POPI method and its implementation over Pig [7].

4.1 The POPI Algorithm

The design principles of POPI are: (1) large scale redistribution of skewed records should be limited, so as to avoid load balancing problems; and (2) duplication-based outer join operations should be avoided to the extend possible, in order to simplify the implementation and also reduce possible redundant communication and computation. Based on this, our algorithm adopts the same partitioning approach as PRPD [9]. We process the partitioned records as follows:

$$R \rtimes S = (R_{redis} \rtimes S_{redis}) \bigcup (R_{dup} \bowtie S_{loc}) \tag{3}$$

Namely, the skewed part is executed as an inner join directly. For clarification, we first give a brief proof of the correctness of Eq. 3 here:

Proof sketch: Assume that L is the set of skewed keys of S, then we have that: (1) L is extracted from the skewed part of S, namely, there is $L = \pi_b(S_{loc})$; (2) because the partitioning of R is based on L, namely, a record of R, $< a, x > \in R_{dup}$ if only if the key meets the condition $a \in L$. Namely, every key of R_{dup} appears in L. In this condition, there will be **no** non-matched results in R_{dup} during its outer join execution with S_{loc}. Therefore, the outer join can be represented as an inner join. Note that, even if a skewed key in S does not appear in R, the inner join between R_{dup} and S_{loc} will still be valid here, since the final left outer join results depend on the match conditions of R only. ■

We can see that our outer join implementation is composed by an outer join and an inner join, which is different from a naive transformation, such as that in Eq. 1, in which there are two outer join operations involved. In the meantime, compared to the PRPS-O as Eq. 2, our approach also greatly simplifies the outer join implementations (with two jobs less over Pig for a left outer join). That is also the motivation behind the naming of our approach, POPI (Partial Outer join & Partial Inner join), since the processing between the skewed part and non-skewed part is different from current approaches and allows us to replace an outer join with an inner join.

Inheriting the advantages of the same data partitioning approach as PRPD, we believe that POPI will be robust and efficient on large outer joins in MapReduce. The reason is that we only need to transfer a small part of keys/records (via DFS), rather than the large number of records in S. Moreover, this method will be more efficient than the PRPS-O algorithm, as the number of MapReduce jobs has been reduced.

Following above, with regard to the case of *skewed-skewed* outer joins (i.e., the relation R is also skewed), we partition R into three parts: the R_{dup} and R_{redis} as we described previous, as well as the locally kept part R_{loc}, which contains all the skewed records in R. Correspondingly, records in S is partitioned into three

parts as well, the S_{loc}, S_{redis} and the duplicated part S_{dup}, in which records contains join key belongs to R_{loc}. Then, the final outputs will be composed by three joins: a left outer join for the non-skew part records, namely $R_{redis} ⟕ S_{redis}$, and two inner joins for the skewed records, namely $R_{dup} \bowtie S_{loc}$ and $R_{loc} \bowtie S_{dup}$. In this case, the outer join $R ⟕ S$ can be presented as:

$$(R_{redis} ⟕ S_{redis}) \bigcup (R_{dup} \bowtie S_{loc}) \bigcup (R_{loc} \bowtie S_{dup}) \tag{4}$$

As the uniform-skew join is the core part of a join [9,11], we will focus on such kind of outer joins in our subsequent implementation and evaluation.

4.2 Implementation

We present a general implementation of our method using Pig Latin [12], a language that can be compiled to produce MapReduce programs used with Hadoop. We have three main advantages using this language: (1) It provides a concise notation for algorithms. (2) The outer join methods, such as *hash*, *replicated* and *histogram*, have been integrated in Pig, allowing us for a fair comparison. (3) In a larger pipeline of operations, we can avail of optimisations that are already implemented in Pig, such as performing multiple operation within a job, re-using partitioning of data or executing multiple jobs in parallel.

The detailed implementation of our method in Pig is shown in Algorithm 1. There, R, S, k, t refer to the left table of the outer join, the right side of the outer join, the sampling rate (referred to as *samplingPercentage* later) and the number of chosen top popular keys (refer as *samplingThreshold*) respectively. Initially, we sample the large table S (line 3), group by its join keys (line 4) and count the number of occurrences of sampled key (line 5). Then we order the keys and pick up the most popular keys based on the threshold t (lines 6–7). After that, the tables R and S are partitioned into two parts respectively based on the skewed keys (lines 9–13). With the partitioned data, we then start the outer joins (line 15) and inner joins (line 16). Finally, the outputs of the outer join are composed by the results from both parts (line 18).

5 Experimental Evaluation

5.1 Experiment Setup

Each computation unit of our experimental system has two 8-core Intel Xeon CPU E5-2690 processors running at 2.90 GHz, resulting in a total of 16 cores per physical node. Each node has 32 GB of RAM and a single 128 GB SSD local disk and nodes are connected by Infiniband. The operating system is Linux kernel version 2.6.32-279 and the software stack consists of Hadoop version 1.2.1, Pig version 0.14.0 and Java version 1.7.0_25.

The evaluation is implemented on two relations R and S. We fix the cardinality of R to 64 million records and S to 1 billion records. Because data in

Algorithm 1. POPI Outer Joins

```
1: DEFINE Skew_resistant_outer_join(R,S,k,t)
2:   RETURNS Result {
3: SS = SAMPLE S k;   //sample S
4: SG = GROUP SS BY S::key;
5: SC = FOREACH S2 GENERATE group, COUNT(SS) as c;
6: OrderedKey = ORDER SC BY c DESC;
7: SkewedKey = LIMIT OrderedKey t;
8:
9: SS = JOIN S BY key LEFT, SkewedKey BY group USING 'replicated';
10: SPLIT SS INTO S_loc IF SkewedKey::group is not null, S_red IF Skewed-
    Key::group is null;
11:
12: RS = JOIN R BY key LEFT, SkewedKey BY group USING 'replicated';
13: SPLIT RS INTO R_dup IF SkewedKey::group is not null, R_dis IF Skewed-
    Key::group is null;
14:
15: JA = JOIN R_dis BY R::key LEFT, S_dis by S::key;
16: JB = JOIN S_loc BY S::key, R_dup BY R::key USING 'replicated';
17:
18: $Result = UNION JA, JB; }
```

warehouses is commonly stored following a column-oriented model, we set the data format to <*key, value*> pairs, where both the key and value are 8-byte integers. We assume that R and S meet the foreign key relationship and when S is uniform, the tuples are created in such a way that each of them matches the tuples in the relation R with the same probability. Meanwhile we only add skew to S, following the Zipf distribution. The skew factor is set to 0 for uniform, 1 for the low skew (top ten popular keys appear 14 % of the time) and 1.4 for high skew dataset (top ten appear 68 %). Joins with such characteristics and workloads are common in data warehouses and column-oriented architectures [10].

In all experiments, we set the following system parameters: *map.tasks.maximum* to 16 and *reduce.tasks.maximum* to 8 and the rest of the parameters are left to the default values. The implementation parameters of our method are configured as follows: *samplingPercentage* is set to 10, *samplingThreshold* to 4000 as default. We measure runtime as the elapsed time from job submission to the job being reported as finished.

5.2 Experimental Results

Runtime. We focus on examining the runtime of three algorithms: the hash-based algorithm (referred to *Hash*), histogram-based method (referred to as *Skewed*) and the proposed POPI approaches. Since the first two methods have been integrated in Pig, we just simple use them directly. Though Pig also provides the replicated implementation, we do not compare with it here, since it is limited by the fact that the replicated relation needs to fit in memory [4].

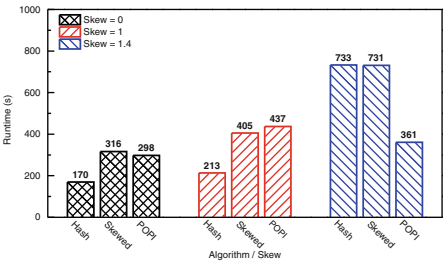

Fig. 1. Runtime of each algorithm.

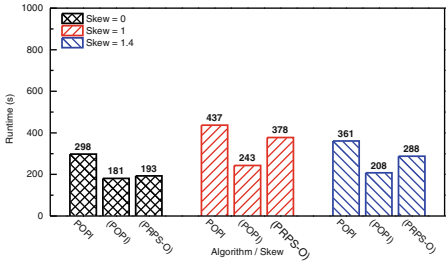

Fig. 2. Compare POPI and PRPS-O.

We implement our tests using over 128 cores (8 nodes) and Fig. 1 shows the runtime of each algorithm. It can be seen that: (1) When S is uniform, Hash is more faster than the other two algorithms. The possible reason is the later two methods have extra-sampling operation and also the overhead of more MapReduce jobs. (2) With low skew, all the runtime increases, which is out of our expectation. As skew handling techniques have been adopted in the later two algorithms, the possible reason could be that not all the highly skewed records were sampled and there remains serious skew in both executions. (3) With high skew, our method becomes the best, which means that, the POPI algorithm can efficiently handle high skew at least.

We also compare the performance of POPI and PRPS-O. The results are shown in Fig. 2. There, the algorithm within "()" means that its sampling operation has been removed. Instead, the top popular keys are stored in a flat file and read as the skewed keys during executions. The reason to do so is for a more precise comparison: the join performance is sensitive to the sampled skew keys and operations like sampling cannot guarantee we always get the skewed keys. In addition, in most data processing pipelines, there is ample opportunity to extract this information as a side-effect of previous jobs. It can be observed that the (POPI) implementation is always faster than the original POPI, which means that the sampling operation could be costly and the sampled skewed keys are also critical for the performance. Moreover, the (POPI) is always faster than (PRPS-O), indicating the more jobs brought by complex implementations in PRPS-O are also costly, about 80 s out of 288 s in a high skewed dataset.

Load Balancing. We also track the detailed time spent on each reducer for each algorithm in the presence of data skew. Under low skew, the results are shown in Fig. 3. It can be seen that there are relatively small discrepancies for all the algorithms. The possible reason is that Hash can not handle data skew and the Skewed and POPI algorithm can not fully catch the skew keys because of their huge number. Furthermore, Fig. 4 shows the results in the condition of high skew. There, Hash is not balanced at all, in comparison, the Skewed method and POPI are much better. We should highlight that POPI achieves excellent load-balancing here. The reason could be that the number of skewed

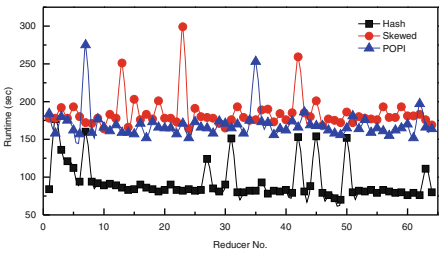

Fig. 3. Runtime of reducer in Skew = 1.

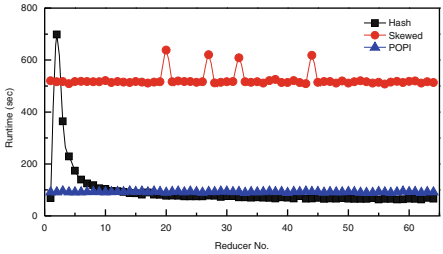

Fig. 4. Runtime of reducer in Skew = 1.4.

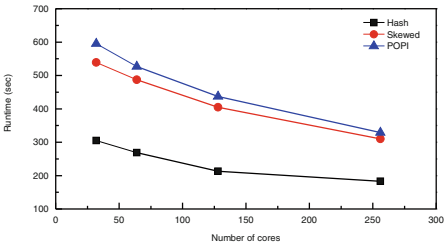

Fig. 5. Scalablity in Skew = 1.

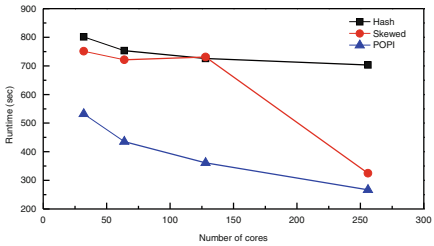

Fig. 6. Scalablity in Skew = 1.4.

key is relative small in the condition of high skew and most of the popular keys are extracted, even when we only sample a small part of the input. Moreover, our runtime is much smaller than the Skewed method, which demonstrates the efficiency of this new approach.

Scalability. We finally test the scalability of our algorithm by varying the number of processing cores. We implement our test on the system from 2 nodes (32 cores) to 16 nodes (256 cores) over the skewed datasets. The detailed time-cost is shown in Figs. 5 and 6. We can see that the Skewed method and our algorithms generally scales well with the number of cores under low skew. However, they are slower than Hash. The reason could be the overhead of their implementations on MapReduce since Hash has only a single mapreduce job. This result is greatly different from the conditions when using other programming languages (e.g., X10 in [4]), where the Hash method is slower. In such scenarios, we believe that the hash-based approach could still be a better choice for MapReduce, under low skew. In comparison, with high skew, our method scales well while the other two are not. More importantly, our approach is significantly faster. Combining this with the good load balancing we have illustrated in Fig. 4, it can be seen that our method could be more suitable for the large outer joins in the presence of high skew.

6 Related Work

Several approaches have been proposed to improve the performance of joins over MapReduce [13], regardless, they have modified the basic MapReduce framework and cannot be readily used by existing platforms like Hadoop. Though the work [5] presents an extensive implementation on joins in MapReduce, they focus on execution profiling and performance evaluation, but not for robust join algorithms in the presence of big data.

As data skew has significant impact on distributed join processing, there has been in-depth research on skew handling in parallel and distributed DBMSs [1,3,4,9,10]. However, as we have explained, their methods could either have performance issues or be complex in MapReduce implementations. In comparison, our POPI algorithm is simple on implementation and also shown to be efficient.

Many algorithms have been introduced on skew handling for joins over MapReduce [14], regardless, most of them focus on inner joins, as opposed to consider the challenges on the complexity of outer join implementations. Moreover, several efforts in designing high level query languages on MapReduce, such as Pig [7] and Hive [15], have employed advanced mechanisms on skew handling in outer joins, however, as we have described, sometimes they could be not very efficient. Additionally, though some platforms (e.g., Stratosphere [16]) have provided efficient techniques on big data analytics, they focus on creating optimized plans of executing jobs, in contrast to the detailed implementation of a single operation as we studied in this work.

Recently, Bruno et al. [17] present three *SkewJoin* transformations to mitigate the impact of data skew in a distributed join operation. To prevent an outer join operator from generating *null* values, they partition the skewed tuples (e.g., in S) in a round-robin way so that each node can see at least one such tuple. In comparison, our approach is more light-weighted, since we do not need to repartition the skewed tuples, the number of which is always huge. Even when that some skewed tuples do not appear on some nodes, we will not generate *null*, as we use an inner join operation for the skewed tuples in our approach.

The PRPD algorithm [9] is a very popular method adopted by many companies (e.g., Teradata [9], Microsoft [17] and Oracle [18]). Nevertheless, as we have analyzed, PRPD cannot be applied to outer joins directly. The underlying data partitioning of our method is the same as PRPD, both are based on the skewed keys, therefore, the statistical information of data skew that is collected by the current systems using PRPD can be applied to POPI directly. This means that POPI can be used to extend the join implementations of current systems (or over current platforms like MapReduce [6] and Spark [19]) and consequently simplify the general executions of data queries. For example, skew statistics on the join keys (a, b) for the inner join implementation $R(a, x) \bowtie S(b, y)$ can be applied to the implementation of $R(a, x) \rtimes S(b, y)$ directly, without any modifications for the underlying join patterns.

7 Conclusions

In this paper, we focus on one data-intensive operation - outer joins - over the MapReduce platform. We have described current applied techniques and discussed the potential performance issues in the condition of using current advanced methods from parallel databases. Based on that, we propose our POPI algorithm for efficient large-scale data outer joins over MapReduce. We describe the detailed design and present the evaluation over a Hadoop cluster and Pig. We show that our new method is simple to implement. In the meantime, the experiment results also show that POPI is scalable, robust and can perform better compared with current implementations, at least in the case of high skew.

Acknowledgments. This work is supported by the German Research Foundation (DFG) within the Collaborative Research Center SFB 912 (HAEC) and in Emmy Noether grant KR 4381/1-1 (DIAMOND).

References

1. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992)
2. Li, F., Ooi, B.C., Özsu, M.T., Wu, S.: Distributed data management using MapReduce. ACM Comput. Surv. **46**(3), 31 (2014)
3. Xu, Y., Kostamaa, P.: A new algorithm for small-large table outer joins in parallel DBMS. In: ICDE, pp. 1018–1024 (2010)
4. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Robust and efficient large-large table outer joins on distributed infrastructures. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 258–269. Springer, Heidelberg (2014)
5. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., et al.: A comparison of join algorithms for log processing in Map Reduce. In: SIGMOD, pp. 975–986 (2010)
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
7. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a high-level dataflow system on top of Map-Reduce: the Pig experience. PVLDB **2**(2), 1414–1425 (2009)
8. Kotoulas, S., Urbani, J., Boncz, P., Mika, P.: Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on pig. In: Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 247–262. Springer, Heidelberg (2012)
9. Xu, Y., Kostamaa, P., Zhou, X., Chen, L.: Handling data skew in parallel joins in shared-nothing systems. In: SIGMOD, pp. 1043–1052 (2008)
10. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Robust and skew-resistant parallel joins in shared-nothing systems. In: CIKM, pp. 1399–1408 (2014)
11. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: SIGMOD, pp. 37–48 (2011)
12. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD, pp. 1099–1110 (2008)

13. Jiang, D., Tung, A., Chen, G.: Map-Join-Reduce: toward scalable and efficient data analysis on large clusters. TKDE **23**(9), 1299–1311 (2011)
14. Liao, W., Wang, T., Li, H., Yang, D., Qiu, Z., Lei, K.: An adaptive skew insensitive join algorithm for large scale data analytics. In: Chen, L., Jia, Y., Sellis, T., Liu, G. (eds.) APWeb 2014. LNCS, vol. 8709, pp. 494–502. Springer, Heidelberg (2014)
15. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a Map-Reduce framework. PVLDB **2**(2), 1626–1629 (2009)
16. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., et al.: The stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014)
17. Bruno, N., Kwon, Y., Wu, M.C.: Advanced join strategies for large-scale distributed computation. PVLDB **7**(13), 1484–1495 (2014)
18. Bellamkonda, S., Li, H.G., Jagtap, U., Zhu, Y., Liang, V., Cruanes, T.: Adaptive and big data scale parallel execution in Oracle. PVLDB **6**(11), 1102–1113 (2013)
19. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28 (2012)