Load-balancing Distributed Outer Joins through Operator Decomposition

Long Cheng[a,*], Spyros Kotoulas[b], Qingzhi Liu[c], Ying Wang[d]

[a] *School of Computer Science, University College Dublin, Ireland*
[b] *IBM Research, Dublin, Ireland*
[c] *Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands*
[d] *ICT, Chinese Academy of Sciences, Beijing, China*

Abstract

High-performance data analytics largely relies on being able to efficiently execute various distributed data operators such as distributed joins. So far, large amounts of join methods have been proposed and evaluated in parallel and distributed environments. However, most of them focus on inner joins, and there is little published work providing the detailed implementations and analysis of outer joins. In this work, we present POPI (Partial Outer join & Partial Inner join), a novel method to load-balance large parallel outer joins by decomposing them into two operations: a large outer join over data that does not present significant skew in the input and an inner join over data presenting significant skew. We present the detailed implementation of our approach and show that POPI is implementable over a variety of architectures and underlying join implementations. Moreover, our experimental evaluation over a distributed memory platform also demonstrates that the proposed method is able to improve outer join performance under varying data skew and present excellent load-balancing properties, compared to current approaches.

Keywords: Distributed join, parallel join, outer join, data skew, load balancing, in-memory computing, Spark

1. Introduction

An increasing number of companies rely on the results of big data analytics to improve their operations, planning, customer service and risk management. For example, in a survey of 476 executives around the world, more than half say that they have made existing services and products more profitable from their data [1]. Although hundreds of large data centers have been built across the globe to support high-performance data analytics, efficient execution of analysis jobs is still very challenging. For example, China Mobile gathers 58TB of phone call records per day [2], just formatting that much data within a specified schema in a timely manner is already a challenge.

As one of the core tasks in such scenarios, distributed joins, which always incur significant costs in communication and computation, have received notable attention from various domains. These areas include but are not limited to data management [3] and high-performance computing [4]. Over the past years, inner join algorithms in parallel and distributed environments have been widely studied [3, 5, 6]. However, there has been relatively little work done on outer joins, especially on their detailed implementations. In fact, outer joins are common in complex queries and widely used in various applications [7]. For instance, it is common to perform left outer joins between customer ids and transaction ids for analyzing purchase patterns in e-commerce [8]. Compared to inner joins, an outer join does not discard records from one (or both) table(s) that do not match with any tuple in the other table. For example, for a left outer join $R(a, x) \leftouterjoin S(b, y)$, in which a and b are the join keys and x and y are the payloads, the output results contain not only the matched tuples in the form of $\langle a, x, y\rangle$, but also the non-matched ones $\langle a, x, \omega\rangle$[1] if there is no matched key for a in S.

Currently, there are two mainstream algorithms for distributed outer join implementations [8]: ROJA (*redistribution outer join algorithm*) and DOJA (*duplication outer join algorithm*). As we will explain later, these two methods suffer from performance issues when data skew is encountered: the former method has load-balancing problems while the latter one induces significant network communication. Since data skew occurs naturally in various applications [5, 9] and join performance is always challenged by large-scale datasets, it is important for practical approaches to perform efficiently in such contexts.

To achieve good load-balancing in the presence of diverse workloads, and consequently to improve the scale-out ability of big data applications, many algorithms have been designed for skew handling for inner joins [5, 6]. However, there is only limited research on the detailed analysis and implementation for distributed outer joins. Some approaches transform the expression of an outer join into multiple (inner/anti) joins, and thus allow us to reuse existing inner join techniques. This could not impact the join performance on a standalone machine. However, because of system overheads (e.g., for additional jobs) and redundant data transferring over networks, rewriting could lead

[*] Corresponding author.
Email addresses: long.cheng@ucd.ie (Long Cheng), spyros.kotoulas@ie.ibm.com (Spyros Kotoulas), q.liu.1@tue.nl (Qingzhi Liu), wangying2009@ict.ac.cn (Ying Wang)

[1] We use ω as a *null* value in outer joins throughout this paper.

to bad performance in a distributed enviroment [10]. Up to the present, several methods have been proposed to optimize distributed outer join implementations from a data-centric angle (e.g., [8]), and they have been shown to be able to outperform the conventional approaches. Nevertheless, these approaches are designed for small-large table outer joins rather than the large-large ones, which are actually common in this Big Data era [11].

In this paper, we present a novel outer join decomposition method called POPI. It is designed for skew-resistant outer joins over distributed environments. On a reference implementation, POPI can outperform current implementations in large-scale data processing scenarios. We list the contribution of this work as follows:

- We propose a novel approach called POPI (Partial Outer join & Partial Inner join) for load-balancing parallel outer joins over distributed systems.
- The POPI join geography does not require major changes to the current implementations of a shared-nothing architecture. Moreover, its underlying implementation is still based on the widely-used redistribution and duplication operations and can thus be applied to current systems/applications directly.
- Our experimental results demonstrate that the proposed POPI method is robust in the presence of various data skew and can efficiently process large table outer joins in a distributed environment.

This manuscript is an extension of our previous work [11]. We have enhanced the core parts of the POPI approach in three aspects, including the theoretical analysis, the applicability analysis and the experimental evaluation. We theoretically prove that the high-level decomposition and parallelism of POPI are correct and also show that our approach can achieve optimal load balancing with less network traffic in the presence of large skew datasets, compared to current approaches. Moreover, we present a detailed discussion on the applicability of our approach, showing that our approach is applicable to many computing systems and is agnostic to the implementation of underlying joins. Therefore, POPI can be applied as a new outer join pattern in conjunction with many current (mainstream) join implementations in current data systems. In terms of experiments, we have conducted a new and also more comprehensive evaluation of the proposed approach in a more challenging environment, i.e., a distributed in-memory computing environment (i.e., Spark [12]) rather than a disk-based platform such as MapReduce [13] as we have described in [11]. This allows us to focus on the performance issues arising from the approach itself, rather than its implementation complexity, i.e., we are not influenced by the overheads from I/O and different numbers of jobs. We believe that all these extensions are necessary and important. They make the proposed POPI more convincing and self-contained, and thus push it as the state-of-the-art distributed outer join approach to be be applied in current data systems.

The rest of this paper is organized as follows: In Section 2, we analyze current outer join implementations. In Section 3, we discuss some advanced skew handling strategies for large data outer joins. In Section 4, we introduce our POPI approach. In Section 5, we present our experimental results. We report on related work in Section 6 while we conclude the paper in Section 7.

2. Problem Description

We focus on the left outer join ($⟕$) in this work, since it is the most common outer joins used in data applications. Specifically, we focus on the joins between two input relations R and S. Without loss of generality, we assume that tuples in both relations are $\langle k, v \rangle$ pairs, where k is the join attribute (or key), and the size of the relation R is smaller than S. A typical parallel join can be decomposed into a data redistribution stage followed by a local join [5]. As the latter has been extensively studied, we will focus on the redistribution process. In this section, we mainly discuss the possible performance issues of current outer join approaches.

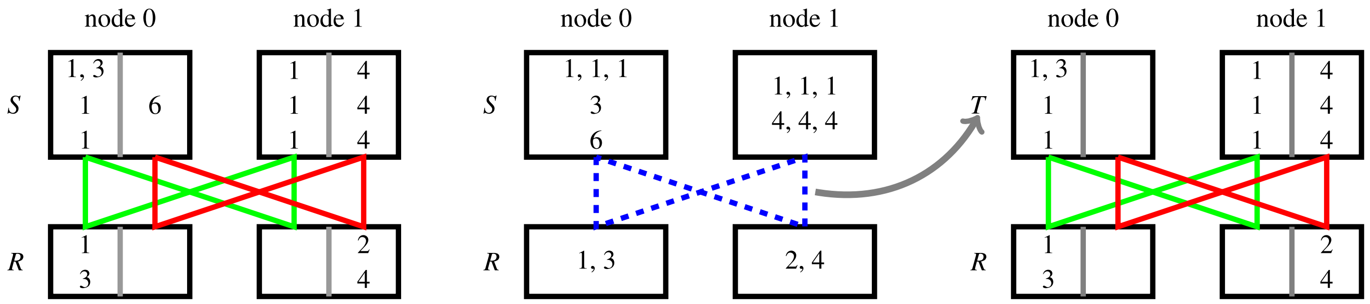
2.1. Two Conventional Approaches

We explain the implementations of the two conventional outer joins algorithms in detail based on the example demonstrated in Figure 1. There, there are 15 input tuples, which are initially distributed over two computing nodes. We only show join keys without the values, since the values are not relevant for the application of the methods described in this paper.

2.1.1. ROJA

The implementation of ROJA (*redistribution-based outer join algorithm*) is very similar to the case for inner joins, and its join pattern is demonstrated in Figure 1(a). There, the input tuples of R and S at each node are firstly redistributed over the system based on the hash values of their join keys. If we just use a simple hash function like $h(k) = k \, mod \, 2$ here, then the tuples with key $\{1, 3\}$ in R and S will be transferred to node 1 and the rest are going to node 0. After this redistribution, local outer joins between received tuples on each node can be executed in parallel. Namely, the green/light color join will commence on node 1, and the red/dark one will be on node 0. Obviously, the number of final output results is 11, including 1 non-matched result (with the key 2) and 10 matched ones (with the keys 1, 3 and 4).

Similar to a redistribution-based inner join algorithm, ROJA can achieve good performance under ideal balancing conditions. However, when the join keys present significant skew, large amounts of tuples would be transferred to some specified computing nodes, and this will result in computation and network hotpots. For example, if the tuples with key 1 in Figure 1(a) appears 1 million times on both the nodes, then there will be 1 million tuples moving from node 0 to 1 while there are only 5 tuples from node 1 to 0 (the ones with key 2 and 4). Such poor load balancing impairs system scalability because employing new nodes will not bring in obvious performance improvement [10].



(a) redistribution-based approach

(b) duplication-based approach

Figure 1: The join patterns of the two conventional outer join methods over two computing nodes. The solid lines in ⋈ format are outer joins and the dashed one are inner joins.

2.1.2. DOJA

Compared to ROJA, DOJA (*duplication-based outer join algorithm*) shows significantly different implementations. As demonstrated in Figure 1(b), its main processing stage contains two sequential phases. Firstly, there is an *inner* join between R and S to formulate the intermediate result T, which is implemented by duplicating all the tuples of R on each node to all other nodes. As we can see in Figure 1(b), T contains 10 tuples (with the key 1, 3 and 4). The second phase is a left outer join between R and T using ROJA. Namely, the tuples with key $\{1, 3\}$ in R and T will be transferred to node 1 and the rest are going to node 0. In this case, we still get the 11 results.

Duplication is an effective way to remove hot spots resulting from join key skews for inner joins. Therefore, the first phase of DOJA can achieve good load-balancing. However, DOJA will still meet performance issues when S is skewed. This is because the number of intermediate results T could be very large. Moreover, T can also incur attribute value skew. Both factors will make the join execution in the second phase very costly.

In fact, DOJA is seldom used in data applications. From its implementation, the method will be only suitable for the case that the join selective is very low, R is very small and S is skew. In this condition, the intermediate results will be small and DOJA could outperform the ROJA algorithm. It should be noted that the join in the first phase of DOJA is an inner join. The reason is that using an outer join would bring either redundant or erroneous non-matched results [10]. To avoid this problem, DOJA redistributes the intermediate inner join results and then outer joins with R to obtain the final output.

2.2. Outer Join Optimization - DER

Xu et al. [8] propose an efficient algorithm called DER (*duplication and efficient redistribution*), which is the state-of-the-art method for optimization of outer join implementations in a distributed environment. The algorithm is composed of two phases. In the first phase, all the tuples of R at each node are duplicated to all other nodes. Then, a local left outer joins between the received tuples of R and S is performed in parallel at each node. Different from a conventional local outer join implementation, the *ids* of all **non-matched** rows of R are recorded as the intermediate result T' at this step. In the second phase, the ids in T' are redistributed according to their hash values and the non-match join results at each node are organized on this basis: the received ids at each node are counted, if the number of times an id appears is equal to the number of computing nodes, then the record in R with this row-id will be extracted to formulate the non-matched results. The final output is the union of the matched results in the first phase and the non-matched results in the second phase.

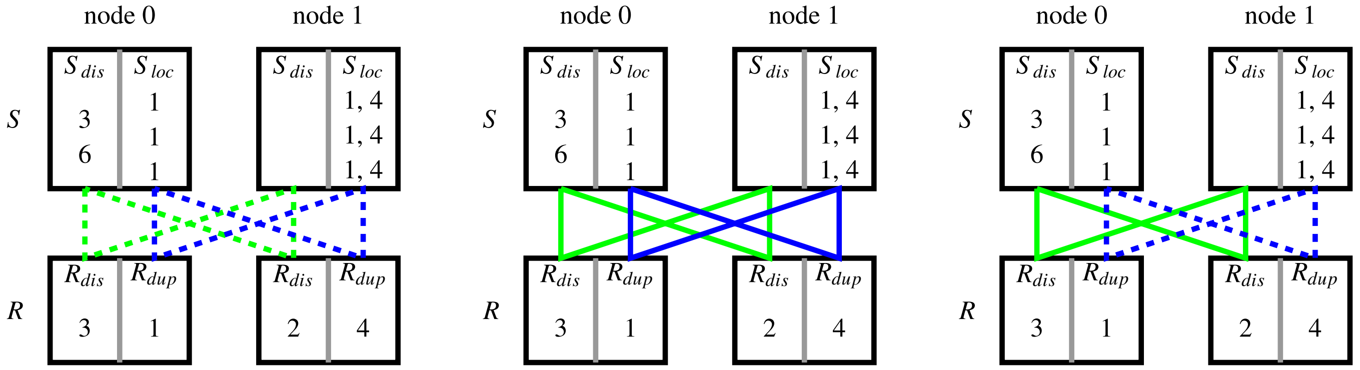
Following the example in Figure 1, assuming that the row-id of each tuple in R is equal to the value of its key, then, after the first phase, we have that the matched results are the tuples with key 1, 3 and 4, and the intermediate result T' contains four ids, i.e., {2, 4} on node 0 and {2, 3} on node 1. By redistributing T', the id 2 appears two times node 0, which is equal to the number of nodes. Therefore, there will be a non-matched result for the tuple with key 2. In contrast, there is no non-matched result from the key 3 and 4, because their responsible ids only appear one time respectively.

Altough DER follows a duplication-based way, compared to DOJA, it uses a more advanced approach, i.e., a customized left outer join to identify non-matched results. This will generate a intermediate result. However, the size of T' will be much more smaller then the T in DOJA. The reason is that T' only records the ids of all non-matched rows of R, the number of which is not greater than the number of tuples in R. This makes DER be able to greatly reduce network communication, compared to DOJA. As presented in [8], DER is actually considered a replacement for DOJA.

Although the work [8] does not focus on the skew handling problem, it can be predicted that DER will be very effective in such aspect. The reason is that DER only redistributes the non-matched ids of R, which is not affected by the skewness of S. Moreover, when R is small, the redistributed data in DER will be very small in size, and redistribute such a small data set will not bring in notable load-imblance even when it is skewed. As DER has to duplicate all tuples of R, the method is designed to work best for small-large outer joins rather than large-large ones.

3. Strategies for Distributed Outer Joins

In this section, we present some advanced skew handling strategies in parallel joins and discuss about the possibility to



(a) PRPD inner joins

(b) PRPS outer joins

(c) POPI outer joins

Figure 2: The join patterns of the different advanced inner and outer join methods over two nodes. The colored solid lines in $\bowtie$ format are outer joins and dashed are inner joins.

apply them to outer join implementations in large data analytics platforms such as Spark.

3.1. The PRPD Method

Xu et al. [6] propose an algorithm named PRPD (*partial redistribution & partial duplication*) for inner joins. For the input tuples in Figure 1, assume that the skew keys are 1 and 4, the detailed data partitioning and join pattern of PRPD is demonstrated in Figure 2(a). There, S is partitioned into two parts: (1) S_{loc}, which comprises skew items; and (2) S_{redis}, which comprises the tuples with low frequency of occurrence. Meanwhile, R is also partitioned into two parts: (1) R_{dup}, which contain the keys in S_{loc}; and (2) R_{redis}, the remaining part of R. For example, the tuples with key 1 in S on node 0 belong to S_{loc} and the tuples with key 4 in R on node 1 belong to R_{dup}. After this partitioning, the inner join between R and S can then be represented as:

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

In detail, the tuples in S_{loc} are kept locally, the tuples in R_{dup} are duplicated to all other nodes, and the rest ones in R_{redis} and S_{redis} are redistributed using a common redistribution-based approach. This design presents an efficient way to process the high skew tuples (i.e., the ones with keys that are highly repetitive) - all such tuples of S are not transferred at all, instead, a small number of tuples containing the same keys from R are duplicated. Therefore, the network communication cost can be greatly reduced and also the hotspots caused by skewed tuples can be avoided.

As PRPD combines both the redistribution and duplication-based join scheme, one possible way for a distributed outer join implementation is that we use outer joins to replace the corresponding inner joins in PRPD. Namely,

$$R \text{⟕} S = (R_{redis} \text{⟕} S_{redis}) \bigcup (R_{dup} \text{⟕} S_{loc}) \tag{1}$$

Following our theoretical analysis in the later Section 4.2, we will see that Eq. 1 is correct. However, this implementation could meet the same performance issue as DOJA: the cardinality of the intermediate results of $R_{dup} \text{⟕} S_{loc}$ could be large. This means that a PRPD-based algorithm is actually not suitable for distributed outer joins. A solution for this is to use DER to process $R_{dup} \text{⟕} S_{loc}$. However, as we will show later in Section 4.1, our proposed method uses a more simple and effective way to solve the problem.

3.2. The PRPS Strategy

Cheng et al. [5] propose an efficient algorithm for inner joins called PRPS. They use a semijoin-alike way to handle data skew, inspiring us to apply it to outer joins. As illustrated in Figure 2(b), the data partitioning of PRPS is the same as PRPD, and its outer join pattern follows Eq. 1. However, unlike a ROJA execution for the skewed tuples, PRPS processes the outer join as following: the unique keys in S_{loc} are extracted to perform an outer join with R_{dup}, and then the matched part of R_{dup} is joined with S_{loc} (inner join), which is unioned with the non-matching part of R_{dup} to formulate the outputs.

For example, for the case in Figure 2(b), the unique keys of S_{loc}, (1 and 4) will outer join with R_{dup}. Then, the matched tuples will be duplicated to be part of an inner join with S_{loc} while the non-matched part will be output directly as a part of final results. Obviously, this PRPS-based *outer* join method (referred to as PRPS-O) will be effective on skew handling, as the skew tuples in S are locally kept and just a small number of unique keys are extracted and transferred. However, as we will explain later, we can use a more straightforward way for the outer join implementation.

3.3. Other Techniques

Track [14] employs of a very fine-grained multi-phase scheduling algorithm to explore data locality for distributed joins and thus it can minimize their network traffic. Although the technique can be used to outer joins, its core data operation (i.e., *select broadcast and migration*) can not be applied to some mainstream data processing platforms. Take the Spark as an example, neither broadcasting tuples to some specified partitions nor migrating tuples between partitions of a RDD will be possible.

Some other algoritihms (e.g., [15]) are shown to be efficient on distributed outer joins. However, their executions will be even more complex than PRPS-O. This is because they focus on a fine-grained control of per-node data movement (e.g., peer-to-peer communication based on the requirements). Moreover, all these techniques (including PRPS-O) do not follow a conventional redistribution and duplication communication pattern, and thus applying them in data analytics applications directly would impact the design of underlying systems (such as query optimization) or implementation choices (such as communication and scheduling [11]). In such a case, we do not consider the detailed implementation and evaluation of these techniques in this work. In comparison, we will propose a new approach, which can be directly applied to current data systems/solutions. Additionally, although the recent DDR algorithm [16] has been shown to be able to achieve comparable performance as DER in cloud computing environments, it just provides an engineering redesign for DER and is only suitable to small-large outer joins. The detailed performance comparison between DER and DDR has been studied in [16], thus we will only compare our proposed approach with DER in the following sections.

4. Our Approach

In this section, we present the details of the proposed POPI approach and analyze how it can address load balancing issues in distributed outer joins. Moreover, we compare it with current approaches and present its detailed implementation over the Spark platform [12].

4.1. The POPI Approach

The basic design principles of POPI are: (1) large-scale redistribution of skewed tuples should be limited, so as to avoid load balancing problems; and (2) duplication-based outer join operations should be avoided to the extent possible, in order to simplify the implementation and also reduce possible redundant communication and computation.

For simplicity, let us consider a uniform-skew condition, i.e., the join keys in S have skewed values and R is not skewed. Assuming that we know the set of skewed keys L in S (detailed discussion on skew selection is give in Section 4.4), then the implementation of POPI can be summarized as the following two phases:

- *Phase 1.*
  - On each computation node i, the sub-relation S_i is partitioned into two parts based on L: (i) a locally-retained part S_{loc}, which comprises all the tuples with popular keys (i.e., key with values in L). This part is not involved in the redistribution operation; and (ii) the remaining part S_{redis}, for which the tuples are hash-redistributed as in the ROJA implementation.
  - Similarly, the relation R at each node is also divided into two parts: (i) the duplicated part R_{dup}, comprising the tuples with key values in L. This part is broadcast to all other computation nodes; and (ii) the remaining part R_{redis}, the tuples that are hash-redistributed.
- *Phase 2.* After the duplication and the redistribution operations, the following two joins are implemented at each computation node in parallel: (i) an outer join between redistributed part of R and S: $R_{redis} \mathbin{⟕} S_{redis}$; and (ii) an inner join between the duplicated part of R and locally kept S: $R_{dup} \bowtie S_{loc}$. The final output is the union of these two joins on all the nodes.

Based on the description above, the outer join between the two relations R and S in our approach can be represented as:

$$R \mathbin{⟕} S = (R_{redis} \mathbin{⟕} S_{redis}) \bigcup (R_{dup} \bowtie S_{loc}) \tag{2}$$

An example of such implementation is illustrated in Figure 2(c). We can see that the data partitioning and the join pattern of POPI are the same as the PPRD and PRPS, but their detailed join executions are different. Namely, there is an inner join and an outer join in POPI rather than two inner joins or two outer joins. Here, we highlight the two key characteristics of the proposed POPI as follows:

1. At a high level, an outer join is composed of an inner join and an outer join implementation. This is different from a naive transformation, such as $R \mathbin{⟕} S = (R_{redis} \mathbin{⟕} S_{redis}) \cup (R_{dup} \mathbin{⟕} S_{loc})$, which involves two outer join operations.
2. At a low level, the two input relations are partitioned into two parts and each partition participates in a join implementation independently. This is different from some traditional transformations such as $R \mathbin{⟕} S = (R \bowtie S) \cup ((R \rhd S) \times \{\omega\})$, in which all the tuples fully participate in the two distributed joins respectively - an inner join and an anti-join.

That is also the motivation behind the naming of our approach, POPI (Partial Outer join & Partial Inner join). Distinguishing between the skewed part and non-skewed part is different from the outer join implementations described in Section 2 and allows us to replace an outer join with an inner join. Although our partitioning strategy is the same as PRPD, their targets and executions are totally different, i.e., we focus on developing an efficient approach for distributed outer joins while PRPD can not be applied to outer joins directly as we have described.

4.2. Correctness of POPI

The implementation of an outer join is more challenging than an inner join in distributed systems, since a careless design will bring in redundant and erroneous non-matched results. To show that POPI can indeed provide correct outputs for outer joins, we prove the correctness of the approach in two aspects. We first give a proof for the correctness of our operator decomposition and then we prove that our parallel implementation in a distributed environment is correct. For simplicity, we focus on the uniform-skew outer joins first, then we discuss about the skew-skew condition.

4.2.1. *Correctness of Operator Decomposition*

Eq. 2 shows that an outer join operator in POPI can be decomposed into two data operators: an outer join and an inner join. Here, we prove that such a decomposition is correct. Assume that there are n computing nodes, and the tuples in the relations R and S on each node i have been partitioned based on the skewed join keys L. As both R and S are split into two disjoint sets, we have:

$$\begin{cases} R = R_{redis} \cup R_{dup}, \;\; S = S_{redis} \cup S_{loc}, \\ R_{redis} \cap R_{dup} = \emptyset, \;\; S_{redis} \cap S_{loc} = \emptyset \end{cases} \tag{3}$$

Since the tuples in R and S are split by their join keys, we have that the following two inner joins will produce the empty set:

$$R_{redis} \bowtie S_{loc} = \emptyset, \;\; R_{dup} \bowtie S_{redis} = \emptyset \tag{4}$$

Based on Eq. 3, there is:

$$\begin{aligned} R ⟕ S &= (R_{redis} \cup R_{dup}) ⟕ (S_{redis} \cup S_{loc}) \\ &= (R_{redis} ⟕ (S_{redis} \cup S_{loc})) \\ &\quad \bigcup (R_{dup} ⟕ (S_{redis} \cup S_{loc})) \end{aligned}$$

In the outer join $R_{redis} ⟕ (S_{redis} \cup S_{loc})$, for any tuple $\langle a, x \rangle \in R_{redis}$, its output will be either $\langle a, x, y \rangle$ for a match case or $\langle a, x, \omega \rangle$ for a non-match one. From Eq. 4, the tuple will never have a match for $R_{redis} ⟕ S_{loc}$. This means that the output tuples in the outer join actually depend on the outer join between R_{redis} and S_{redis}, i.e., we have $R_{redis} ⟕ (S_{redis} \cup S_{loc}) = R_{redis} ⟕ S_{redis}$.

Similarly, for the outer join $R_{dup} ⟕ (S_{redis} \cup S_{loc})$, the outputs will only depend on outer join between R_{dup} and S_{loc}, because a tuple in R_{dup} does not have any matched tuples in S_{redis} from Eq. 4. As L is the set of skewed keys of S, thus 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 tuples in R is based on the keys in L, namely, a tuple of R, $\langle a, x \rangle \in R_{dup}$ only if the key meets the condition $a \in L$. Namely, every key of the tuples in 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, i.e., $R_{dup} ⟕ (S_{redis} \cup S_{loc}) = R_{dup} \bowtie S_{loc}$. 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. Then, we have Eq. 2 for POPI. ■

4.2.2. *Correctness of Parallel Implementation*

In fact, the above proof only shows that POPI is correct from a logical (or task) perspective. If we consider the operator decomposition from an angle of parallelism, we can only conclude that the *task-parallel computations* of POPI are correct. Namely, a job (outer join) can be divided into two tasks (joins), and these two tasks can be computed in parallel (i.e., independent). Here, on the basis of the decomposition, we prove that actually the *data-parallel computations* of POPI are also correct.

A superscript is used to indicate the tuples for a give subrelation on a specified node. For example, R^1_{redis} means that tuples in R_{redis} on node 1. Then, we have:

$$\begin{cases} R_{redis} = \cup_{i=1}^{n} R^i_{redis}, \;\; R_{dup} = R^i_{dup} \; \forall i \in [1, n] \\ S_{redis} = \cup_{i=1}^{n} S^i_{redis}, \;\; S_{loc} = \cup_{i=1}^{n} S^i_{loc} \end{cases} \tag{5}$$

From Eq. 2 and Eq. 5, there is

$$\begin{aligned} R ⟕ S &= (\cup_{i=1}^{n} R^i_{redis} ⟕ \cup_{i=1}^{n} S^i_{redis}) \\ &\quad \bigcup (R_{dup} \bowtie \cup_{i=1}^{n} S^i_{loc}) \\ &= (\cup_{i=1}^{n} (R^i_{redis} ⟕ \cup_{j=1}^{n} S^j_{redis})) \\ &\quad \bigcup (\cup_{i=1}^{n} (R^i_{dup} \bowtie S^i_{loc})) \end{aligned}$$

The tuples in R_{redis} and S_{redis} are redistributed based on their join keys, a tuple in R_{redis} matches a tuple in S_{redis} if and only if the two tuples are located on the same computing node after the redistribution. This means that $R^i_{redis} \bowtie S^j_{loc} = \emptyset, \forall i \neq j$. Similar as our above analysis, the outputs of the outer join $R^i_{redis} ⟕ \cup_{j=1}^{n} S^j_{redis}$ depend on the outer join between R^i_{redis} and S^i_{redis}, i.e., there is:

$$\begin{aligned} R ⟕ S &= (\cup_{i=1}^{n} (R^i_{redis} ⟕ S^i_{redis})) \\ &\quad \bigcup (\cup_{i=1}^{n} (R^i_{dup} \bowtie S^i_{loc})) \\ &= \cup_{i=1}^{n} ((R^i_{redis} ⟕ S^i_{redis}) \bigcup (R^i_{dup} \bowtie S^i_{loc})) \end{aligned}$$

It should be noticed that here the $R^i_{dup} \bowtie S^i_{loc}$ is a duplication-based join, because the R^i_{dup} on each node i is actually the R_{dup} (because of duplication) as presented in Eq. 5. From the equation, it can be seen that the data on each node can be computed in parallel, and the two joins (an inner and outer join) on each node can be also executed in parallel if possible. Therefore, our outer join can be implemented in a distributed system in parallel. ■

4.2.3. *Skew-skew Outer Joins using POPI*

We focus on the uniform-skew condition above. Regarding to the case of *skew-skew* outer joins (R is also skewed), similar to PRPD [6], we partition R into three parts: the R_{dup} and R_{redis} as we have described previous, as well as the locally kept part R_{loc}, which contains all the skewed tuples in R. Correspondingly, tuples in S are partitioned into three parts as well, the S_{loc}, S_{redis} and the duplicated part S_{dup}, in which tuples contain join keys belonging to R_{loc}. Then, the final outputs will be composed of three joins: a left outer join for the non-skew tuples, namely $R_{redis} ⟕ S_{redis}$, and two inner joins for the skewed tuples, namely $R_{dup} \bowtie S_{loc}$ and $R_{loc} \bowtie S_{dup}$. In this case, the outer join $R ⟕ S$ can be presented as the following statement, which can be also proved in a similar way as the uniform-skew condition.

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

It can be seen that data partitioning is a very important operation in our approach. Therefore, how to identify the skewed keys in both input relations is critical for our outer join implementation. We will discuss this problem later when we analyze the *applicability* of our method. Moreover, because the uniform-skew join is the core part of a join [6] [17], we will focus on such kind of outer joins in our following analysis and also the evaluation in Section 5.

4.3. Advantages of POPI

In this subsection, we theoretically analyze why POPI can achieve good load-balancing and reduce network traffic for distributed outer joins in the presence of data skew. Moreover, we also discuss that POPI is simpler to implement by comparing with current advanced techniques.

4.3.1. Load Balancing

The proposed POPI approach can efficiently address the data skew problem in two aspects: (1) unlike ROJA, POPI does not transfer any skewed tuples in the redistribution process, thus hotspots induced by data redistribution can be avoided (improving load-balancing); (2) we only broadcast part of the tuples to join with the locally kept tuples, and this duplication operation does not bring in any hot spots resulting from join key skews.

To investigate the load-balancing of POPI in detail, we focus on tracking the number of transferred/received tuples in an outer join and compare it with the most commonly used ROJA. The reason is that the metric gives the insight into the workloads and network communication, i.e., the larger the number of joined tuples a node receives, the greater its associated workload will be. We call a relation is skew when its join keys are unevenly distributed in the relation, i.e., some keys appear more frequently than others.

For the simplicity of our analysis, we assume that the skew tuples are evenly distributed over computing nodes. Moreover, without loss of generality, we assume that each node has the same number of tuples before distributed joins, and the non-skew tuples are uniformly distributed. We use the notations in Table 1. There, the value of the load-imbalancing factor is the ratio of skewed tuples on the hot nodes (i.e., nodes that to process a disproportional load) over the number of tuples on non-hot nodes. Its obvious that there is $B_i \geq 1$, and the greater the value of an imbalancing factor is, the worse the load balancing will be.

For a simple ROJA method, all the skew tuples S are flushed to the hot nodes while others are redistributed, therefore the number of tuples on each node W_1 will be:

$$W_1 = \begin{cases} \alpha|S| + \frac{(1-\alpha)|S|}{n} + \frac{|R|}{n} & \text{(hot nodes)} \\ \frac{(1-\alpha)|S|}{n} + \frac{|R|}{n} & \text{(non-hot nodes)} \end{cases}$$

Thus, the value of the load-imbalancing factor B_1, is

$$B_1 = 1 + n \cdot \frac{\alpha|S|}{(1-\alpha)|S| + |R|} \tag{6}$$

In comparison, after the redistribution and duplication operations, the number of tuples on each node W_2 for POPI is shown as below. There, the $|L|$ is the number of duplicated tuples in R and $(|R| - |L|)$ is the redistributed part.

$$W_2 = \frac{\alpha|S|}{n} + \frac{(1-\alpha)|S|}{n} + \frac{|R| - |L|}{n} + |L|$$

Therefore, the load-imbalancing factor B_2 for POPI :

$$B_2 = 1 \tag{7}$$

From Eq. 6, we can see that ROJA will meet serious load-balancing problems when n and α is large. For example, if the skew tuples dominate about 20% of S and n is 100, then B_1 will be 26. In contrast, B_2 still equals to 1. This means that POPI can always achieve an optimal load balancing, which is critical for the scalability of joins in a distributed environment.

In above analysis, we assume that skewed tuples are evenly distributed over computing nodes, i.e., each node has the same number of skew tuples. To capture the upper bound of B_2, we consider an extreme condition for the case that the skew is not evenly distributed: all the skew tuples are only located in the first few nodes (according to their numbering), while there is no skew tuples for the final nodes. In this case, for the relation S, all the $\frac{|S|}{n}$ tuples in the first few nodes are skew and thus will be locally kept in joins. In the meantime, all the non-skew tuples, i.e., $\frac{(1-\alpha)|S|}{n}$ will be redistributed as normal. Therefore, the number of tuples on each node W_2' for POPI is:

$$W_2' = \begin{cases} \frac{|S|}{n} + \frac{(1-\alpha)|S|}{n} + \frac{|R|-|L|}{n} + |L| & \text{(first few nodes, i.e., hot nodes)} \\ \frac{(1-\alpha)|S|}{n} + \frac{|R|-|L|}{n} + |L| & \text{(non-hot nodes)} \end{cases}$$

Because the number of unique skew keys is normally quite small compared to input tuples, i.e., $|L| \lll |R| \leq |S|$, we have B_2':

$$B_2' = 1 + \frac{|S|}{(1-\alpha)|S| + |R|} < 1 + \frac{1}{(1-\alpha)} \tag{8}$$

We can see from Eq. 8 that the value of B_2' is actually increasing with increasing the value of α. However, for any give data skew, POPI still demonstrates a much better load balancing than ROJA. For example, there is $B_2' < 2.25$ rather than $B_1 = 26$ when $\alpha = 20\%$ and $n = 100$. It should be noticed that in real applications, we can use various strategies to remedy such kind of uneven skew such as redistribute skew tuples [6].

4.3.2. Network Traffic

In addition to the advantage of better load balancing, the proposed POPI can also effectivly reduce the network traffic for distributed outer joins. This advantage is important, as any communication reduction in a distributed join will be directly translated to faster execution [14].

The network traffic of each join algorithm can be measured by the amount of tuples transferred over network in join executions (assume the size of each tuple is the same). As all the tuples are redistributed in ROJA, its network traffic N_1 is:

$$N_1 = (|S| + |R|) \cdot \frac{n-1}{n} \tag{9}$$

Table 1: Table of notations

Notation	Meaning
T	*an input relation of an outer join (i.e., R and S)*
$\lvert T\rvert$	*the number of tuples in relation T*
n	*the number of computing nodes in the system*
α	*the portion of skew tuples in S*
L	*set of skewed keys in S*
W_i	*number of to be join tuples on each node for the i-th algorithm*
N_i	*the number of transferred tuples for the i-th algorithm*
B_i	*the load imbalancing factor for the i-th algorithm*

For POPI, the skew tuples, the number of which is $\alpha|S|$, are not transferred at all. Instead, only a small number of tuples from R (i.e., $|L|$) is duplicated. Therefore, the network traffic N_2 is:

$$N_2 = ((1-\alpha)|S| + (|R| - |L|)) \cdot \frac{n-1}{n} + |L|(n-1) \tag{10}$$

Moreover, following the details in Section 2.2, the DER algorithm duplicates the relation R and redistributes the non-matched row-ids. We use $\mathcal{X}|R|$ to represent the network traffic for latter operation. Here, the value of $\mathcal{X}$ depends on the size of a row-id (compare to a tuple) and the also selectivity of the join in the first phase of DER. Obviously, we have $\mathcal{X} < 1$, and the network traffic N_3 for DER is:

$$N_3 = (n-1)|R| + \mathcal{X}|R| \tag{11}$$

From Eq. 9 and Eq. 10, we have $N_1 - N_2 \approx \alpha|S|$. Namely, POPI will transfer less data than ROJA, and this difference will be obvious when the data skew is high. Consider the case between POPI and DER based on Eq. 10 and Eq. 11, we can see that DER would perform better than POPI when R is small. For an extremely case such as that $|R| = |L| \lll |S|$, we then have that $N_2 - N_3 \approx (1-\alpha)|S|$. However, POPI will have an obvious advantage in network traffic when R is large, i.e., our approach will be able to efficiently process large-large table outer joins while DER can not. The reason is that POPI does not rely on the operation of duplicating a full input relation but DER does. Namely, N_3 will be much greater than N_2 when $|R|$ is large in a distributed environment. For example, there is $N_3 - N_2 \approx (n-3+\alpha)|S|$ when $|R| = |S|$. In fact, as we will show in our later evaluation that POPI outperforms DER even when the input relation R is relatively small (i.e., 1 million tuples, around 138 MB in size).

4.3.3. Implementation Complexity

The proposed POPI is very easy on implementation. For example, it does not require the additional operations used in DOJA and DER to identify the non-matched results. Moreover, as depicted in Figure 2, we can observe that the data partitioning and join pattern of POPI is very similar to the PRPD and PRPS-O approach. However, as we have discussed that the PRPD algorithm can not be effectively applied to outer joins. On the other hand, although PRPS-O can efficiently process distributed outer joins, its detailed implementation is much more complex than POPI, i.e., for the skewed tuples, POPI uses a straightforward inner join rather than a semijoin-like approach. In such scenarios, POPI will be simpler to implement and also be more efficient. Therefore, unlike our previous work [11], we will not compare the detailed performance difference between POPI and PRPS-O in this work, although the difference could be insignificant in a memory-based distributed platform.

4.4. Applicability

As we have discussed in Section 3, some join techniques are efficient and can be used for outer join executions, however, they could meet problems on being applied to current data systems. In contrast to this, POPI is actually a very practical approach and can be integrated to current systems directly. In terms of parallel and distributed data processing, redistribution and duplication are the two most popular patterns used in current big data platforms (frameworks) such as MapReduce [13] and Spark [12]. POPI is based on the two patterns, therefore it can be easily applied to these computing platforms for data operators including outer joins, without any changes to their underlying architectures.

We have described that the PRPD algorithm [6] is very efficient on skew handling. In fact, PRPD is a one of the most mainstream approaches for inner joins. It has been adopted and studied in various commercial solutions (e.g., Microsoft [3] and Oracle [18]) and data applications (e.g., semantic web [19]). The proposed POPI uses the same data partitioning strategy as PRPD, therefore it can be seamlessly integrated in the current solutions and applications which have adopted PRPD. Moreover, advanced strategies designed for PRPD (e.g., skew quantification and tuning) can be also applied to POPI directly. For instance, statistical information of underlying data collected by a system for inner join implementations such as data skew will be able to be used by POPI. In detail, skew statistics on the join keys (a, b, c) for the inner join implementation $R(a, x) \bowtie S(b, y) \bowtie T(c, z)$ can be applied to the implementation of $R(a, x) ⟕ S(b, y) \bowtie T(c, z)$ directly. Actually many approaches can be used to collect such statistics in database systems and other data processing platforms. Examples include histograms [20] and bifocal-sampling [21] etc., and all such approaches are orthogonal to POPI. In our later implementations, we have chosen a very simple method, i.e., we sample the whole input to quantify the data skew[2].

It should be noticed that POPI focuses on high-level decomposition and parallelism for outer-joins rather than providing detailed underlying join implementations. In terms of join implementations, $R_{redis} ⟕ S_{redis}$ and $R_{dup} \bowtie S_{loc}$ in our case, we will be able to use any existing distributed join method. For example, we can adopt the conventional redistribution and duplication-based joins, or the advanced designs such as the ones considering data locality [22] and network bandwidth conditions [23]. Moreover, we can simply change to the outer join in $R_{redis} ⟕ S_{redis}$ to an inner join when processing an inner join between the relations R and S. In such scenarios, we believe that POPI has provided a general decomposition and parallel framework for distributed join executions, and this approach will be very suitable for processing large and skew data joins.

The outer join between the non-skew part in POPI will not generate any data skew. However, the inner join between the skewed tuples could possibly bring in the *join product skew* [24], because of unevenly partitioned skew. In this case, current approaches against such kind of skew appearing in inner joins can be also directly applied to our implementation (e.g., skew redistribution [3]), as these operations are independent with our underlying join implementation. We do not consider such kind of skew in our later evaluations, as we focus on handling the more common attribute skew in this work. Actually, as stated in [6], unevenly partitioned skew is rare. Moreover, in a real data processing system, a cost based optimizer will determine whether the cost of applying POPI is smaller. We will leave out the discussion on how an optimizer computes and compares the costs of POPI with current approaches such as ROJA and DER, which is beyond the scope of this paper.

4.5. An Implementation

We present a general implementation of our method using Spark [12]. Specifically, we have implemented all approaches using Scala on Spark over HDFS [25] in our evaluations. We choose Spark as it supports distributed in-memory computing, can achieve high performance for data processing and is very popular. Moreover, it has provided concise notations for various join implementations, such as the redistribution-based outer joins (e.g., `leftOuterJoin`) and inner joins (e.g., `join`). This means that local join operations as well as data transfer processes (e.g., shuffle and broadcast) have been integrated in the platform, which allows us for a fair performance comparison in our later evaluations. In the following, we assume that readers are familiar with the concepts of HDFS and the RDD in Spark.

The general implementation of POPI over Spark 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 and the threshold to chose skewed keys respectively. Initially, all the tuples in R and S are read as key-value pairs from underlying HDFS system. Then, we sample the table S (line 1) and count the number of occurrences of join keys in sampled tuples on each RDD partition. All the statistic information will be collected and the skewed key will be filtered out by the input threshold t (line 3). Based on the extracted skewed keys, all the tuples in R and S are partitioned into two parts respectively by checking whether each of their join key is skewed or not (lines 4-8). With the partitioned data, we then start the outer join (line 9) and the inner join (line 10-11). It should be noticed that there is always a match for each tuple in the later inner join, thus there we can output the joined result of each tuple directly. Finally, the outputs of the outer join are composed by the results from the two joins (line 12).

Algorithm 1 POPI Outer Joins

```
     Input: R, S, k, t
     Output: R ⟕ S
 1:  read R and S as key-value pairs from HDFS
     //skew collection
 2:  var SSample = S.sample(false, k)
 3:  var SkewedKey = SSample.map(x ⇒ (x._1,1))
         .reduceByKey(_+_).collectAsMap()
         .filter(x ⇒ x._2 ≥ t)
     //data partitioning
 4:  val Bsk = sc.broadcast(SkewedKey)
 5:  var SLoc = S.filter(x ⇒ Bsk.value.get(x._1) != None)
 6:  var SDis = S.filter(x ⇒ Bsk.value.get(x._1) == None)
 7:  var RDup = R.filter(x ⇒ Bsk.value.get(x._1) != None)
 8:  var RDis = R.filter(x ⇒ Bsk.value.get(x._1) == None)
     //join execution
 9:  var Results1 = RDis.leftOuterJoin(SDis)
10:  val Brd = sc.broadcast(RDup)
11:  var Results2 = SLoc.map(x ⇒ (x._1, Brd.value.get(x._1), x._2))
12:  output the union of Results1 and Results2
```

In this implementation, we have to sample and partition the input relations before the joins. However, these operations will be very light-weight in a distributed environment, since the former processing is only a simple statistics-based job and latter one contains only local scans. In fact, in real data systems, the required statistic information are normally collected by optimizer when the data is stored. In this condition, we will be able to partition the relations directly by the collected statistic.

It is obvious that implementing an algorithm using different programming languages or systems would lead to different execution times. However, the technique for parallel execution rather than the language or library used for implementation is more important. As we have theoretically analyzed in Section 4.3, POPI can always achieve optimal load-balancing and is able to process large outer joins, regardless of underlying systems. In such scenarios, we believe that implementing current outer join approaches over other distributed in-memory platforms will demonstrate a similar result as that on Spark. Additionally, although Spark SQL [26] has provided the functional programming APIs on relational processing, it mainly focuses offering optimization for query execution (i.e., multiple database operators). Namely, its detailed join implementations still rely on the conventional redistribution and duplication-based joins. Since we have provided an outer join approach building these operation, POPI can be used in Spark SQL to enrich its APIs or as part of the optimisation process.

[2] Note that this is the worst case for our implementation. In the results section, we can see that POPI still performs faster than current outer join approaches.

5. Evaluation

In this section, we present a detailed experimental evaluation of POPI and compare its performance with the approaches as described in Section 2. Because the DER algorithm is an optimized method for DOJA, we only use ROJA and DER in our comparisons. The test code for each approach we have used in this section is available at `https://github.com/longcheng11/POPI`.

5.1. Platform

We have evaluated the performance of our approach on a cluster of up to 17 computing nodes. Each node has two 12-core Intel Xeon CPU E2680 processors running at 2.50 GHz, resulting in a total of 24 cores per physical node. Each node has 128GB of RAM, a single 128GB 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 Spark version 2.0.0, Hadoop version 2.7.3, Scala version 2.11.4 and Java version 1.7.0_25.

5.2. Datasets

We adopted the widely used TPC-H benchmark [27] in our tests. We used the query shown as below in our experiments. For simplicity, we refer the relation CUSTOMER as R and SUPPLIER as S in the following.

```
select *
from CUSTOMER as R left outer join SUPPLIER as S
on R.NATIONKEY = S.NATIONKEY
```

For the datasets generated by TPC-H, each tuple has 8 attributes in the relation R and 7 attributes in S. We choose *Nationkey* as the their join key and other attributes as the payload. Since there are only 25 unique uniform *Nationkey* values in TPC-H, if we generate a large number of tuples, all tuples in both R and S will be skewed. To highlight data skew and also control the skew in our experiments, similar to the many approaches [6], we increase the number of unique *Nationkey* to 50000, and randomly choose a portion of tuples in S and change their *Nationkey* to a specified value. For example, the following statement specifies that the skewness of S is 10%. In this way, we can easily understand exactly what experiment is being performed and also be able to capture the essence of a Zipfian distribution [28, 29].

```
update SUPPLIER
set S.NATIONKEY = 1
where random(1,100) ≤ 10
```

We vary the scale factor of TPC-H to generate datasets with different sizes. We fix the tuples of the relation S to 100 million (around 13.4 GB) and vary the number of tuples in R using 50 million as the default value. Moreover, we vary the inner join cardinality (referred to as **select**) of R and S to 0%, 50% and 100% by controlling the values of join keys in R while keeping the sizes of R and S constant. We set 50% as the default value.

5.3. Setup

We set the following parameters for the underlying Spark platform: *spark. worker.memory* and *spark.executor.memory* are set to 120 GB and *spark. worker.cores* is to 24. Because the size of the query results could be very large (e.g., around 10 billion rows for a full match condition), outputting all of them will be costly. To focus on the runtime performance of each outer join implementation, we only record the number of the final outputs, rather than materialising the output. Moreover, as there is only one skewed key in our data sets, we just extract the skew by checking whether the value of the key equals 1 or not, after collecting the data statistic information in our implementation. In all our experiments, the operations of input file reading and final result output are both on the HDFS system. We configure HDFS to use the SSD on each node, to try our best to prepare a fully in-memory computing environment. We measure runtime as the elapsed time from job submission to the job being reported as finished and we record the mean value based on three measurements.

5.4. Runtime

We examine the runtime of three algorithms: the ROJA, DER and POPI. We implement our tests using 9 nodes in the cluster, one master and 8 slaves (i.e., worker nodes, 192 cores), on the default datasets with varying skew.

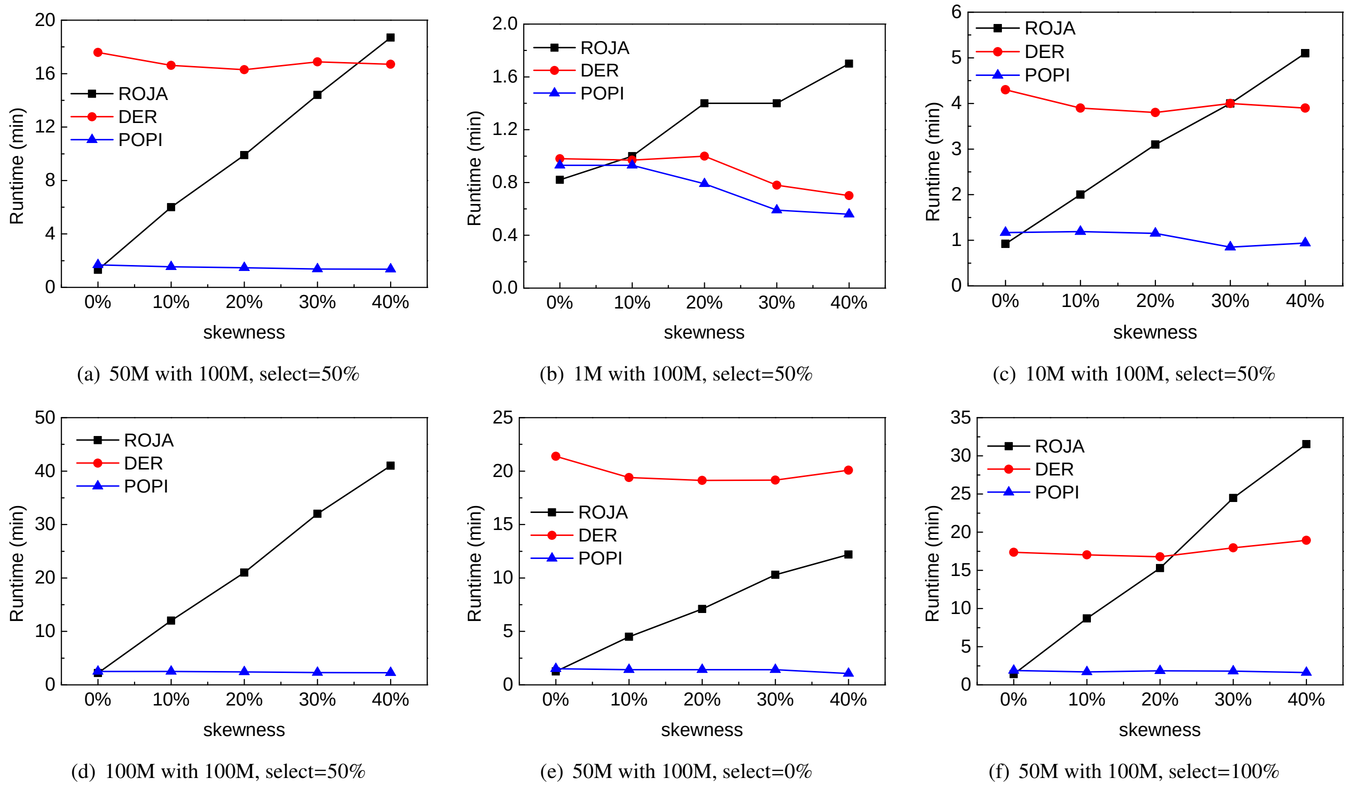
5.4.1. Experiments with Default Setting

Figure 3(a) shows the runtime of each algorithm under varying skew. We see that, with increasing data skew, the runtime of ROJA increases sharply, showing its performance issues in the presence of data skew. In comparison, the runtime of DER and POPI remains fairly constant or slightly lower, which means that these two algorithms are more robust in the presence of data skew. Moreover, though the DER algorithm is a specific approach on outer join optimization, we can observe that its performance is generally worse than that of ROJA algorithm, except in the case of very high skew (i.e., 40%). As DER mainly relies on data duplication, we can see that this operation could lead to performance issues on outer join implementation (i.e. it could be much more costly compared to data redistribution in the ROJA algorithm). In all cases, we can see that POPI always performs the best[3]. Additionally, compared to DER, POPI is significantly faster on performing outer joins. As shown in Table 2, the average runtime (under varying skew) of DER is 16.8 mins while POPI is only 1.5 mins, demonstrating a speedup of 11.2.

5.4.2. Experiments by Varying Cardinality

We examine the outer join execution time by varying the cardinality of the relation R. Using the TPC-H data generator, we create additional data sets with the number of tuples set to 1

[3] We are slightly slower than the ROJA algorithm when the skew is 0, as we have the additional sampling operation. Regardless, the time difference is small. For example, sampling the full input datasets in our tests (again, the worst case in real applications) takes 0.22 minutes.



(a) 50M with 100M, select=50%

(b) 1M with 100M, select=50%

(c) 10M with 100M, select=50%

(d) 100M with 100M, select=50%

(e) 50M with 100M, select=0%

(f) 50M with 100M, select=100%

Figure 3: Runtime of each algorithm under varying skew, with varying cardinality and selectivity over 8 worker nodes (192 cores).

million, 10 million and 100 million. The first relation can be considered as a very small dataset and the latest one can be considered as a large dataset. We vary the data skew of S and record the runtime.

The results under these conditions are shown in Figure 3(b), Figure 3(c), and Figure 3(d) respectively. We see that, (1) for a very small R, the runtime of DER and POPI is very close to each other and smaller than the ROJA algorithm when the data is skewed; (2) for a relatively small R (i.e., 10M), our POPI approach starts to outperform the other two algorithms, and DER starts to become to be slower than ROJA on the condition of data skew is not very high; (3) for the case with 100 million tuples for R, DER times out[4]. The reason could be that broadcasting large data set is costly and time spend on redundant computation dominates the runtime. In addition, we can see that POPI is much faster than ROJA under skew.

In the meantime, we can see that the runtime of DER and POPI remains fairly constant with increasing data skew, which shows their robustness in the presence of skew under different cardinality conditions. Moreover, we can observe that with the increase of the size of R, the performance advantage of our POPI approach becomes more pronounced. Obviously, the runtime of all the three approaches increases with the growing of the input size. However, DER runtimes are increasing at a higher rate than those of POPI (see Table 2) illustrating its inefficiency in large-large outer joins. In comparison, the runtime of POPI increases slightly with increasing the size of the input, indicating its very good scalability in this aspect. Moreover, it shows that POPI can processing large outer joins very quickly. Looking at edge cases, when one input relation would be *extremely* small, DER would outperform POPI (as POPI has the overhead of data partitioning and skew sampling). Regardless, for a small case here (i.e., 1 million tuples), we can see that POPI is still faster than DER.

5.4.3. *Experiments by Varying Selectivity*

We also examine how join selectivity affects the performance for each algorithm. For datasets with different skew distributions, we created two different R that have the same cardinality as the default dataset but with 0% and 100% of the tuples having a match in S. The results for these two conditions are presented in Figure 3(e) and Figure 3(f) respectively. Again, they demonstrate that DER and POPI can efficiently handle data skew under conditions with different join selectivity. In conjunction with the results in Figure 3(a), we notice that ROJA and POPI show increased runtime with increasing selectivity, and the runtime of ROJA increases much more severely than POPI under skew in this process. In comparison, DER firstly decreases and then increases (also can be seen in Table 2). The possible reasons for this are: (1) For ROJA, local computation increases since the number of final results increases with increasing the selectivities[5]. In the meantime, the skew amplifies

[4] The implementation of DER approach suspends for more than two hours and thus we stop the execution manually.

[5] Note that Spark uses a single `map` data structure to collect all the tuples in both R and S in the form of $(K, (iterable[V_r], iterable[V_s])$, and the local join

Table 2: Average time cost (mins) by varying the skew under different cadinalities and inner join cardinality.

parameter /method	# tuples in R 1M	10M	50M	100M	selectivity 0%	50%	100%
DER	0.9	4	16.8	INF.	19.8	16.8	17.6
POPI	0.8	1.1	1.5	2.4	1.4	1.5	1.8
speed-up	1.1	3.6	11.2	INF.	14.1	11.2	9.8

this increase for the hotspot nodes and thus the general runtime increases. (2) POPI, though it is not impacted by data skew, is still impacted by the increased cost of result collection. Moreover, the size of the duplicated part also increases with increasing the selectivity, which could incur extra time cost. (3) For DER, the redistribution of non-matched row-ids decreases with increasing selectivity, which leads to a decrease on runtime as it reduces network communication. With selectivity reaching 100%, local computing gradually outweighs the reduction effects of network communication, which leads to an increase in runtime. To summarise the difference between DER and POPI, we see that POPI is always notably faster than DER, but the difference is more pronounced for low selectivity, which can be also observed in Table 2.

Summary. Combining the results presented above, we can see that, both the DER and POPI algorithms can efficiently deal with data skew under various cardinality and selectivity conditions, while the ROJA approach can not. In the meantime, POPI outperforms the other two approaches in general, especially compared to the state-of-art DER algorithm, demonstrating its advantages in big data applications.

5.5. Network Communication and Load Balancing

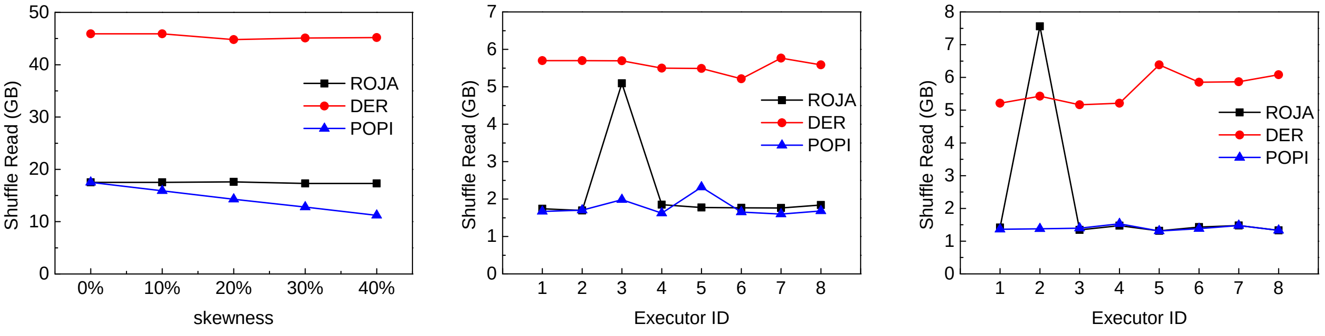
We analyze the network communication by recording the metric *shuffle read*, as provided by Spark. This number quantifies the amount of data in bytes read from remote executors (physical machines) but not the data read locally. This means that this metric indicates the data transfers over the network during executions. Results by varying skew over 8 workers (192 cores) are shown in Figure 4(a). We can see that ROJA and POPI transfer almost the same amount of data in the absence of skew. This is reasonable, since all tuples in these two algorithms are processed only by redistribution. In the meantime, we can see that the transferred data by the DER algorithm is much higher than ROJA and DER. The reason for this is that the entire relation R is broadcast. Moreover, with increasing data skew, the transferred data remains roughly constant for DER and ROJA while significantly decreasing for our POPI approach. The reason is that the redistributed and the duplicated data in ROJA and DER do not change. In comparison, skewed tuples are local kept in POPI, the more skewed the data is, the more the data will be local kept, thus the network communication is decreasing with increasing the skew.

We measure the load-balancing characteristics of the three algorithms based on the *shuffle read at each executor* metric provided by Spark. This metric records the received data by each executor. The more data an executor receives, the more time will be spent on data transfer (and join operations) on this node. We record the results for the conditions in which the skew is set to 20% and 40% and present them in Figure 4(b) and Figure 4(c) respectively. We can observe that the ROJA algorithm have a obvious peak in its two curves respectively. This means that a large number of tuples has been flushed to a single worker node, demonstrating the bad load balancing of the algorithm in the presence of data skew. Moreover, with increasing skew, the load imbalance is becoming more pronounced (as the peak value becomes larger). In comparison, the amount of data read from remote nodes of each executor is generally the same for DER and POPI for the two cases, showing their good load balancing on outer join implementations under various skews. Additionally, we can see that the data read at each executor for POPI algorithm is much less than that of DER, reaffirming its advantage on network communication.

5.6. Scalability and Speedup over ROJA Algorithm

We test the scalabilty (scale-out) of our implementation by varying the number of executors (worker nodes) under varying skew over 4 nodes (96 cores), 8 nodes and 16 nodes (384 cores). The results for the runtime are demonstrated in Figure 5(a). We can observe that the proposed POPI algorithm can achieve significant speedups with increasing the number of nodes under different skew. For example, when the skew is 20%, its runtime is 1.99 mins, 1.47 mins and 0.84 mins respectively by doubling the number of nodes, demonstrating that it scales well with the number of nodes. For different worker configurations, we notice that we can get almost linear speedup when doubling the number of workers from 8 to 16 while relatively small speedup when doubling the number of workers from 4 to 8. We attribute this to the following two reasons: (1) the inter-machine communication is limited and very fast in a small system of four nodes. Namely data redistribution and duplication could be relatively cheaper. In such a case, the additional overhead of data sampling and partitioning for POPI plays a relatively more important role in performance; and (2) the increased transferred data is becoming less with increasing the number of nodes and thus bring additional runtime improvements. This is also supported by recording the network communication as shown in Figure 5(b). There, the size of the retrieved data across the entire system increases with increasing the number of nodes. The reasons could be: (1) data transmission on the broadcast part increases, and (2) the number of local assigned tuples for the redistributed part decreases. However, we can see that the increases of the network communication is at a reducing ratio to

results are formulated based on iterate operations over the `map`.

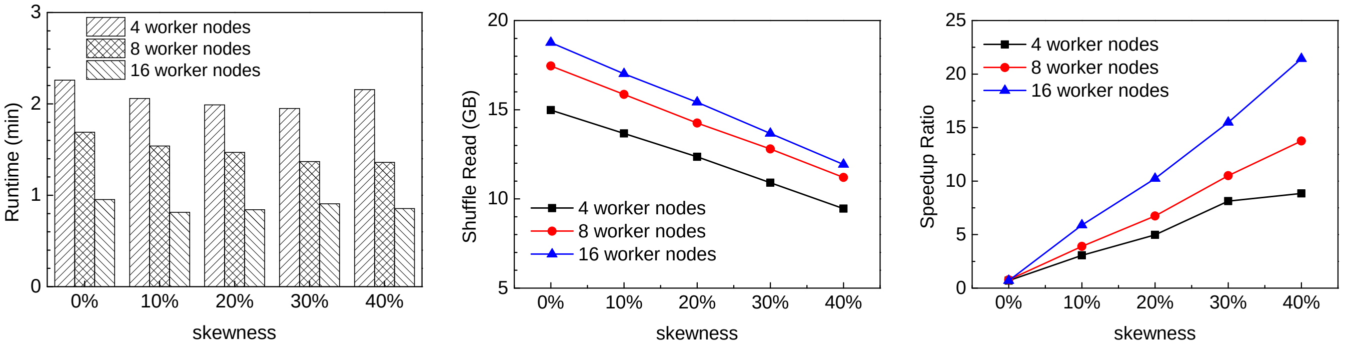


(a) 50M with 100M, select=50%

(b) 50M with 100M, skew=20%

(c) 50M with 100M, skew=40%

Figure 4: The size of shuffle read data and detailed read data for each executor under different skews. Select=50%, 8 worker nodes (192 cores).



(a) 50M with 100M, select=50%

(b) 50M with 100M, select=50%

(c) 50M with 100M, select=50%

Figure 5: Runtime and the shuffle read data of POPI algorithm as well as its speedup ration over the ROJA algorithm by varying the number of worker nodes in the presence of different skews.

the number of workers. Namely, for larger numbers of workers, the retrieved data at each executor will obviously decrease when doubling the number of workers.

We conclude our analysis with the presentation of speedup against the most widely used ROJA method as a baseline, by analyzing the performance improvement achieved by POPI for different numbers of nodes. The results are presented in Figure 5(c). As we can see, our POPI approach can achieve obvious speedups under different skews, and the speedup ration increases with increasing the data skew. In the meantime, under different skews, the larger the systems is, the higher the speedup we achieve, indicating superior scalability.

6. Related Work

Research in parallel joins on shared memory systems has already achieved significant performance speedups through improvements in architecture at the hardware-level of modern processors. Moreover, to support real-time data analytics, current studies have tried to accelerate stream joins over in-memory multi-core platforms [30]. Orthogonally to these techniques, we focus on the challenges of *data skew* in *large* distributed outer joins in this work.

Besides the conventioanl databases, data skew is also a significant problem for various data applications. Generally, there are four main types of skew in distributed joins [31]: tuple placement skew, selectivity skew, redistribution skew and join product skew. The first two types can be effevtively avoided [6]. For example, tuple placement skew can be removed by a good hash function. In contrast, the latter two types of skew bring significant challenges on performance of real applications. To alleviate these problems, there has been in-depth research on skew handling in parallel and distributed systems [3, 5, 6, 18, 28, 20]. Most approaches focus on inner joins and there has been relatively little done on the topic of outer joins - a surprising fact given that outer joins are common in complex queries and in data warehousing scenarios. This makes some applications (e.g., [19]) be able to process their inner joins efficiently, but they can only use ROJA for their outer joins. In such scenarios, we believe that POPI has provided a new and effective option for outer joins in big data applications.

Several efforts in designing high level query languages in the large-scale data analytics community, such as Pig [32] and Hive [33] as well as the systems (and techniques) presented in the suvey [34], have employed mechanisms on skew handling in outer joins (e.g., histogram-based outer join), however, as we have shown in our previous evalutions [11], sometimes they could be not very efficient. Additionally, although some platforms (e.g., Stratosphere [35]) 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 data operation as we have studied in this work. On the other aspect, our approach can be also applied to all above frameworks and platforms to process outer joins.

Current research on outer joins focuses on optimization of existing methods, which mainly includes outer join elimination [36], outer join reordering [7] and view matching for outer join views [37]. There is little done on the skew handling on outer join implementations. The reason for this may be the assumption that inner join techniques can be simply applied to outer joins [8]. However, as we have analysis in [10], sometimes, applying such techniques for outer joins directly may lead to poor performance (e.g., the PRPD algorithm [6]). To the best of our knowledge, there are only few approaches designed for skew handling in outer joins. The work [38] proposes an efficient method called OJSO (outer join skew optimization) for outer joins. However, the approach focuses on handling the skew generated by an outer join, which is different from the attribute skew problem as we have studied in this work. Moreover, although the approach DER [8] (as well as DDR [16]) can process attribute skew, it can be only applied for small-large table outer joins but not for the large-large ones. As we have shown in our evaluation, the proposed POPI can perform much better than DER even when a dataset is relatively small.

Recently, Bruno et al. [3] presented three *SkewJoin* transformations to mitigate the impact of data skew in a distributed join operations. There, the alternatives *F-SkewJoin* and *H-SkewJoin* are able to handle outer joins with skewed distributions. 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-weight, since we do not need to repartition the skewed tuples, the number of which is always huge. Even in the condition that some skewed tuples do not appear on some nodes, we will not generate *null* values, as we use an inner join operation for the skewed tuples in our approach.

Moreover, Cheng et al. [15, 10] introduce several new approaches on skew handling for large table outer joins. However, their proposed *query-based* implementations either rely on fine-grained control of data movement at a thread level, or have to rewrite data structures of underlying frameworks. Compared to this, the proposed POPI approach uses a more efficient and simple way - inner joins, on the skewed part of the data. On the other hand, this also means that well-studied inner join techniques can be applied in our outer join implementation.

7. Conclusions

In this paper, we have introduced an efficient outer join approach, called POPI (Partial Outer join & Partial Inner join), which specifically handles data skew in a distributed environment. We have presented the detailed implementation of our approach and conducted an extensive performance evaluation by comparing with the current methods. The experimental results have shown that our method is efficient, scalable and presents good load balancing characteristics.

Since POPI follows the conventional redistribution and duplication operations, we anticipate that our approach will be a supplement to existing schemes on parallel joins, and thus be applied on a variety of existing systems, ranging from parallel database systems to analytics frameworks. As join-like operations are ubiquitous in the area of big data, it would be valuable to quantify the benefit of our approach across different datasets and applications (e.g., large-scale machine learning). In the meantime, with the advent of more advanced computing architectures and platforms, we are also interested in applying our approach in more complex computing environments (e.g., such as mobile and cloud computing systems [39, 40]). Our long-term goal is to develop a high-performance data analytics system which can process big data in a robust and efficient manner in large-scale distributed scenarios.

Acknowledgments

Part of this work is supported by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 799066. The authors also thank the reviewers for their careful reading of the manuscript and many insightful comments and suggestions, which has great help for improving the work.

References

[1] B. Marr, Big data facts: How many companies are really making money from their data? (2016).

[2] 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: Proc. 2010 ACM SIGMOD Int. Conf. Management of Data, 2010, pp. 975–986.

[3] N. Bruno, Y. Kwon, M.-C. Wu, Advanced join strategies for large-scale distributed computation, Proc. VLDB Endowment 7 (13) (2014) 1484–1495.

[4] M. A. H. Hassan, M. Bamha, An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems, in: Proc. 2009 Int. Conf. High Performance Computing, 2009, pp. 350–358.

[5] L. Cheng, S. Kotoulas, T. E. Ward, G. Theodoropoulos, Robust and skew-resistant parallel joins in shared-nothing systems, in: Proc. 23rd ACM Int. Conf. Information and Knowledge Management, 2014, pp. 1399–1408.

[6] Y. Xu, P. Kostamaa, X. Zhou, L. Chen, Handling data skew in parallel joins in shared-nothing systems, in: Proc. 2008 ACM SIGMOD Int. Conf. Management of Data, 2008, pp. 1043–1052.

[7] C. Galindo-Legaria, A. Rosenthal, Outerjoin simplification and reordering for query optimization, ACM Trans. Database Systems 22 (1) (1997) 43–74.

[8] Y. Xu, P. Kostamaa, A new algorithm for small-large table outer joins in parallel DBMS, in: Proc. IEEE 26th Int. Conf. Data Engineering, 2010, pp. 1018–1024.

[9] S. Kotoulas, E. Oren, F. van Harmelen, Mind the data skew: distributed inferencing by speeddating in elastic regions, in: Proc. 19th Int. Conf. World Wide Web, 2010, pp. 531–540.

[10] L. Cheng, S. Kotoulas, Efficient skew handling for outer joins in a cloud computing environment, IEEE Trans. Cloud Computing 6 (2) (2018) 558–571.

[11] L. Cheng, S. Kotoulas, Efficient large outer joins over MapReduce, in: European Conf. Parallel Processing, 2016, pp. 334–346.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proc. 9th USENIX Conf. Networked Systems Design and Implementation, 2012, pp. 15–28.

[13] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[14] O. Polychroniou, R. Sen, K. A. Ross, Track join: distributed joins with minimal network traffic, in: Proc. 2014 ACM SIGMOD Int. Conf. Management of Data, 2014, pp. 1483–1494.

[15] L. Cheng, S. Kotoulas, T. E. Ward, G. Theodoropoulos, Robust and efficient large-large table outer joins on distributed infrastructures, in: Proc. 20th European Conf. Parallel Processing, 2014, pp. 258–369.

[16] L. Cheng, I. Tachmazidis, S. Kotoulas, G. Antoniou, Design and evaluation of small-large outer joins in cloud computing environments., Journal of Parallel and Distributed Computing 110 (2017) 2–15.

[17] S. Blanas, Y. Li, J. M. Patel, Design and evaluation of main memory hash join algorithms for multi-core CPUs, in: Proc. 2011 ACM SIGMOD Int. Conf. Management of Data, 2011, pp. 37–48.

[18] S. Bellamkonda, H.-G. Li, U. Jagtap, Y. Zhu, V. Liang, T. Cruanes, Adaptive and big data scale parallel execution in Oracle, Proc. VLDB Endowment 6 (11) (2013) 1102–1113.

[19] S. Kotoulas, J. Urbani, P. Boncz, P. Mika, Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on Pig, in: Proc. 11th Int. Semantic Web Conf., 2012, pp. 247–262.

[20] M. A. H. Hassan, M. Bamha, F. Loulergue, Handling data-skew effects in join operations using mapreduce, in: Proc. International Conference on Computational Science, 2014, pp. 145–158.

[21] S. Ganguly, P. B. Gibbons, Y. Matias, A. Silberschatz, Bifocal sampling for skew-resistant join size estimation, ACM SIGMOD Record 25 (2) (1996) 271–281.

[22] W. Rödiger, T. Muhlbauer, P. Unterbrunner, A. Reiser, A. Kemper, T. Neumann, Locality-sensitive operators for parallel main-memory database clusters, in: Proc. 30th IEEE International Conference on Data Engineering, 2014, pp. 592–603.

[23] L. Rupprecht, W. Culhane, P. Pietzuch, Squirreljoin: network-aware distributed join processing with lazy partitioning, Proceedings of the VLDB Endowment 10 (11) (2017) 1250–1261.

[24] V. Poosala, Y. E. Ioannidis, Estimation of query-result distribution and its application in parallel-join load balancing, in: Proc. 22nd Int. Conf. Very Large Data Bases, 1996, pp. 448–459.

[25] D. Borthakur, HDFS architecture guide, Hadoop Apache Project (2008) 53.

[26] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., Spark SQL: Relational data processing in Spark , in: Proc. 2015 ACM SIGMOD Int. Conf. Management of Data, 2015, pp. 1383–1394.

[27] T. P. P. Council, TPC-H benchmark specification, Published at http://www.tcp.org/hspec.html.

[28] D. J. DeWitt, J. F. Naughton, D. A. Schneider, S. Seshadri, Practical skew handling in parallel joins, in: Proc. 18th Int. Conf. Very Large Data Bases, 1992, pp. 27–40.

[29] G. K. Zipf, Human behavior and the principle of least effort: An introduction to human ecology, Ravenio Books, 2016.

[30] M. Najafi, M. Sadoghi, H.-A. Jacobsen, SplitJoin: A scalable, low-latency stream join architecture with adjustable ordering precision., in: USENIX Annual Technical Conference, 2016, pp. 493–505.

[31] C. B. Walton, A. G. Dale, R. M. Jenevein, A taxonomy and performance model of data skew effects in parallel joins, in: Proc. 17th Int. Conf. Very Large Data Bases, 1991, pp. 537–548.

[32] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, U. Srivastava, Building a high-level dataflow system on top of Map-Reduce: the Pig experience, Proc. VLDB Endowment 2 (2) (2009) 1414–1425.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: a warehousing solution over a Map-Reduce framework, Proc. VLDB Endowment 2 (2) (2009) 1626–1629.

[34] Y. Zhang, T. Cao, S. Li, X. Tian, L. Yuan, H. Jia, A. V. Vasilakos, Parallel processing systems for big data: a survey, Proceedings of the IEEE 104 (11) (2016) 2114–2136.

[35] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al., The stratosphere platform for big data analytics, The VLDB Journal 23 (6) (2014) 939–964.

[36] J. Rao, H. Pirahesh, C. Zuzarte, Canonical abstraction for outerjoin optimization, in: Proc. 2004 ACM SIGMOD Int. Conf. Management of Data, 2004, pp. 671–682.

[37] P.-Å. Larson, J. Zhou, View matching for outer-join views, The VLDB Journal 16 (1) (2007) 29–53.

[38] Y. Xu, P. Kostamaa, Efficient outer join data skew handling in parallel DBMS, Proc. VLDB Endowment 2 (2) (2009) 1390–1396.

[39] Y. Mao, J. Wang, B. Sheng, M. C. Chuah, Laar: Long-range radio assisted ad-hoc routing in manets, in: Proc. 22nd IEEE International Conference on Network Protocols, 2014, pp. 350–355.

[40] J. Wang, Y. Yao, Y. Mao, B. Sheng, N. Mi, Fresh: Fair and efficient slot configuration and scheduling for Hadoop clusters, in: CLOUD, 2014, pp. 761–768.