Efficient Skew Handling for Outer Joins in a Cloud Computing Environment

Long Cheng and Spyros Kotoulas

Abstract—Outer joins are ubiquitous in many workloads and Big Data systems. The question of how to best execute outer joins in large parallel systems is particularly challenging, as real world datasets are characterized by data skew leading to performance issues. Although skew handling techniques have been extensively studied for inner joins, there is little published work solving the corresponding problem for parallel outer joins, especially in the extremely popular Cloud computing environment. Conventional approaches to the problem such as ones based on hash redistribution often lead to load balancing problems while duplication-based approaches incur significant overhead in terms of network communication. In this paper, we propose a new approach for efficient skew handling in outer joins over a Cloud computing environment. We present an efficient implementation of our approach over the Spark framework. We evaluate the performance of our approach on a 192-core system with large test datasets in excess of 100GB and with varying skew. Experimental results show that our approach is scalable and, at least in cases of high skew, significantly faster than the state-of-the-art.

Index Terms—Distributed join; outer join; data skew; cloud computing; Spark; HDFS

1 Introduction

Data warehouses and the Web comprise enormous numbers of data elements and the performance of data-intensive operations on such datasets, for example for query execution, is crucial for overall system performance. *Joins*, which facilitate the combination of records based on a common key, are particularly costly and efficient implementation of such operations can have a significant impact in improving the performance on a wide range of workloads, ranging from databases to decision support and Big Data analytics.

With data applications growing in scale, Cloud environments play a key role in application scale-out, exploiting parallelisation to speed up operation and extending the amount of memory available. In this light, efficient parallelisation of joins on shared-nothing systems is becoming increasingly desirable. Various distributed join algorithms have been studied [1] [2] [3] [4] [5] [6], however, 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 data analytics applications. For example, in the Semantic Web domain, queries containing outer joins account for as much as 50% of the total number of queries, based on the analysis of DBPedia query logs [7]. Moreover, in online e-commerce, customer ids are often left outer joined with a large transaction table for analyzing purchase patterns [8].

In contrast to inner joins, outer joins do not lose any tuples from one (or both) table(s) that do not match with any tuple in the other table [9]. As a result, the final join contains not only the matched part but also the non-matched part. Similarly to inner joins, there are two conventional approaches for distributed outer join implementations [8]: hash-based and duplication-based implementations. As we will explain later, these two methods suffer from performance issues when data skew is encountered: the former method suffers from poor load balancing and the latter induces redundant and costly network communication. As data skew occurs naturally in various applications [10], it is important for practical data systems to perform well in such contexts.

Though many algorithms have been designed for handling skew for inner joins [11] [12] [13] [14] [15], little research has been done on outer joins. The reason for this may be the assumption that inner join techniques can be simply applied to outer joins [8]. However, as shown in our evaluations later in this manuscript, applying such techniques for outer joins directly may lead to poor performance. Moreover, although many systems can convert outer joins to inner joins [16], providing an opportunity then to use inner join techniques, this approach necessitates rewriting mechanisms, which also prove complex and costly. Finally, methods which have been designed specifically for outer joins achieve significant performance improvements [8] over the aforementioned approaches. We will later see though that these state-of-the-art methods are design variations of the two conventional approaches (i.e. redistribution and duplication), making them only applicable in small-large table outer joins.

In this paper, we propose an efficient *query-based* approach, aiming at efficiently against data skew in massively parallel outer joins over shared-nothing systems. We implement our method over the Spark [17] framework and conduct a performance evaluation over a data stored in HDFS [18] on an experimental configuration consisting of 16 nodes (192 cores) and datasets of up to 106GB with a

- *L. Cheng is with the Faculty of Computer Science, TU Dresden, Germany. E-mail: long.cheng@tu-dresden.de*
- *S. Kotoulas is with IBM Research, Dublin, Ireland. E-mail: spyros.kotoulas@ie.ibm.com*

range of values for skew. We summarize the contributions of this work as below:

- We present a new algorithm called *query-based outer joins* for directly and efficiently handling skew in parallel outer joins.
- We analyze the performance of two state-of-art techniques in currently DBMSs: (1) PRPD [12], for skew handling in inner joins; and (2) DER [8], for optimizing inner join implementation of small-large table outer joins. We find that the composition of these methods (referred to as PRPD+DER) can potentially handle skew in large-large table outer joins. Our experimental results confirm this expectation.
- We present the detailed implementation of our design over the Spark platform and our experimental evaluation shows that our algorithm outperforms PRPD+DER at least in the case of high skew. Moreover, the results also demonstrate that our method is scalable, results in less network communication and presents good load balancing under varying skew.

This manuscript is an extension of our previous work [19]. Specifically, we extend our method applied in a DBMS/cluster to a Cloud computing environment. Compared to that approach (i.e. *query with counters*), we present two main changes on the design and implementation:

- We remove the *counter* from our previous implementation, since we are lacking the fine-grain, thread-level control over data ordering that is available in a cluster environment.
- We introduce an efficient method to identify data locality and consequently realize the *query* process, so as to enable our method to execute in environments where we do not have fine-grain control over data (re-) location, such as in a data processing framework like Spark. We expect that removing these restrictions allows the deployment of our method on a Cloud environment.

We believe that our approach is very important for join implementations over scale-out data platforms best suited to Cloud environments (such as MapReduce [20] or Spark [17]). Such platforms are sometimes preferable for certain non-transactional workloads, since they allow for easy deployment and straightforward scale-out capability, compared to the conventional parallel DBMSs [21]. For example, Facebook gathers almost 6TB of new log data every day, just formatting and loading such volumes of data into a data processing framework in a timely manner is a challenge [22]. Current cloud-based implementations integrate parallelization, fault tolerance and load balancing in a *simple* programming framework, and can be *easily* deployed in a large computing center or Cloud, making them extremely popular for large-scale data processing[1]. In fact, most vendors (such as IBM) provide solutions, either on-premise or on the cloud, to compute on massive volumes of structured, semi-structured and unstructured data for their business applications.

The rest of this paper is organized as follows: In Section 2, we present background on outer join algorithms and current techniques. We present our *query-based* algorithms in Section 3 and its implementation in Section 4. We provide a quantitative evaluation of our algorithms in Section 5. We report on related work in Section 6 while we conclude the paper and suggest future work in Section 7.

2 Background

In this section, we describe the two conventional outer join approaches, hash-based and duplication-based outer joins, and discuss their possible performance issues. Then, we present some current techniques that can be used for efficiently handling skew and improving performance over outer joins. As *left outer joins* are the most commonly used outer joins, we focus on this type of join in the following. The query below shows a typical left outer join between a relation R with attribute a and another relation S with attribute b, which is evaluated by the pattern $R \overline{\bowtie} S$.

```
select R.a R.x S.y
from R left outer join S        (Query 1)
on R.a = S.b
```

2.1 Conventional Approaches

Distributed outer joins are, in general, composed of a distribution stage followed by a local join process. To capture the core performance of queries, we focus on exploiting the parallelism within a single join operation between two input relations R and S over a n-node system[2]. We assume that both relations are in the form of <key, value> pairs, where *key* is the join attribution. Additionally, we assume R is uniformly distributed and S is skewed.

Hash-based. For hash-based approaches, as shown in Figure 1, the parallel outer joins contain two main phases, which is similar to the case for inner joins:

- *Phase 1.* The initially partitioned relation R_i and S_i at each node i are partitioned into distinct sets R_{ik} and S_{ik} respectively, according to the hash values of their join key attributes. Then, each of these sets is then distributed to a corresponding remote node. For example, tuples in R_{ik} and S_{ik} at node i will be transferred to the k-th node.
- *Phase 2.* A local outer join between received R_k (i.e. $\bigcup_{i=1}^{n} R_{ik}$) and S_k (i.e. $\bigcup_{i=1}^{n} S_{ik}$) at each node k is implemented in parallel to formulate the final outputs.

Duplication-based. The duplication-based outer join approach is shown in Figure 2. Its implementation includes two main phases, which has significantly differences compared to inner joins.

- *Phase 1.* R_i at each node is duplicated (broadcast) to all other nodes. Then, an inner join between R_i (i.e.

1. Note that, frameworks like MapReduce and Spark lack many features (e.g. schemas and indexes), which could make them not suitable for some cases, regardless, the detailed discussion in this aspect will be beyond the scope of this paper.

2. Note that, here, we focus on explaining the join pattern in a distributed environment. In terms of terminology, a node here means a computing unit (e.g. an execution core in Spark).

$\bigcup_{i=1}^{n} R_i = R$) and S_i is implemented in parallel at each node i. This formulates an intermediate result T_i at each node.

- *Phase 2.* The outer join between R and the intermediate join results T_k to construct the final outputs. This process is the same as the hash-based method described above.

2.2 Performance Issues

Every step for the two approaches above is implemented in parallel across the computing nodes, and the number of execution units can be increased by deploying additional machines. Both distributed schemes show the potential for scalability in terms of processing massively parallel outer joins (i.e. these approaches can be applied in scale-out architectures). However there are significant performance issues with both approaches.

While researchers have shown that implementations of the hash-based scheme can achieve near linear speed-up on parallel systems under ideal balancing conditions [2], when the data to be processed has significant skew, the performance of such parallel algorithms dramatically decreases [11]. This performance hit arises from the redistribution of tuples in relation R and S through the hashing function, and all the tuples having the same value for the join attribute being transferred to the same remote node. When the input is skewed, the popular keys will flood into a small number of nodes and cause hot spots. Such issues impact system scalability, which will be reduced as employing new nodes cannot yield improvements - the skew tuples will still be distributed to the same nodes. In addition to performance issues, load imbalance can also lead to memory exhaustion.

Duplication-based methods can reduce hot spots by avoiding hash redistribution of tuples with problematic (i.e. popular) keys. Nevertheless, when R is large, the broadcast of every R_i to all the nodes is costly. In addition, the inner join operation over large numbers of tuples (i.e. $\bigcup_{i=1}^{n} R_i$) at each node in the first stage impacts performance due to the associated memory- and lookup-cost. Furthermore, even if R is small, the cardinality of the intermediate join results will be large when S is highly skewed, which makes the second stage costly and consequently decreases the overall performance.

2.3 Dealing with Skew

As there are no specific methods for handling skew in large table outer joins, we discuss two typical techniques used for inner joins - one implements load assignment by histograms [13] and the other is the state-of-art PRPD [12] method. As the later approach is considered as a very important join strategy for current large-scale distributed computation [23], we will apply it in outer joins and evaluate its performance later.

Histograms. Hassan et al. [13] employ a method based on distributed histograms, which can be divided into two parts:

- *Phase 1.* Various histograms for R, S and $R \bowtie S$ are built at each node, in either local or global view or both.

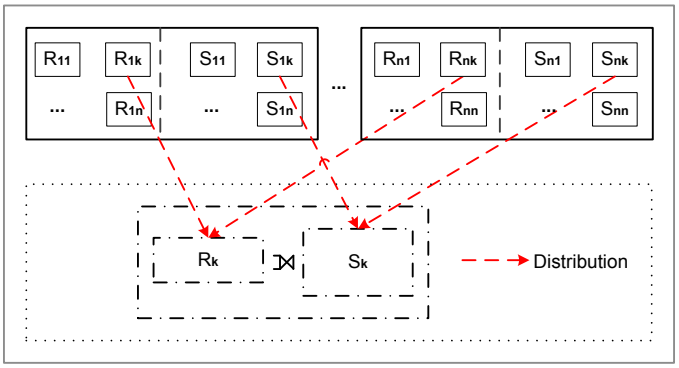

Fig. 1. Hash-based method. The initially partitioned relation R_i and S_i at each node are firstly partitioned and redistributed to all nodes based on the hash values of join attributes, and then the outer joins are implemented in parallel at each node. The dashed square refers to the remote computation nodes and objects.

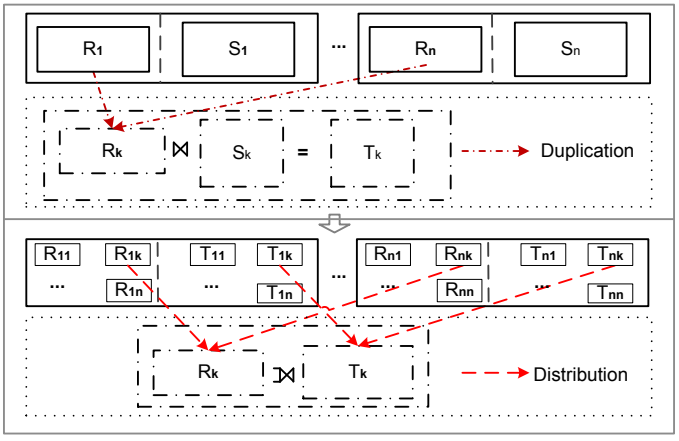

Fig. 2. Duplication-based method. R_i at each node is simply duplicated to all the nodes and then inner joins commence in parallel at each node (above). After that, the intermediate results T implements outer joins with R through the hash-based way (below).

- *Phase 2.* Based on the complete knowledge of the distribution and join information of the relations, a redistribution plan to balance the workload for each node is formulated.

As the primary innovation in that work is the improvement of the redistribution plan to process data skew, this method has the potential to be used for outer joins. While their experimental results demonstrate that this method is efficient and scalable in the presence of data skew, there are still two weak points: (1) histograms are built based on the redistribution of all the keys of R and S, which leads to high network communication, and (2) though only the tuples participating in the join are extracted for redistribution, which reduces part of the network communication, this operation is based on the pre-join of the distributed keys, which incurs a significant time cost.

PRPD. Xu et al. [12] propose an algorithm named PRPD (*partial redistribution and partial duplication*) for inner joins, which is a state-of-the-art method used for data skew handling for inner joins over a distributed system. For a single skew relation S, their implementation can be divided into the following two phases:

- *Phase 1.* S is partitioned into two parts: (1) S_{loc}, which comprises high skew items, and will be kept locally during the whole join processing; and (2) S_{redis}, which comprises the tuples with low frequency of occurrence and is redistributed using a common hash-based implementation. In the meantime, the relation R is also divided into two parts: (1) R_{dup}, the tuples in which contain the key in S_{loc}, which will be duplicated (broadcast) to all other nodes; and (2) R_{redis} - the rest part of R that is to be redistributed as normal.
- *Phase 2.* After the duplication and the redistribution operations described above, the final join will be composed by the received tuple at each node, namely $R_{redis} \bowtie S_{redis}$ and $R_{dup} \bowtie S_{loc}$.

This method illustrates an efficient way to process the high skew tuples (keys are highly repetitive): all these tuples of S are not redistributed at all, instead, just a small number of tuples containing the same keys from R are broadcast. The experimental results presented in [12] have shown that PRPD can achieve significant speedup in the presence of data skew, compared to the conventional hash-based method.

In fact, PRPD is a hybrid method combining both the hash-based and duplication-based join scheme. Therefore, we can simply use the $R_{redis} \rightarrowtail\!\!\!\!\bowtie S_{redis}$ and $R_{dup} \rightarrowtail\!\!\!\!\bowtie S_{loc}$ to replace the corresponding inner joins in the scenarios of outer joins. Nevertheless, we notice that: (1) this algorithm is based on the assumption that they have knowledge of the data skew, which requires global statistical operations for R and S are required initially, and (2) the cardinality of the intermediate results in $R_{dup} \rightarrowtail\!\!\!\!\bowtie S_{loc}$ will be large because the S_{loc} here is high skewed, and this will bring in significant time-costs, as we analyzed for the duplication-based approach above. We will exam this in our evaluations in Section 5.

2.4 Outer Join Optimization

DER. Xu et al. [8] propose another algorithm called DER (*duplication and efficient redistribution*), which is the state-of-the-art method for optimization of outer joins. The method comprises two phases.

- *Phase 1.* Tuples of R_i at each node i are duplicated to all other computing nodes. Then, a local left outer join between the received tuples of R and S is implemented in parallel at each node. In contrast to a conventional approach, the **ids** of all non-matched rows of R are recorded at this phase.
- *Phase 2.* The recorded ids are redistributed according to their hash values and the non-match join results at each node are organized on this basis[3]. The final output is the union of the inner join results in the first phase and the non-matched results in this phase.

In fact, this method presents a very efficient way to extract non-matched results. Note that the *join* in the first phase of the conventional duplication-based method is an **inner** join rather than an outer join, the reason is that an outer join would bring either redundant or erroneous non-matched output. For example, in a two-node system, if the output of the duplicated tuple (1, a) is (1, a, null) on both nodes, which means there is no match for this tuple in S, we will get duplicate output. In the meantime, if the (1, a, null) appears only on one node and there is a match on the other node (e.g. (1, a, b)), then outputting (1, a, null) will bring in an error.

The conventional approach described above (i.e. duplication-based) to alleviate this problem is by redistributing the intermediate (inner join) results. We can also use another, naive, way to solve this problem by outputting the non-matched results and then redistribute them. Regardless, DER uses a better way, in that each tuple can be indicated by a row-id from the table R, which is redistributed. Consequently, the network communication and the workload can be greatly reduced, and the experimental results presented in [8] demonstrate that the DER algorithm can achieve significant speedups over competing methods.

2.5 A Hybrid Approach - PRPD+DER

Though the work in [8] does not focus on the skew handling problem, it can be predicated that the DER algorithm will be very efficient in this aspect as well. The reason is that DER does not redistribute any intermediate results but only the non-matched ids of R, the size of which is not affected by the skewness of records in S (only by the join selectivity). Moreover, on the condition that R is small, the redistributed data in DER will be also very small even when S is skewed (as the number of non-matched ids is always less than $|R|$ at each node), and redistribution of a small data set will not bring in notable load-imblancing even when such part of data is skewed. However, as DER must broadcast R_i, it is designed to work best for small-large table outer joins.

In contrast with the PRPD algorithm, the broadcast part R_{dup} is typically small, and we expect that integrating DER into PRPD can fix the performance problem as described for $R_{dup} \rightarrowtail\!\!\!\!\bowtie S_{loc}$ previously. Accordingly, this hybrid method can be applied to handle skew in common large-large outer joins. We refer to this approach as PRPD+DER and we will examine its performance in Section 5 as well. For outer joins implemented directly by PRPD (namely the part $R_{dup} \rightarrowtail\!\!\!\!\bowtie S_{loc}$ is implemented by the conventional duplication-based outer join method), we refer to this approach as PRPD+Dup.

Note that the operations of redistribution and duplication are the most commonly used methods in data platforms that are popular in the Cloud such as MapReduce and Spark, and the implementations of PRPD+Dup and PRPD+DER only rely on these two schemes, thus these two methods can be easily implemented in a cloud computing environment. Some examples of applications using PRPD over MapReduce and Pig demonstrating their efficiency on Big Data analytics have already appeared in the literature [24] [25].

3 Our Approach

In this section, we first introduce our *query-based* approach. Then, we analyze how this scheme can directly and efficiently handle data skew in outer joins. We also present

3. The details are that the received ids at each node are counted, if the number of time 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.

an account of the advantages and disadvantages of the approach compared to current techniques in this domain.

3.1 Query-based Outer Joins

As shown in Figure 3, our approach has two different communication patterns - distribution and query, which occur between local and remote nodes. This distinguishes the method from the conventional hash-based and duplication-based outer joins. For a left outer join between R and S on their join attributes a and b, processing can be divided into four phases:

- *Phase 1.* R distribution, which is shown as ① in the figure. This process is very similar as the first phase of the hash-based implementation. Namely, each R_i is partitioned into n chunks, and each tuple is assigned according to the hash value of its key by a hash function $h(k) = k \ mod \ n$. After that, all the chunks R_{ij} will be transferred to the j-th node.
- *Phase 2.* Push query keys, which is shown as ② in the figure, includes the following two steps:
  - The unique keys[4] $\pi_b(S_i)$ of S_i at each node are extracted and then grouped based on their hash values. The tuple assignment is according to the hash function $h(k) = k \ mod \ n$ as well, such that the tuples having the hash value j are put into the jth chunk S_{ij} on each node i.
  - The grouped chunks are transferred to the remote nodes according to the hash values. Namely, a key with hash value j in the chunk $\pi_b(S_{ij})$ is pushed to the j-th node, where these keys are called the *query keys* of the node j in our approach.
- *Phase 3.* Local outer join and push values back, which is shown as ③ in the figure. The detailed process can be divided into the following two steps:
  - The received tuples $R_k = \bigcup_{i=1}^{n} R_{ik}$ from the first step are **left outer joined** with each received key fragment $\pi_b(S_{ik})$ at each node k.
  - The outer join results consist of two parts, the non-matched part and matched part. The matched tuples are sent back to each node i (namely the requester). These tuples are called *returned values*, because we push these values back to the nodes where the query keys originally come from.
- *Phase 4.* After receiving sets of returned values from remote nodes, we **inner join** them with locally kept S_i at each node i. Their output will be the matched part of the final output. The reason is that each query key is extracted from S, and a returned value means that this key exists in R as well. Then, the final outer join results are composed by the non-matched results from the third phase and the output ones in current phase.

4. Here, we use the operator π_b for presenting the duplicate-removing *projection* on the join attribute b of the relation S.

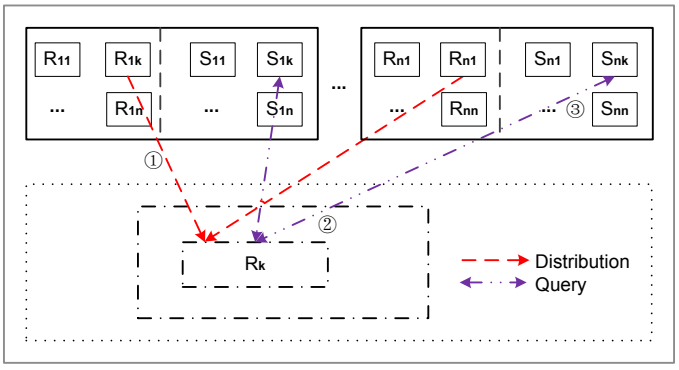

Fig. 3. The query-based approach for outer joins. The dashed square refers to the remote computation nodes and objects.

Compared to our original method as presented in [19], there are two main differences in order to cater for Cloud-based data processing platforms:

- The format of returned values: the values in [19] are in the form of <value> while here we are using <key, value>. The reason is that, through thread-level controls, in [19] we can easily (1) identify the requester of each query key; and (2) keep the ordering of the query keys and returned values and thus we are able to combine them in the form of <key, value> in the fourth phase (e.g. by array indexes). In comparison, in a cloud computing environment, these operations would become much more complex (e.g. by adding location information to keys or values and then join them) and would result in additional cost. In this case, retrieving <key, value> from remote nodes will be a simple and efficient approach (detailed issues and challenges of this pattern see Section 4.2.2).
- Identification of non-matched results: [19] uses a *counter* to distinguish the matched and non-matched results while here we use a simple local left outer join. The reason is that counters in [19] are also used to support the sequential access of received query keys and thus we are able to keep the retrieved values in order. In comparison, such a function is not required here (because we do not need to reconstruct the <key, value> pairs in the *phase 4* any more). Furthermore, we want to rely to the extend possible to existing join implementations in current data platforms (for details see Section 4.2.1).

Even with the aforementioned changes, the outer join approach presented here still follows the "query" pattern. The reason is that the process of transferring keys to remote nodes and retrieving the corresponding values still looks like a query as previous. Therefore, it will inherit the advantages of our previous method [19], such as skew handling etc., as we will present in the following Subsection.

It should be noticed that our method is different from the distributed semijoins (e.g. the implementation proposed in [22]), because we do not broadcast the matched part of R. Instead, we use a fine-grained *query* mechanism to retrieve the data. Namely, we only send back data to specified requesters in the retrieval process. Actually, this brings

about an additional challenge on how to efficiently identify a requester, as we will present in Section 4.2. On the other side, semijoins are studied primarily in two domains: (1) joins in P2P systems, for reducing network communication based on the high selectivity of a join [5]; (2) pre-joins in distributed systems which seek to avoid sending tuples which will not participate in a join (see [13] for a common implementation and [22] for application to the MapReduce framework). In contrast, we apply a refined pattern, the query-based scheme, with full parallelism to outer joins on a distributed architecture and use it for handling skew directly.

3.2 Handling Data Skew

Though S is skewed, we do not transfer any tuples of this relation in our approach. Instead, we just transfer the keys of S. More precisely, we only transfer the **unique** keys of S to a remote node just once per current node (i.e. based on the *projection* operation), irrespective of its popularity.

Assume that there exist such skew tuples (i.e. tuples with join keys that appear frequently), which have the same key k_s, and appear n_s (large number) times in the relation S. Using the conventional hash-based method, all these n_s tuples will be transferred to the $h(k_s)$-th node, which results a hot spot both in communication and the following local join operations. By comparison, our method efficiently addresses this problem in two aspects: (1) each node will receive **only** one key (or maximum n keys if these tuples are distributed on the n nodes), and (2) each query key is treated as the same in the retrieval process.

3.3 Comparison with other Approaches

Compared with the conventional approaches, in addition to efficient handling of data skew, our scheme has two other advantages: (1) network communication can be greatly reduced, because we only transfer the unique keys of S and their corresponding returned values, and (2) computation can be decreased when the join selectivity is high, because retrieved results at each node will be very small, leading to the final local joins to be very light.

Taking a higher level comparison with the *histograms* [13] and the two PRPD-based [12] methods as described in Section 2, there are two other advantages to our approach with respect to handling skew: (1) we do not need any global knowledge of the relations in the presence of skew while [13] and [12] require a global statistic to quantify the data skew, and (2) our approach does not involve redundancy of join (e.g. lookup) operations while the other two do, because each node in our method is just *querying what it is relevant for it*, while [13] and [12] *broadcast*, such that some nodes may receive some tuples what they do not really need. Furthermore, we can directly identify the non-matched results by using a local outer join, while [13] and [12] needs more complex pre-distribution or redistribution operations. In the meantime, although the DER [8] algorithm has done specified optimization for the inner implementation of outer joins, it still needs to redistribute the row-ids. All of these highlight that our approach is more straightforward on processing outer joins and will be easier to implement on current data platforms.

In our method, we have to extract unique keys for S_i at each node, which could be time-costly. Additionally, when the skew is low, the number of query keys will be large, and the two-sided communication could decrease the performance. We assess the balance of these advantages and disadvantages through evaluation with real-world datasets and an appropriate parallel implementation in Section 5.

4 Implementation

We present a detailed implementation of the query-based algorithm using Scala on Spark [17] over a HDFS file system. We have picked Spark as the underlying framework, rather than MapReduce [20], because Spark is becoming more popular[5]. Moreover, various data-parallel applications based on MapReduce can be expressed and executed efficiently using Spark. We compare our method with the conventional hash-based algorithm as well as the two PRPD-based algorithms. The latter two do not provide any code-level information. In the interest of a fair comparison, we have implemented the PRPD+Dup and PRPD+DER algorithms in Scala.

4.1 An overview of Spark

Spark [17] is a parallel computing platform developed by Berkeley AMP Lab. Compared to MapReduce [20], which always requires disk I/O for reloading the data at each iteration, Spark supports the distributed in-memory computing which can improve performance. Moreover, Spark provides high-level APIs allowing users to develop parallel applications very easily and consequently increase programmer productivity. For example, using the functions *map* and *reduce*, we can easily write a MapReduce application by only several lines of codes.

Unlike parallel databases, Spark is very well suited to a Cloud computing environment, since it is elastic: whereas parallel databases have to be carefully tuned to the specification of each node, and adding or removing nodes is a very expensive operation, both from a computational and and administration point of view, Spark is elastic both in terms of storage (through the use of HDFS) and computation. In addition, unlike parallel DBMSs, computation is not tightly coupled to storage. In this light, we implement our methods using Spark, a data processing environment better suited to Cloud computing. In what follows, we briefly introduce two prerequisites - the Hadoop Distributed File System (HDFS) [18] and Resilient Distributed Datasets (RDD).

HDFS is a distributed file system designed to provide high throughput access to large-scale data. Internally, a data file is split into one or more blocks and these blocks are stored across different computing nodes. Compared to other file systems, HDFS is highly fault-tolerant. There are main two reasons for this [26]: (1) data replication across machines in a large cluster ensures very large files can be reliably stored. When part of the data on a node is lost, replicas stored on other nodes will still be accessible; and (2) HDFS has a master/slave architecture and applies heartbeats between

5. As reported by *Cloud Computing Technology Trends in 2015*: "Spark will largely replace MapReduce as the go-to model for big data analytics. MapReduce will still be widely used, but developers will choose Spark over MapReduce whenever possible".

the master node and slaves to check the availability of each node. Currently, HDFS also provides a robust and convenient file system for Spark.

RDD is a central abstraction for Spark. It is a fault-tolerant and parallel data structure that can let user store data in memory or disk and control its partitioning. Spark provides two types of parallel operations on RDDs [17]: *transformations* and *actions*. Transformations, including operations like *map* and *filter*, create a new RDD from the existing one. Actions, such as *reduce* and *collect*, conduct computations on an RDD and return the result to the driver program. Computation in Spark is expressed using functional transformations over RDDs. An RDD cannot be modified, however, a new RDD can be created from HDFS files or constructed by transforming an existing RDD.

Elements in an RDD can be partitioned across machines based on a key in each record. Each partition has an index and operations such as *mapPartitionsWithIndex* will return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition. Spark's partitioning is available on all RDDs of <key, value> pairs. For example, we can use the *partitionBy* on an RDD to hash-partition it by passing a *HashPartitioner* object to the transformation. As data location is important for network communication in distributed join operations. To better understand the implementation given in the following section, we explain a common join operation between two <key, value> RDDs by considering their partitioning:

- If both their *partitioners* are *none*, then a hash-based join between the two RDDs will be performed. Namely, elements in both RDDs are hash-partitioned so that keys that have the same hash value appear on the same node.
- If only one of the RDDs has a *partitioner*, then the elements in this RDD will be not shuffled during the join implementation and the elements in the other RDD will be partitioned based on the known partitioner to perform the join.
- If both RDDs have a *partitioner*, then the join will be based on the first one.

4.2 Implementation with Spark

In this subsection, we first introduce the detailed local join operations we used in our implementation. Then, we present the challenges we meet for distributed outer joins over Spark. Finally, we give our detailed implementation over Spark.

4.2.1 Local Joins

Spark provides several join functions (e.g. *leftOuterJoin*), in which the local join process is based on the commonly used hash-join approach. However, unlike a conventional (hash table) *build* & *probe* approach [27], Spark uses a single `map` data structure to collect all the tuples in both R and S in the form of $(K, (iterable[V_r], iterable[V_s]))$. Namely, all the tuples with the same key will be combined and located on a same bucket of the `map`. Then, based on different join types, Spark uses additional *map* or *filter* operations to formulate the output. For example, Spark will scan each bucket to check whether the $iterable[V_s]$ is *null* or not for a left outer join. If the object is null, then non-matched results will be output based on iteration over $iterable[V_r]$. In contrast, a non-null value means there exist matches between R and S, and the matched results will be the cross product between all the values in $iterable[V_r]$ and $iterable[V_s]$ on that bucket.

As this local join implementation is complex and has to put all the input data in a `Map`, we believe it should be slower than the conventional one, and also slower than our previous approach [19], where a light-weighted *counter* is used to identified the non-matched results. Regardless, for the join process, Spark provides advanced strategies to avoid out of memory issues (e.g. spill data to disk above threshold). In this scenario, for a fair comparison, we adopt the provided local join approach in all our implementations. That is an additional difference with our previous work [19], where we use another data structure (i.e. *map with counters*) for joins.

4.2.2 Challenges

Unlike implementations using thread-level parallelism (e.g. X10 in our previous implementation [19]), Spark focuses on partitioning and does not allow fine-grain control over the location of data. This means that the underlying data is invisible to the programmer, which brings about a challenge for our outer join implementation: In our algorithm, we have to keep all input tuples of S locally, and then retrieve the matched results from remote nodes. This is easily done in X10 [28] (or other frameworks such as MPI or a custom C++ implementation), because we know the location of each subset of S (e.g. by *place* id in X10) and these locations are fixed. This is not straightforward for Spark, since, in the abstraction followed, the programmer has no control over where the subsets are and consequently does not know the destinations (i.e. requesters) for the retrieved results in the third phase of our algorithm as presented in Section 3.1.

A solution is that we mark the tuples in each *partition* with its partition index. For example, a tuple (10, 10) of S in the partition 0 will be marked as (0, (10, 10)). Then, we know that this tuple is in partition 0. However, in the join between retrieved values and locally kept S, the shuffle operation still exists. The reason is that the RDD of S is constructed from read files and does not have a *partitioner*. In this case, we can not guarantee that tuples in partition 0 of S will be always on a same **physical** node during the joins. Namely, if a retrieved tuple is (0, (10, 20)), then we know this tuple should be sent to partition 0 to join with the tuple (0, (10, 10)), but the node corresponding to partition 0 may change. For our algorithm, (10, 10) should be not moved for performance reasons, i.e. dynamic partitioning can have a detrimental effect on performance, due to data movement.

Moreover, even though we can keep all the tuples in S locally during the joins, the final joins will be very time cost. The reason is that: in order to transfer (i.e. in the shuffle process) the retrieved values to their responsible requesters, in the above case, we have to use the partition index as the join key. This would bring heavy computation for each node, as a cross product over S_i and retrieved values will need to be implemented. For example, we can transfer the matched value (0, (10, 20)) to partition 0 based on its join with the tuples at partition 0 over the join key 0. As all the

Algorithm 1 HashedRDD

```
1: class HashedRDD[T: ClassTag](
     var prev: RDD[T], n: Int)
   extends RDD[T](prev) {
2: override val partitioner =
3: if (n > 0) then
4:     Some(new HashPartitioner(n))
5: else
6:     None
7: end if
8: override def getPartitions: Array[Partition] =
     prev.partitions
9: ...
10: }
```

tuples of S is marked with key 0, the cross product between them will commerce, even the actual key of a tuple is not 10. This will be extremely costly when the two inputs are huge. In this meantime, additional local operations like mapping and joins will be required to identify the final results. For example, in the above case, after the join between (0, (10, 10)) of S and retrieved (0, (10, 20)), we have to flatmap the output and conduct the join between (10, 10) and (10, 20). All of these will make the join process very slow.

Based on above analysis, we summarize the two challenges, we meet on the implementation of the proposed query-based joins over Spark, as following:

1. *how can we efficiently keep S_i at each partition i locally and do not move them at all during the whole join process*?
2. *how can we efficiently retrieve matched values and join them with local kept S_i at each partition i on their join keys directly?*

We address these two issues in our real implementation as follows.

4.2.3 Parallel Implementations

For each *action* in Spark, the framework can guarantee partitions in two different RDDs can be co-located, but can not guarantee that they are located on a specific node. We address the above problems by writing a subclass of RDD named *HashedRDD* as shown in Algorithm 1. Namely, we keep the partitions of an RDD and assign a *HashPartitioner* to it according to the parameter n (lines 2-7). In our implementation, we can set the n to the number of partitions for the read-in relation S, then we can easily keep all its tuples locally during the final join. The reason is that, if a RDD has a partitioner already then no shuffle will happen on the included tuples, following the three conditions we described previous. In this case, the first challenge we described above will be addressed. Note that, due to the immutability and dynamic partitioning built into Spark, this comes at no detriment to the fault tolerance properties of the framework.

Followed by the above design, the detailed implementation of our join approach is given in Algorithm 2. Firstly, we read the relation R and S from HDFS to create two RDDs r_pairs and s_paris. According to the number of partitions of s_pairs, we create a new RDD *s_hash* with a hash-partitioner (lines 1-2). Then, we operate on each partition of *s_hash* independently and extract their unique keys. All

Algorithm 2 Query-based Implementation

The input relations R and S are read from underlying HDFS system, results in two RDDs in the form of $< key, value >$: r_pairs and s_paris

```
1: val n = s_pairs.partitions.size
2: val s_hash = new HashedRDD(s_pairs, n)
3: Use mapPartitionsWithIndex(idx, iter) to extract the
   unique keys uni_keys at each partition of s_hash in the
   form of (key, idx)

4: val join1 = r_pairs.leftOuterJoin(uni_keys)
5: val non_match = join1.filter(._2._2 == None)
6: Save non_match on HDFS

7: val match_r = join1.filter(._2._2 != None).map(
   x ⇒ (x._2._2.get, (x._1, y._2._1)))

8: val part = s_hash.partitioner
9: val part_r = match_r.partitionBy(part.get).
   mapPartitions( iter ⇒
     for (tuple ← iter)
     yield (tuple._2._1, tuple._2._2) , true)

10: val matched = s_hash.join(part_r)
11: Save matched on HDFS
```

these keys will be tagged with the partition index where the key is located (line 3), so as to identify the requester afterward. After that, we implement the left outer joins between *r_pairs* and the unique keys. This process entails the redistribution of *r_pairs* and pushing of the unique keys as we stated in the first two phases of our approach in Section 3.1. We do not need to manually implement these two steps as the *join* API in Spark contains the redistribution process of the two join parts already.

As data elements in *join1* are in the form of $(K_r, (V_r, \textbf{Some}(idx)))$, we can easily distinguish the matched and non-matched results using a filter function as shown in lines 5-7. For the non-matched results, we can simply output them as a part of the final results. Note that there will be no redundant or error output here because the left outer join is based on the hash-based implementation as default. For the matched part, we can easily track the requester by the value of the idx. As we can not move each element to the requester directly, we treat each idx as the key i.e. in the form of $(idx, (K_r, V_r))$. To avoid their joins with s_hash based on idx in the final step (i.e. the second challenge as we described), we manually partition the matched part of R (i.e. $match\_r$) using the same paritioner as s_hash and map the tuples in the form of (K_r, V_r), as shown in line 9. Then, all the tuples in s_hash and $match\_r$ will be co-located on their *actual* keys[6], thus we can use the provided *join* directly now and save the remaining part of results (line 10-11). The entire outer join process terminates when all results are output.

It can be seen that setting an initially defined *partitioner*

6. Recall again that *s_hash* has a partitioner defined by *HashedRDD*, thus the data will be locally kept during the joins.

to the relation S is essential for realizing our algorithm in Spark since it allows avoiding the shuffle operation in the final joins. In the meantime, the used tag to indicate where a tuple comes from is also important as it allows us to identified which is the requester. Moreover, to avoid time-cost joins over the tag, assigning a manual partitioner (i.e. the used **partitionBy**) to the matched results is critical for our approach as well. Additionally, from above implementation, we can see a key advantage of Spark in terms of code succinctness and conciseness.

4.3 The PRPD-based Methods over Spark

For our purposes, the implementation of the two algorithms PRPD+Dup and PRPD+DER over Spark are as described in the previous section. Additionally, to identify the skewed tuples and correspondingly partition the tuples, for the PRPD implementation, we add a *key sampling* process on S, wherein we use a *hashmap* counter with two parameters: (1) *sample rate*, namely the ratio of the tuples to be sampled, and (2) *threshold*, namely the number of occurrences of a key in the sample after which the corresponding tuples are considered as skew tuples.

5 Evaluation

In this section, we present an experimental evaluation of our algorithm and compare its performance with the approaches as we described in Section 2.

5.1 Platform

Our evaluation platform is the HRSK-II system of ZIH at TU Dresden. Each node we used has two 6-core Intel Xeon CPU X5660 processors running at 2.80 GHz, resulting in a total of 12 cores per physical node. Each node has 48GB of RAM and 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 1.2.1, Hadoop version 1.2.1, Scala version 2.10.4 and Java version 1.7.0_25.

5.2 Datasets

Our evaluation is based on a join between tables R and S. We fix the cardinality of R to 64M tuples[7] and S to 1B tuples. Because data in warehouses is commonly stored following a column-oriented model, we set the data format to <key, value> pairs, whereas the key is 8-byte integers and each tuple in R has exactly 100 bytes while each in S has 100 bytes on average (i.e. the input relations are about 106GB in total). 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. Joins with such characteristics are common in data warehouses, column-oriented architectures and non-relational stores [14] [27] [29].

As we aim to evaluate the skew handling ability of different algorithms, we add skew on S, the key distribution of which follows a Zipf distribution (details are presented in Table 1, there the **skew** means the the Zipf's factor). There, according to skewness, we classify the skew as no skew, low skew, moderate and high skew[8]. For example, for moderate skewed data, the Zipf factor is 1.1 and the top ten popular keys appear 29% of the time. Moreover, we vary the inner join cardinality (referred to as *selectivity factor*) of R and S by controlling the values in $R.a$ while keeping the sizes of R and S constant. We set selectivity factor to 0%, 50% and 100%, where the 50% means of the 50% tuples of R are kept as the original ones and the rest have their keys changed to their negative values, so that they do not match any key in S. In our tests, we set 50% as the default value. Here, we are using a similar configuration as the one presented in most literature on joins [13] [15] [22] [30].

TABLE 1
Details of the Skewed Datasets

type	dataset	skew	# unique keys	top1	top10
no	1	0	1,073,741,824	0%	0%
low	2	1	50,241,454	5%	14%
moderate	3	1.1	21,281,889	11%	29%
high	4	1.2	8,089,031	18%	45%
	5	1.3	3,090,359	25%	58%

5.3 Setup

For Spark, we set the following system parameters: *spark_worker_memory* and *spark_executor_memory* are set to 40GB and *spark_worker_cores* is to 12. The parameter *sample rate* is set to 10%, and the *threshold* is set to a reasonable number 100, based on preliminary results. Note that, 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 and use a 64MB block size. 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 five measurements. Additionally, 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.

5.4 Runtime

We examined the runtime of four algorithms: the basic hash-based algorithm (referred as Hash), PRPD+Dup, PRPD+DER and our query-based approach (referred to as Query). We implement these tests using 17 nodes in the cluster, one master and 16 slavers (i.e. workers, 192 cores), on the default datasets with varying skew. As the runtime of the first two algorithms becomes very long (more than 2 hours) for the highly skewed datasets, thus we only report their runtime over the datasets with less skew.

General Performance. The detailed results in Figure 4 show that: (1) when S is uniform, Hash performs the best and

7. Throughout this paper, when referring to tuples, M=2^{20} and B=2^{30}.

8. Note that, this classification comes from our experimental results in the following and is just used for simplifying our presentation. For example, in our experiments, Hash algorithm has very long runtime when skew=1.2 and 1.3, then we can simply say that its runtime is long for the high skew condition.

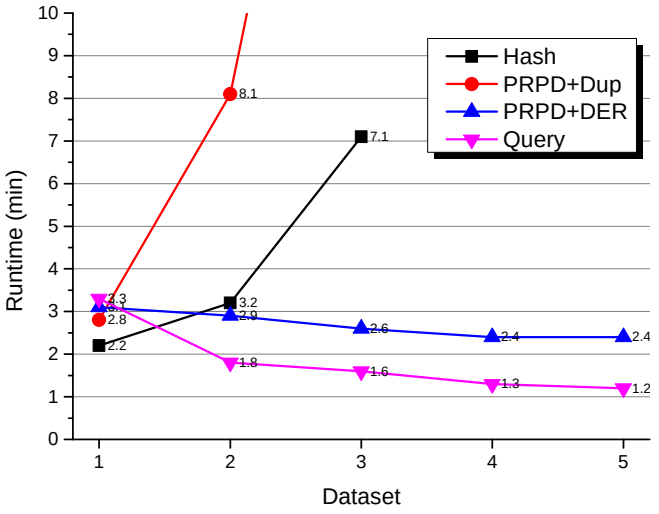

Fig. 4. Runtime of the four algorithms under varying skew (with selectivity factor 50%, 192 cores).

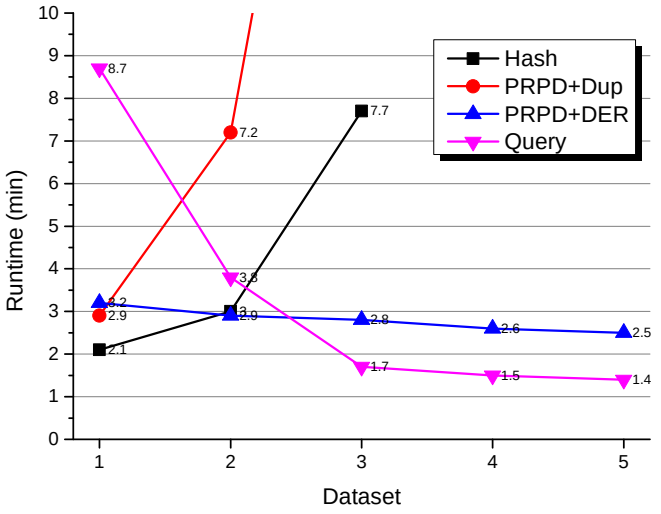

Fig. 5. Runtime of the four algorithms under varying skew (with selectivity factor 100%, 192 cores).

the other three algorithms perform roughly the same; (2) with data skew, PRPD+DER and our Query method become faster and perform better than the other two methods. Moreover importantly, our approach outperforms the other three. In this process, the method PRPD+DER performs very well under varying skew, which confirms our expectation in Section 2.5. At the same time, the PRPD+Dup implementation shows poor performance under skew[9], which is even worse than Hash. This illustrates that skew handling techniques designed for inner joins can not always be applied for outer joins directly.

We also observe that, with increasing data skew, the time cost of Hash increases sharply while our scheme decreases notably, which indicates that our approach has symmetrical behavior regarding skew compared with the commonly used hash-based outer join algorithm. The reason that our method performs better with higher skew is that tuples with popular keys (which also constitute a significant volume of data), do not need to be moved. The higher the skew, the larger proportion of tuples falls under this category, the less the network traffic and the lower the runtime. At the other side of the coin, under no skew, Query results in more network traffic that the hash redistribution-based approach, as we have analyzed in Section 3.3.

In the meantime, although both the PRPD+Dup and PRPD+DER algorithms can be considered as hybrid methods on the basis of the conventional hash-based and duplication-based methods, the runtime of PRPD+Dup increases even more sharply than Hash, while PRPD+DER decreases with skew and shows its robustness against skew. This confirms that state-of-the-art optimization for outer joins can bring in significant performance improvements. Query performs the best under skew conditions, where conventional methods fail. As such, *our method can be considered as a supplement for existing schemes*.

To give an impression of the overall performance of each method, we consider the average runtimes across all datasets. Hash and PRPD+Dup time out for any skew value over 1.1. In practice, this means that *any application exploiting this implementation is would suffer from performance failures when the input presents significant skew*. This is particularly important in a Cloud environment and in a analytical data processing framework like Spark, where it is common that the programmer is not very familiar with the input. The average runtime for PRPD+DER is 2.7 minutes while for Query it is 1.8 minutes, illustrating a significant performance advantage of our method.

Selectivity Experiments. We also examine how join selectivity affects the performance for each algorithm by varying a selectivity factor parameter. For all skew distributions, we choose additional selectivity values of 100% and 0%.

The results for these two conditions are presented in Figure 5 and Figure 6 respectively. Again, they demonstrate that PRPD+DER and Query can efficiently handle data skew under conditions with different join selectivity. Moreover, in conjunction with the results presented in in Figure 4, we notice that the runtime of Hash and PRPD+DER slightly decreases with decreasing the selectivity. The reason for this could be the local computation workload decreases for Hash. In the meantime, though the number of the non-matched results increases with decreasing selectivity, PRPD+DER only needs to redistribute the non-matching row-ids for $R_{dup} \rtimes S_{loc}$, which remains small because R_{dup} is always small.

In comparison, the runtime of PRPD+Dup and Query shows significant variation. For PRPD+Dup, when the selectivity goes to 0%, it performs robustly by varying skew and can even outperform PRPD+DER. The reason could be that PRPD+Dup has to process the intermediate matched join results, the number of which is equal to 0 when the selectivity gets to 0%. In fact, it is these intermediate results that would cause the load balancing problems with higher selectivity. In contrast, PRPD+DER has to redistribute all 64M non-matched ids in this case. For Query, the runtime decreases sharply with decreasing the selectivity in the condition of low skew. The reason is that the retrieved values become

9. Not shown in the Figure is that the runtime is around 23 mins when skew = 1.2.

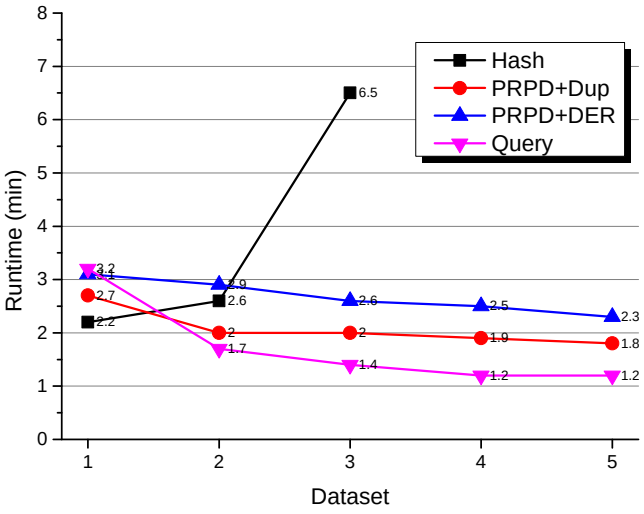

Fig. 6. Runtime of the four algorithms under varying skew (with selectivity factor 0%, 192 cores).

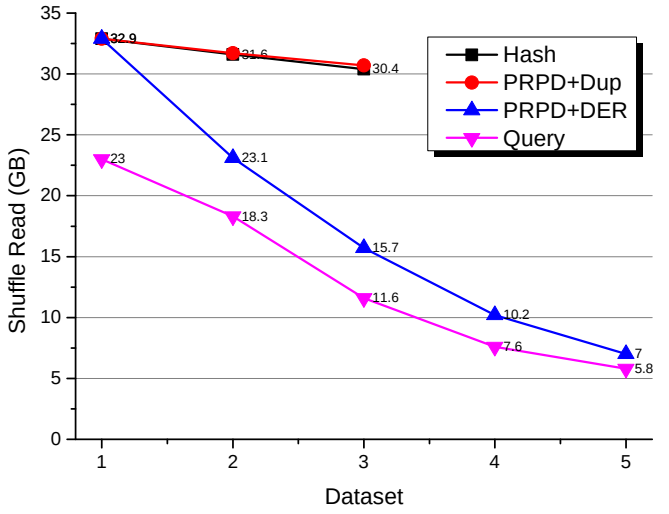

Fig. 7. Inter-machine communication of all four algorithms under varying skew (with selectivity factor 50%, 192 cores).

fewer with decreasing selectivity, reducing both the network communication and local computing. In fact, Query presents the worst performance with 100% selectivity, since it will result in significant communication costs. For high skew, as the queried keys (namely the unique keys) are much fewer anyway, the role of selectivity becomes less important.

Discussion. Combining the results presented above above, we can see that, in the condition of moderate and high skew, our query-based approach is robust and also always outperforms the other three methods. In production environment, the optimizer could pick the best implementation based on the skew of the input (e.g. low skew) so as to minimize runtime.

5.5 Network Communication

Performance regarding communication costs is evaluated by recording the metric *Shuffle Read*, as provided by Spark. It records the data in bytes read from remote executors (physical machines) but not the data read locally. This means that this metric indicates the data transfer around the network during join implementations. The results by varying skew over 16 workers (192 cores) are shown in Figure 7.

We can see that Hash, PRPD+Dup and PRPD+DER transfer the same amount of data when the dataset is uniform. This is reasonable, since all tuples in Hash, PRPD+Dup and PRPD+DER are processed only by redistribution as there is no skew. Moreover, the size of data transfer is around 32.9GB, much less than the input relations, which indicates that there are heavy local reads in the implementation. Query results in much less network communication. The reason could be that that query does not move any tuples in S, but just transfers the unique keys and retrieves matched tuples in R, the size of which is much smaller than the redistribution of all tuples.

With increased skew, the size of transferred data in Hash and PRPD+Dup slightly decreases. The reason could be that reading data locally increases with increasing the data skew. In contrast, PRPD+DER and our method show a significant decrease, demonstrating they can handle skew effectively. Moreover, our method transfers less data than PRPD+DER. This shows that our implementation can reduce network communication more than others under skew.

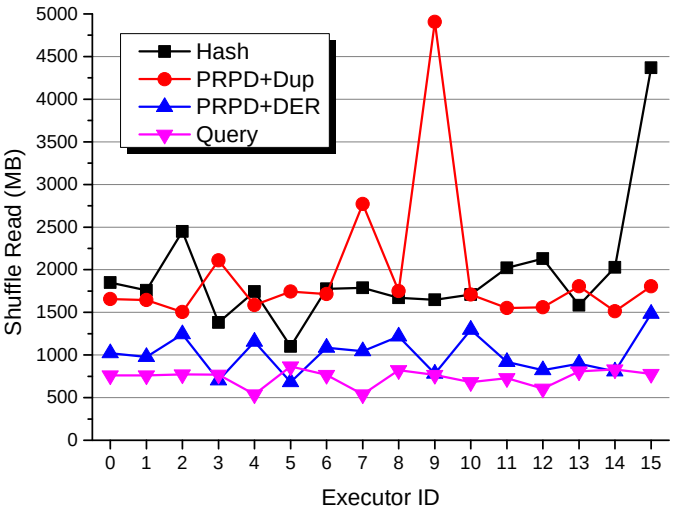

Fig. 8. The retrieved data of each executor for all the four algorithms (with skew=1.1, selectivity factor 50%, 192 cores). In a perfectly load-balanced system, the lines would be flat.

5.6 Load Balancing

We analyze the load balancing properties of each algorithm based on the *Shuffle Read at each executor* metric provided by Spark. This number indicates both the communication and computation time cost. The more data an executor receives, the more time will be spent on data transfer and join operations at this place (because the initial input relations are similar- size partitioned).

Figure 8 presents the results when skew is 1.1. We can obviously see that the data read from remote executors has great variation for the Hash and PRPD+Dup algorithm, which means that they have bad load balancing under skew.

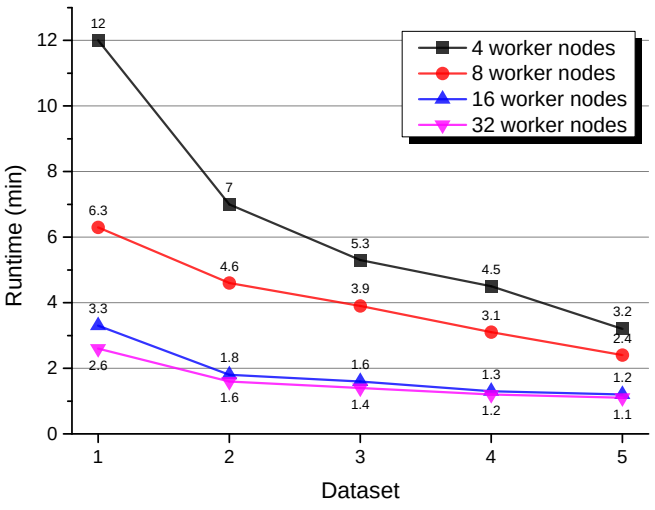

Fig. 9. The runtime the proposed query-based algorithm under different data skew by varying the number of worker nodes (with selectivity factor 50%).

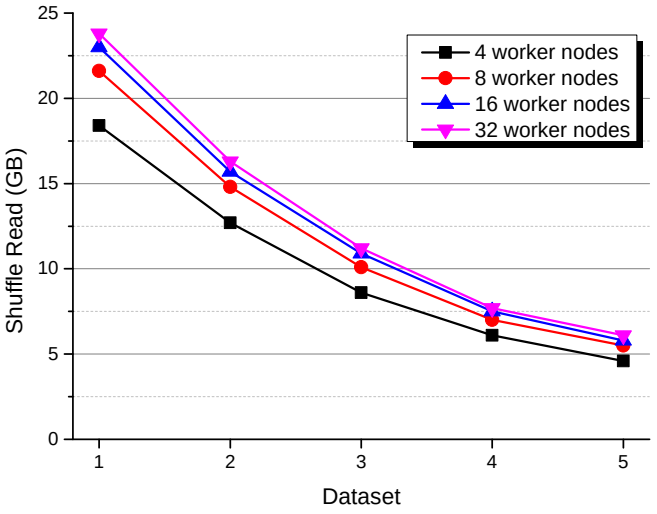

Fig. 10. The inter-machine communication (across all nodes) for the query-based algorithm under varying data skew by varying the number of worker nodes (with selectivity factor 50%).

In comparison, though PRPD+DER shows much improvement for that condition, our query-based approach is still much better than PRPD+DER.

5.7 Scalability

We test the scalability (scale-out) of our implementation by varying number of slaves (workers) under varying skew, from 48 cores (4 nodes) to 384 cores (32 nodes). The results are shown in Figure 9. We can see that the runtime of decreases with increasing the number of workers under varying skew, which means that our implementation generally scales well with the number of workers. This is also supported by recording the network communication as shown in Figure 10. There, the size of the retrieved data across the entire system increases at a reducing ratio to the number of workers. Namely, for larger numbers of workers, the retrieved data at each executor will be halved when doubling the number of workers.

In general, we can see that the benefit of adding more workers (i.e. the scaled speedup) decreases as the runtime becomes lower. We attribute this to platform overhead and coordination costs. In detail, we can observe the achieved runtime speedups under low skew is higher than that under high skew. The reason is that the transferred data is small for the high skew datasets (as shown in Figure 10) and is comparably small for the underlying platform. In the meantime, for different worker configurations, we can see that we can get linear speedup when doubling the number of workers from 8 to 16 and almost linear speedup when doubling the number of workers from 4 to 8. In comparison, for other conditions, the speedups are initially sub-linear but significant and then become very small. Possible reasons could be that (1) the size of retrieved data is increasing with increasing the number of workers, (2) load balancing becomes an issue, or (3) when the number of workers is large (e.g. 32 nodes), the processed data is relative small for the underlying platform, so platform overhead becomes significant. Reasons (1) and (2) are excluded by the results presented in Figure 8 and Figure 10 respectively, so we conjecture that the reason for the decreasing speedup is platform overhead.

6 Related Work

Data skew is a significant problem for multiple communities, such as databases [30], data management [27], data engineering [29] and Web data processing [10]. For example, joins with extreme skew can be found in the Semantic Web field. In [10], the most frequent item in a real-world dataset appeared in 55% of the entries.

The study of parallel joins on shared-memory systems has already achieved significant performance speedups through improvements in architecture at the hardware-level of modern processors [30] [29]. Nevertheless, as applications grow in scale, the associated scalability is bounded by the limit on the number of threads per processor and the availability of specialized hardware predicates. Though GPU computing has become a well-accepted high performance parallel programming paradigm and there are many reports on implementations of parallel joins [31], as in shared-memory architectures, when the data reaches a very large scale, the memory and I/O eventually become the bottleneck.

As we presented previously, various techniques have been proposed for distributed inner joins to handle skew [5] [12] [13]. Often, the assumption is that inner join techniques can be simply applied to outer joins, as identified in [8]. Regardless, as we have shown in our experiments, applying such techniques for outer joins directly may lead to bad performance (i.e. the PRPD algorithm [12]).

Current research on outer joins focuses on join reordering, elimination and view matching [32] [9] [33] [34]. To the best of our knowledge, there are only few approaches designed for skew handling in outer joins. The work [35] proposes an efficient method called OJSO to handle redistribution skew for outer joins. Regardless, they only consider the skew of redistributing non-matched results in a pipeline

execution of outer joins, but do not address the challenge of attribute skew. Moreover, as we presented, though DER [8] can efficiently handle skew, it is designed for small-large table outer joins and not suitable for large datasets. A recent work, [36] introduces an approach called REQC for large table outer joins. However, the implementation is still based on a fine-grain control of data process at a thread level and does not address the challenges in a Cloud computing environment. Though we have shown that the hybrid method PRPD+DER [12] [8] can efficient handle data skew on large outer joins, the proposed query-based method does not rely on the conventional redistribution and duplication operations and performs significantly better under high skew.

With regards to joins in a Cloud computing environment, many new approaches have been proposed to improve performance over the MapReduce platform [37] [38]. However, they have modified the basic MapReduce framework and cannot be readily used by existing platforms like Hadoop. Though the work [22] presents an extensive implementation on joins in MapReduce, they focus on execution profiling and performance evaluation, but not for robust join algorithms. Some work like [20] handles skew using speculative execution but they do not address skew in joins, since the large tasks are not broken up. Several efforts in designing high level query languages on MapReduce, such as Pig [39] and Hive [40] etc., have employed methods (i.e. using statistics and scheduling) to address data skew in outer join processing, however, as discussed in [25], they can not efficient handle skew during join execution. In comparison, as we have showed in our implementation and evaluation that our new method can be applied to such environments to efficiently process data skew.

7 Conclusions

In this paper, we have introduced a new outer join algorithm, query-based outer join, which specifically targets processing outer joins with high skew in a Cloud computing environment. We have presented an implementation of our approach using the Spark framework. Our experimental results over the HDFS file systems show that our approach is scalable and can efficiently handle skew. Compared to the state-of-art PRPD+DER techniques [12] [8], our algorithm is faster and results in less network communication, at least under high skew. In the future, we will combine this method with approaches that partition data according to key skew (e.g. [12]) to further reduce communication and coordination overheads.

Acknowledgments

Long Cheng is supported by the DFG in projects DIAMOND (Emmy Noether grant KR 4381/1-1) and HAEC (CRC 912). The computations were performed on the Bull HPC Cluster at the Center for Information Services and High Performance Computing (ZIH) at TU Dresden.

References

[1] C. B. Walton, A. G. Dale, and 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.

[2] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, Jun. 1992.

[3] K. Imasaki and S. Dandamudi, "An adaptive hash join algorithm on a network of workstations," in *Proc. Int. Parallel and Distributed Processing Symp.*, 2002, p. 8.

[4] X. Zhang, T. Kurc, T. Pan, U. Catalyurek, S. Narayanan, P. Wyckoff, and J. Saltz, "Strategies for using additional resources in parallel hash-based join algorithms," in *Proc. 13th Int. Symp. High Performance Distributed Computing*, 2004, pp. 4–13.

[5] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, Dec. 2000.

[6] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: A survey," *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, Jan. 2012.

[7] M. Atre, "Left bit right: For SPARQL join queries with OPTIONAL patterns (left-outer-joins)," in *Proc. 2015 ACM SIGMOD Int. Conf. Management of Data*, 2015, pp. 1793–1808.

[8] Y. Xu and 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] G. Bhargava, P. Goel, and B. Iyer, "Hypergraph based reorderings of outer join queries with complex predicates," in *ACM SIGMOD Record*, vol. 24, no. 2, 1995, pp. 304–315.

[10] S. Kotoulas, E. Oren, and 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.

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

[12] Y. Xu, P. Kostamaa, X. Zhou, and 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.

[13] M. Al Hajj Hassan and 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.

[14] L. Cheng, S. Kotoulas, T. E. Ward, and 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.

[15] M. A. H. Hassan, M. Bamha, and F. Loulergue, "Handling data-skew effects in join operations using MapReduce," *Procedia Computer Science*, vol. 29, pp. 145–158, 2014.

[16] B. Glavic and G. Alonso, "Perm: Processing provenance and data on the same data model through query rewriting," in *Proc. IEEE 25th Int. Conf. Data Engineering*, 2009, pp. 174–185.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and 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.

[18] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Project*, p. 53, 2008.

[19] L. Cheng, S. Kotoulas, T. Ward, and G. Theodoropoulos, "Efficient handling skew in outer joins on distributed systems," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing*, 2014, pp. 295–304.

[20] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[21] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proc. 2009 ACM SIGMOD Int. Conf. Management of Data*, 2009, pp. 165–178.

[22] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and 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.

[23] N. Bruno, Y. Kwon, and M.-C. Wu, "Advanced join strategies for large-scale distributed computation," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1484–1495, 2014.

[24] W. Liao, T. Wang, H. Li, D. Yang, Z. Qiu, and K. Lei, "An adaptive skew insensitive join algorithm for large scale data analytics," in *Proc. 16th Asia-Pacific Web Conf.*, 2014, pp. 494–502.

[25] S. Kotoulas, J. Urbani, P. Boncz, and 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.
[26] C.-Y. Lin, C.-H. Tsai, C.-P. Lee, and C.-J. Lin, "Large-scale logistic regression and linear support vector machines using Spark," in *Proc. IEEE 3rd Int. Conf. Big Data*, 2014, pp. 519–528.
[27] S. Blanas, Y. Li, and 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.
[28] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 519–538, 2005.
[29] G. A. Cagri Balkesen, Jens Teubner and M. T. Öszu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *Proc. IEEE 29th Int. Conf. Data Engineering*, 2013, pp. 362–373.
[30] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009.
[31] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proc. 2008 ACM SIGMOD Int. Conf. Management of Data*, 2008, pp. 511–524.
[32] G. Hill and A. Ross, "Reducing outer joins," *The VLDB Journal*, vol. 18, no. 3, pp. 599–610, Jun. 2009.
[33] P.-Å. Larson and J. Zhou, "View matching for outer-join views," *The VLDB Journal*, vol. 16, no. 1, pp. 29–53, 2007.
[34] C. Galindo-Legaria and A. Rosenthal, "Outerjoin simplification and reordering for query optimization," *ACM Trans. Database Systems*, vol. 22, no. 1, pp. 43–74, 1997.
[35] Y. Xu and P. Kostamaa, "Efficient outer join data skew handling in parallel DBMS," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1390–1396, Aug. 2009.
[36] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and efficient large-large table outer joins on distributed infrastructures," in *Proc. 20th Int. European Conf. Parallel Processing*, 2014, pp. 258–369.
[37] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proc. 2007 ACM SIGMOD Int. Conf. Management of Data*, 2007, pp. 1029–1040.
[38] D. Jiang, A. Tung, and G. Chen, "Map-Join-Reduce: Toward scalable and efficient data analysis on large clusters," *IEEE Trans. Data Engineering*, vol. 23, no. 9, pp. 1299–1311, 2011.
[39] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of Map-Reduce: the Pig experience," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1414–1425, 2009.
[40] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a Map-Reduce framework," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

Long Cheng is currently a Post-Doctoral Researcher at TU Dresden, Germany. His research interests mainly include Distributed computing, Large-scale data processing, Data management and Semantic web. He has worked at organizations such as Huawei Technologies Germany and IBM Research Ireland. He holds a B.E. from Harbin Institute of Technology, China (2007), M.Sc from Universität Duisburg-Essen, Germany (2010) and Ph.D from National University of Ireland Maynooth, Ireland (2014).

Spyros Kotoulas is a Research Scientist at IBM Research Ireland. His research interests lie in data management for semi-structured data, parallel methods for data intensive processing, Semantic Web, Linked Data, reasoning with Web data, flexible data integration methods, stream processing, peer-to-peer and other distributed systems. He holds a BSc (2004) from the University of Crete as well as an MSc (2006) and a PhD (2009), both from the VU University Amsterdam.