Robust and Skew-resistant Parallel Joins in Shared-Nothing Systems

Long Cheng[1,2,3], Spyros Kotoulas[2], Tomas E Ward[1], Georgios Theodoropoulos[4]

[1] National University of Ireland Maynooth, Ireland [2] IBM Research, Ireland
[3] Technische Universität Dresden, Germany [4] Durham University, UK

long.cheng@tu-dresden.de, spyros.kotoulas@ie.ibm.com, tomas.ward@nuim.ie, theogeorgios@gmail.com

ABSTRACT

The performance of joins in parallel database management systems is critical for data intensive operations such as querying. Since data skew is common in many applications, poorly engineered join operations result in load imbalance and performance bottlenecks. State-of-the-art methods designed to handle this problem offer significant improvements over naive implementations. However, performance could be further improved by removing the dependency on global skew knowledge and broadcasting. In this paper, we propose PRPQ (partial redistribution & partial query), an efficient and robust join algorithm for processing large-scale joins over distributed systems. We present the detailed implementation and a quantitative evaluation of our method. The experimental results demonstrate that the proposed PRPQ algorithm is indeed robust and scalable under a wide range of skew conditions. Specifically, compared to the state-of-art PRPD method, we achieve $16\% - 167\%$ performance improvement and $24\% - 54\%$ less network communication under different join workloads.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*

General Terms

Algorithm, Performance

1. INTRODUCTION

In data-intensive environments such as data warehouses and the Web, efficient execution of query operations is crucial for the overall performance of the system. An essential component of query operations is the *join*, which is widely used in various database management systems (DBMSs). A *join* facilitates the combination of two relations based on a common key. For example, the join between a relation R with attribute a and another relation S with attribute b, is evaluated by the pattern $R \bowtie S$ where $R.a = S.b$. Joins can incur significant costs and hence improving efficiency of this operation can have a significant impact on the performance of database queries [17].

Various parallel join algorithms over distributed architectures have been proposed [22, 10, 24], all of which can be considered variations of two fundamental distributed frameworks: hash-based and duplication-based joins. Such approaches can be broadly decomposed into an initial distribution stage followed by a local join process. This latter process is well studied and techniques such as the sort-merge join and the hash join are commonly used. We have selected the hash-join as the local join process for our analysis. 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 an n-node system, assuming both R and S are in the form of $<$*key, value*$>$ pairs and $|R| < |S|$ in the following.

In the hash-based framework, the basic parallel join algorithm contains four phases, as illustrated in Figure 1: *partition*, *distribution*, *build* and *probe*. In the first phase, the initially partitioned relation R_i and S_i at each node are partitioned into distinct sets R_{ik} and S_{ik} respectively, according to the hash values of their join key attributes. Each of these sets is then distributed to a corresponding remote node in the second phase. These two phases can be considered as a redistribution process, after which, the sequential join of local fragments commence. In the build phase, the relation R_k composed from the redistribution at each node (namely $R_k = \bigcup_{i=1}^{n} R_{ik}$) will be scanned, and an in-memory hash table will be created with the join key attribute. The final probe phase scans each tuple in S_k ($S_k = \bigcup_{i=1}^{n} S_{ik}$) to check whether the join key is in the hash table, and the output will be created in the case of a match.

The duplication-based distributed join framework is shown in Figure 2. The join implementation includes three phases: *duplication*, *build* and *probe*. The first phase just simply duplicates (broadcasts) the tuples of R_i at each node to all other nodes. This means that, after the broadcast, the composed relation R_k at each node will be equal to the full input R, namely, $R_k = \bigcup_{i=1}^{n} R_i = R$. The following two phases are very similar to the final two phases of the hash-based implementation, i.e. that local lookups for S_k will commerce once the in-memory hash table of R_k is created.

Since each phase can be parallelized across nodes, both the schemes above offer the potential for scalability. How-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM'14, November 3–7, 2014, Shanghai, China.
Copyright 2014 ACM 978-1-4503-2598-1/14/11 ...$15.00.
http://dx.doi.org/10.1145/2661829.2661888.

ever there are significant performance issues with both approaches. For the hash-based scheme, while a near linear speedup has been demonstrated under ideal balancing conditions [10] the presence of significant data skew dramatically impacts performance [11] due to node hot spots. Although duplication-based methods can handle skew, the broadcasts of each R_i to all the nodes incurs a heavy time-cost and building a large hash table based on $\bigcup_{i=1}^{n} R_i$ at each node has detrimental impact on performance due to the associated memory and lookup cost [12].

As data skew occurs naturally in various applications [20], it is important for practical data processing systems to perform efficiently in such contexts. In this paper, we introduce the *semijoin-alike joins*, a novel parallel join approach for handling data skew in distributed architectures. Using this design as a basis, we propose an efficient and robust join algorithm referred to as PRPQ (*partial redistribution & partial query*) which is capable of higher performance than current techniques. We implement our algorithm using the parallel programming language X10 [6] and evaluate its performance on an experimental configuration consisting of 192 cores (16 nodes) and datasets of 1 billion tuples. Moreover, we present a quantitative performance comparison with the standard hash-based algorithm as well as the state-of-the-art technique, PRPD, presented in [24].

Our results demonstrate that the proposed PRPQ algorithm is: (a) *robust against data skew*, exhibiting excellent load balancing in the presence of different skew conditions, (b) *scalable*, with speedup increasing with the number of nodes (threads), (c) *highly efficient*, since we can process the join $64M \bowtie 1B$ with high skew in only 10.8 seconds, faster than all methods in the literature, (d) *simple*, since we do not need global operations such as dataset-wide statistical measures to quantify skew and, in fact, algorithm progression on each node is full determined by its local skew, (e) *robust against parameter tuning*, since, unlike the state of the art, it is not overly sensitive to thresholds for skew detection and (f) *novel*, since the algorithm does away with the need for a duplication strategy, an approach which has underpinned nearly all other skew-aware technique to date. Finally, our method is shown to consistently outperform the PRPD algorithm with $16\% - 167\%$ runtime improvement and $24\% - 54\%$ less network communication, depending on the parameters used.

The rest of this paper is organized as follows: In Section 2, we report on related work and details on the state-of-art method. In Section 3, we introduce our PRPQ approach and its differences with existing approaches. The detailed implementation of our algorithm is presented in Section 4. Section 5 provides a quantitative evaluation of our algorithm while Section 6 concludes the paper outlining plans for future work.

2. RELATED WORK

Related work on joins. Data skew is a significant problem for multiple communities, such as databases [17], data management [3], data engineering [5] and Web data processing [20]. Joins with extreme skew can be found in the Semantic Web field. For example, in [20], the most frequent item in a real-world dataset appeared in 55% of the entries.

Research in parallel joins on shared memory systems [17, 2, 5] and GPUs [14, 16] has already achieved significant performance speedups through improvements in architecture at the hardware-level of modern processors. Nevertheless, as applications grow in scale, their associated scalability is limited by either the number of threads available or the system memory and I/O. Efficient parallelisation of joins on distributed memory machines can provide gains in an orthogonal manner to such approaches.

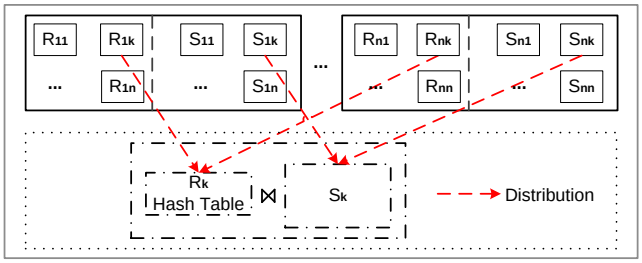

Figure 1: Hash-based distributed join framework. The dashed rectangle refers to the remote computation nodes and objects.

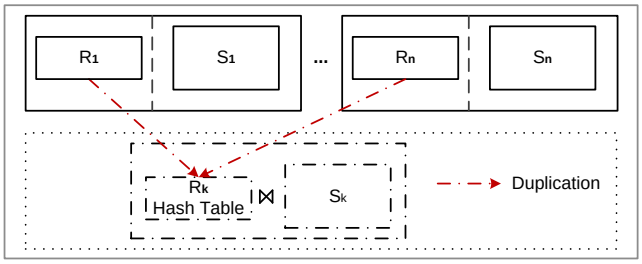

Figure 2: Duplication-based distributed join framework.

The duplication-based approach is seldom adopted, except for some work on its variants [12, 13], which heavily rely on underlying high-speed networks. The hash-based method is widely used and various algorithms have been proposed to handle the join skew and achieve load balancing [23, 19, 15], but all so far rely on the conventional frameworks already described and are suboptimal in terms of computational and communication overhead owing to the inherent performance limitations of the aforementioned frameworks.

Different techniques, such dynamic scheduling [21] and statistics based methods [1], have been applied in the implementation of joins to handle skew. These techniques are based on high-level coordination, bringing associated overhead. For example, to balance the workload, [1] builds *histograms* through the pre-joins of distributed join keys, which incurs a significant time cost. In this work, we focus on capturing the essence performance of a parallel join operation without any additional heavy operations.

Distributed semijoins have been extensively studied, primarily in two domains: (1) joins in peer-to-peer systems for reducing network communication based on the high selectivity of a join [19], such as approaches described in [18]; (2) pre-joins with full parallelism in distributed systems which seek to avoid sending tuples which will not participate in a join, such as the method described in [1], for a common implementation, and in [4], using the MapReduce framework. Compared to these, in our previous work [7, 8, 9], we have employed the semijoin-alike pattern with **full paral-**

lelism as a new **distributed geography** (namely not just a simple join operation) for handling data skew and apply it for parallel inner joins and outer joins **directly**. In this work, we focus on the inner joins (namely *joins*). We introduce the *semijoin-based joins* and the *query-based joins* [7] and analyse their differences. Moreover, we have refined the methods to achieve better robustness and performance on parallel joins.

State-of-the-art PRPD. Xu et al. [24] proposed an algorithm named *partial redistribution & partial duplication* (PRPD), which can be considered as a hybrid method combining both the hash-based and duplication-based join scheme.

For the two input relations[1], they partition S into two parts: (1) locally kept part S_{loc}, the high skew part are kept locally and do not join the redistribution phase, and (2) the redistributed part S_{redis}, the tuples with low frequency key are redistributed as in a common hash-based implementation. The relation R is divided into two parts as well: (1) the duplicated part R_{dup}, the tuples in which contain the keys in S_{loc}, which will be broadcast to all other nodes, and (2) the redistributed part R_{redis}, the remaining part of R that is to be redistributed to a single node as normal. After the duplication and the redistribution operations, the final join can be composed by $R_{redis} \bowtie S_{redis}$ and $R_{dup} \bowtie S_{loc}$ at each node. This method efficiently processes the high skew tuples (keys are highly repetitive). All these tuples of S are not redistributed at all; instead, they just broadcast a small number of tuples contains the same keys from R.

Their experimental results show that PRPD can achieve significant performance when compared with the basic hash approach, in the presence of data skew. Even so, PRPD may still suffer from two major problems. (1) *Global skew*, global operations like statistical calculations or broadcasts for the skew keys at each node are required a priori. As the split of R and S fully relies on the skew keys in S, the final join will fail if any node does not have global knowledge of such keys. (2) *Broadcasting*, the duplicated part from R will lead to significant network communication as the number of such tuples as well as the number of nodes increases, especially if the system is not well tuned.

In comparison, our proposed PRPQ approach only needs to quantify the local skew and does not duplicate data. As a result it is faster and more robust than PRPD in our evaluation presented in Section 5.

3. OUR APPROACHES

In this section, we first introduce two efficient skew-resistant join methods: the *semijoin-based joins* and its variant *query-based joins* [7]. From that basis, we propose a refined method so as to further improve the join performance and robustness in the presence of data skew.

3.1 Basic Approaches

Semijoin-based joins. The approach of semijoin-based distributed joins is shown in Figure 3, where the two communication patterns (*redistribution* and *retrieval*) with full parallelism makes it different from the conventional join approaches and the commonly-used semijoins.

For the join between R and S on their join attributes a and b, the detailed processing can be divided into four steps:

[1]For simplicity, we assume R is uniformly distributed and S is skewed for all of our examples, unless otherwise specified.

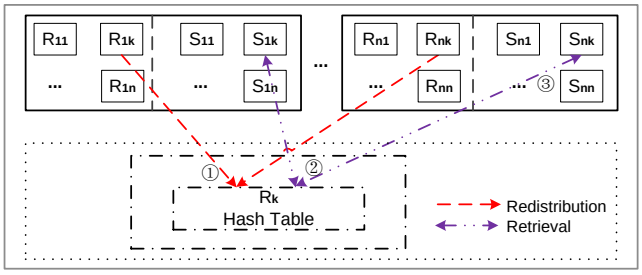

Figure 3: The semijoin-based join approach.

1. tuples in each R_i is hash-redistributed to all the computation nodes according to their join attributes. This process is shown as ① in the figure.

2. the unique keys[2] $\pi_b(S_i)$ of S_i at each node are extracted and transferred to the remote nodes according to their hash values. The process is shown as ② in the figure.

3. each received key fragment $\pi_b(S_{ik})$ at each node k joins with the received tuples $R_k = \bigcup_{i=1}^{n} R_{ik}$ from the first step. Then the matched tuples are sent back to each node i (namely the requester). This process is shown as ③ in the figure.

4. retrieved tuples are joined with the local fragment S_i to formulate the final output.

With the projection operation, any given key of S can possibly move to a remote node just once per current node irrespective of its popularity (remember that, in our case, only S presents skew). Therefore, the semijoin-based method will be very efficient on handling data skew. In the meantime, similarly as the semijoins used in peer-to-peer systems, we only transfer the *unique* keys of S and their corresponding matched tuples, rather than the large number of tuples in S. Therefore the network communication can be potentially reduced during the join processing.

Query-based joins. For a join with a very high selectivity factor, using the semijoin-based method as described, the overlap between the keys in the retrieved tuples of the third step and the transferred keys in the second step will be large, bringing redundancy in terms of network communication. For example, if the factor is 100%, the transferred keys and the keys in retrieved tuples will be exactly the same (both in size and content). To further improve the performance in such a case, we refine the step 3 and step 4 of semijoin-based joins as below:

3'. the received keys $\pi_b(S_{ik})$ are joined with the received R_k at each node k and the retrieved *values* are sent back. If there is no matching keys, the value is set to *null*.

4'. the transferred keys and their respective retrieved *values* are joined with the locally kept S_i for the final output.

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

In this process, we only retrieve *values* (instead of tuples). The reason is that we can always keep the transferred keys and retrieved values in the same sequence (e.g. by array indexes) so that the <*key, value*> pair can be easily identified to compute the final join as described in the fourth step. We call this variant as *query-based joins* because the process of transferring keys to remote nodes and retrieving the corresponding values looks like a query. We also name the distributed pattern used by the two basic algorithms as *semijoin-alike* approach, as it is derived from the conventional semijoin method.

Obviously, the query-based joins can outperform the semijoin-based method when processing joins with high selectivity factors as it removes part of the redundant key transferring. In contrast, it will be slower when the selectivity factor is small, because it needs to fill the unmatched values as *null* and move such useless values to the requester(s) to keep the sequence of the <*key, value*> pairs for the final joins. In fact, using a simple counter to record the ratio of the *nulls* appearing in step 3 of the query-based method can easily guide us to choose a suitable method dynamically during join processing. Namely, when the ratio is high, we can use the query method. Otherwise, we use the basic semijoin approach. In such scenarios, the semijoin-based approach is a sub-instance of the query-based implementation, but sacrificing part of the available system memory (as the transferred keys in the second step need to be kept in memory during join processing when using the query method).

Performance issue. We are more interested in computing joins directly in distributed memory rather than frameworks such as MapReduce [4], which is optimized for on-disk processing. Therefore, the network communication will be critical for performance. In such a case, the above two methods will meet performance issues in the face of data with little skew: Since the number of transferred keys and retrieved tuples (or values) will be huge when processing large joins, and the communication overhead will become unacceptable consequently. To address this issue and achieve robustness and higher performance in the presence of skew, we propose our optimized method in the following.

3.2 The PRPQ Approach

As we focus on join performance over different **distributed join patterns**, and both the two basic methods above use the same *semijoin-alike* geography, for simplification, we only choose the more capable *query-based joins* as a study case in the following[3]. More specially, we only choose the basic query method without any counters to record the ratio of *nulls*, since the time difference between the semijoin-based and query-based method will be very small using our new method, regardless of different selectivity factors (which we explain later).

3.2.1 PRPQ *Algorithm*

Similar to the PRPD algorithm, we divide the skewed input relation into two parts: (1) the low skew part, which is processed by the conventional hash-based method, and (2) the high skew part, using the basic query algorithm as described. In this context, the implementation can be considered as a hybrid approach based on both the hash-based and query-based implementation and thus we call it

[3]As we do not consider memory consumption in this work.

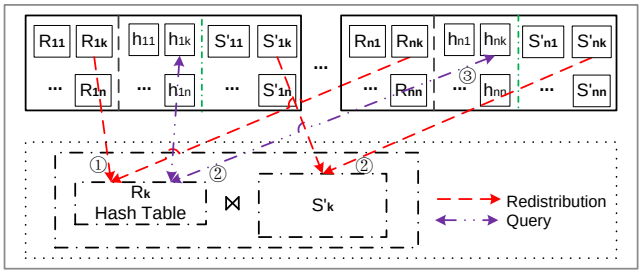

Figure 4: The PRPQ join approach. Only the high skew part of S implements the query operations, and the rest is processed as the basic hash method.

as PRPQ (*partial redistribution & partial query*). Under such a scheme, for the example using previously, the join process is demonstrated in Figure 4, which can be outlined in the following four steps.

R distribution: Tuples in R_i at each node i are redistributed to remote nodes based on the hash values of their attributes a. This process is shown as ① in the figure.

Push query keys: Tuples in S are firstly partitioned into two parts, the high skewed part h (with tuples whose keys appear in the local fragment more times than threshold t) and the remaining part S'. Then, both the S'_i and the unique keys $\pi_b(h_i)$ of h_i at each node i are hash redistributed to the respective computation nodes. The process is shown as ② in the figure, where we refer to the unique keys as *Query keys*.

Return queried values: A local hash table at each node k is built based on the received tuples $\bigcup_{i=1}^{n} R_{ik}$ from the first step. Then the received tuples $\bigcup_{i=1}^{n} S'_{ik}$ are looked up in the local hash table and we output the matched results. Each key fragment $\pi_b(h_{ik})$ is also looked up in the hash table and we formulate the matched values. If there are no matching keys, the value will be set to *null*. All the values are called *returned values*, as all of them are kept in the same order as the received keys and sent back to each requester node i. This process is shown as ③ in the figure.

Result lookup: After receiving sets of returned values from remote nodes, each node i scans these values so as to join with the local kept tuples h_i, based on the sequentially kept keys $\pi_b(h_i)$. In details, we first check whether the value is null. If it is null, we continue scanning the next value. If not, it means that there is a match between R and h. The reason is that each query key is extracted from h, and a non-null returned value means that this key exists in R as well. The final join results are composed by the output of the third step and the matched ones in current step.

3.2.2 *Compared to the* Query-based *Algorithm*

We apply the query scheme only for the high skew tuples and all the low skew tuples are just simply redistributed. When we have large numbers of low skewed tuples, the number of query keys and returned values will still be small. In case of no skew, the method essentially devolves to a hash-based join. Therefore, the PRPQ can efficiently remedy the shortcoming of the basic query algorithm and improve its robustness.

Moreover, inheriting the advantages of the basic query algorithm, PRPQ can also highly reduce the network communication when processing skewed data. The reason is

that none of the highly skewed tuples are distributed, but only their unique keys as well as the corresponding returned values, which are always small. Furthermore, as PRPQ adopts the complementary advantages of both hash-based and query-based implementations, the method should, for any kind of inputs, outperform both algorithms for a suitable threshold t. We will exam this conjecture through our evaluation. In addition, PRPQ has an extra operation, namely quantifying the skew so as to partition the tuples. However, we only need to quantify the **local** skew (namely for each S_i) at each node, which has low computation cost.

Looking back to the semijoin-based joins, if we apply the hybrid idea to that scheme to organize the algorithm PRPS (*partial redistribution & partial semijoin*), the same as PRPQ, its robustness will be highly improved compared to the basic semijoin-based method. Meanwhile, its join performance will be nearly the same as PRPQ. The reason is that the number of transferred keys is relatively small, which brings little difference to the number of retrieved tuples or values in the case of different selectivity factors. Consequently, this will bring little different in terms of time cost on a common cluster with Gigabit Ethernet. That is also why we focus on the performance of PRPQ, when we compared with the state-of-the-art PRPD algorithm [24], in our later evaluation.

3.2.3 *Comparison with* PRPD

Taking a higher level comparison with the PRPD [24] method, there are two main advantages to our approach: (1) we do not need any global knowledge of the relations in the presence of skew, because the skew quantification at each node is totally independent from each other, while [24] requires a global operation to quantify or exchange skew information; and (2) our approach does not involve any redundant join (or lookup) operations because each node in our method is either *distributing* or *query what it needs*, while [24] is *broadcasting*, so that some nodes may receive tuples that they do not need.

In fact, the first advantage leads to our method being more flexible or more efficient in the face of different join workloads, especially for the unevenly distributed ones. Taking an extreme condition for example, for a 10^6-node system, if a key in S follows the linear distribution over the computation nodes (e.g. appearing 10^6 times on the first node, $10^6 - 1$ on the second node etc. and only 1 time on the final node), then *how can we define the global skew using PRPD?* [24] proposes a solution that redistributes the skew tuples evenly to all the nodes before the join. However, this pre-redistribution will generate extra communication costs, while more complex and careful global statistical operations for all tuples of S are required. The authors in [24] do not provide any detailed implementation or experimental details regarding this pre-procesing. Therefore, for PRPD in the following, we do not consider any rebalancing operations for the uneven skew of S but just adopt a general method, namely each node just broadcasts its local skew keys so as to organize the global skew. In contrast to these, simply using a threshold such as $10^6/2$, PRPQ will know that the key is skewed in the first half million nodes and not-skewed for the rest of nodes.

Moreover, in the condition with many mid-skewed tuples, for instance, the relation S_i at some node i contains 1 million unique keys (assuming uniformly distributed) with each key appearing 40000 (or any other numbers) times, *should we consider these keys as skew?* If so, each node under the PRPD scheme has to broadcast the responsible 1 million tuples of R_i to all nodes, which means that each node will receive $10^6 \cdot 10^6 = 10^{12}$ tuples over the 10^6-node system. In comparison, using PRPQ, each node just receives $10^6/10^6 \cdot 10^6 = 10^6$ keys and the responsible 10^6 values. This indicates that PRPQ can further highly reduce the network communication in some cases and potentially improve the join performance over PRPD. We will investigate these detailed performance differences using different workloads in our evaluation in Section 5.

Additionally, the main difference between the PRPQ and PRPD algorithm is in processing skewed tuples, namely using *query*, a duplication-free approach, to replace the conventional *duplication* method. Thus, the extension or theoretical analysis from PRPD [24] can be applied to our approach directly. For example, regarding to the *skewed-skewed* joins[4], similar to the approach taken in PRPD, if R is skewed, the skewed part of R can be used to query the corresponding non-skewed part of S, the skewed part of S can be used to query the corresponding non-skewed part of R, and others would be hash-redistributed, for our PRPQ method.

4. IMPLEMENTATION

We compare our PRPQ algorithm with the hash-based, the basic query [7] and the state-of-art PRPD algorithm [24]. Since the work in [24] does not provide any code-level information, in the interest of a fair comparison, we have also implemented all these methods using the parallel language X10 [6].

As extracting skew tuples at each node is based on local skew quantification, we add in the parameter *threshold* in our implementations, namely the number of occurrences of a key after which the corresponding tuples are considered as skew tuples. We first discuss how we deal with this parameter and then describe the PRPQ implementation.

4.1 Local Skew

There are various ways to measure local skew quickly, such as sampling, scanning etc. However efficient skew measurement does not concern us here and so we just count key occurrences and store them in descending order at each node in a flat file. In each test with parameter t, each node will pre-read the responsible keys (keys appear more then t times) in an `ArrayList` and consider them as the required skew keys (and tuple partitioning is based on these keys). These pre-processes make the performance comparison more fair and meaningful because: (1) The total join performance is very sensitive to the chosen skew keys and operations like sampling cannot guarantee the same set of keys are selected in different implementations, and (2) the extra time cost for skew extraction is removed, so that the focus is on analyzing runtime performance only.

4.2 Parallel Processing

We are interested in high performance distributed memory join algorithms, therefore, we only report the detailed

[4]Recall that *uniform-skewed* joins are the core part of a join, similar as current approches [24, 17, 3, 5], we will focus on such joins in the following implementation and evaluation.

Algorithm 1 R Distribution

```
1: Initialize R_c:array[array[tuple]](n)
2: for tuple ∈ list_of_R do
3:     des ← hash(tuple.key)
4:     R_c(des).add(tuple)
5: end for
6: for i ← 0..(n − 1) do
7:     Push R_c(i) to r_R_c(i)(here) at node i
8: end for
```

Algorithm 2 Push Query Keys

```
1: Initialize T:array[hashmap[key,ArrayList(value)]](n),
       S'_c:array[array[tuple]](n) and skew:hashset[key]()
2: Read the skew keys in skew based on t
3: for tuple ∈ list_of_S do
4:     des ← hash(tuple.key)
5:     if tuple.key ∈ skew then
6:         Add tuple in T(des)
7:     else
8:         Add tuple in S'_c(des)
9:     end if
10: end for
11: for i ← 0..(n − 1) do
12:     Extract keys in T(i) to key_c(here)(i)
13:     Push key_c(here)(i) to remote_key(i)(here),
            S'_c(i) to r_S'_c(i)(here) at the node i
14: end for
```

Algorithm 3 Return Queried Values

```
1: Initialize T':hashmap, value_c:array[value]
2: for i ← 0..(n − 1) do
3:     Put received tuples of r_R_c(here)(i) into T'
4: end for
5: for i ← 0..(n − 1) do
6:     Lookup received r_S'_c(i) in T'
7:     Output join results of non-skew part
8: end for
9: for i ← 0..(n − 1) do
10:    for key ∈ remote_key_c(here)(i) do
11:        if key ∈ T' then
12:            value_c.add(T'.get(key).value)
13:        else
14:            value_c.add(null)
15:        end if
16:    end for
17:    Push value_c(i) to r_value_c(i)(here) at node i
18: end for
```

Algorithm 4 Result Lookup

```
1: for i ← 0..(n − 1) do
2:     for value ∈ r_value_c(here)(i) do
3:         if value ≠ null then
4:             Look corresponding key in T(i)
5:             Output join results of the skew part
6:         end if
7:     end for
8: end for
```

implementation on memory following the four phases as described previously.

p1: We first read all the tuples in an `ArrayList` at each node, and then commence distribution of the relation R. The detailed process is given in Algorithm 1. The array R_c is used to collect the grouped tuples, and its size is initialized to the number of computing nodes n. Then, each thread reads the `ArrayList` of R and groups the tuples according to the hash values of their keys. Next, the grouped items are sent to the corresponding remote nodes. Note that the term *here* means the id of current computing node (core).

p2: The implementation of the second step is given in Algorithm 2. The skew keys are first read into a `hashset` based on the parameter t. Next all the tuples in S will be checked for skew such that `hashmap` collects the skew tuples while the arrays S'_c collects the non-skew tuples. After processing all the tuples, the keys in each hash table will be extracted by an iteration on its `keyset`. These keys will be kept in key_c, the same as S'_c, both are pushed to the assigned remote nodes for further processing.

Both the T and key_c are kept in memory for the subsequent lookup results, as mentioned in Section 3.2. We synchronize the operation here to guarantee the completion of the data transfer at each node before the next phase commences.

p3: The implementation of this phase at each computing node is similar to a sequential hash join. The received tuples and key arrays, representing the distributed R, S' and grouped query keys respectively. For the tuples, all the <*key,value*> pairs of R are placed in the local hash table T', and S' looks up the match in T' to output the join results for the non-skew tuples. Meanwhile, the query keys access T' sequentially to get their values. In this process, if the mapping of a key already exists, its value is retrieved, otherwise, the value is considered as *null*. In both cases, the value of the query key is added into a temporary array so that it can be sent back to the requester(s). The details of the algorithm are given in Algorithm 3.

p4: The join results for the skewed tuples can be looked up after all the values of the query keys have been pushed back. Since the query keys and their respective values are held in order inside arrays, we can easily look up the keys in the corresponding hash tables to organize the join results as shown in Algorithm 4. The entire join process terminates when all individual computation nodes terminate.

5. EXPERIMENTAL EVALUATION

This section presents a comparative quantitative analysis of the proposed algorithm.

5.1 Platform

Our experiments were executed on the *High-performance Systems Research Cluster* in IBM Research Ireland. Each computation unit of this cluster is an iDataPlex node with two 6-core Intel Xeon X5679 processors running at 2.93 GHz, resulting in a total of 12 cores per physical node. Each node has 128GB of RAM and a single 1TB SATA hard-drive and nodes are connected by a Gigabit Ethernet. The operating system is Linux kernel version 2.6.32-220 and the software stack consists of X10 version 2.3 compiling to C++ and gcc version 4.4.6.

5.2 Datasets

The datasets used as benchmarks were chosen to mimic joins in decision support environments. We mainly focus on the most expensive operation in such scenarios: the join between the intermediate relation R (the outcome of various operations on the dimension relations) with a much larger fact relation S [3]. We fix the default cardinality of R to 64M

tuples[5] 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, where both the *key* and *value* are 8-byte integers.

We use similar workloads as used in recently studies on parallel joins [17, 3, 5]. Keys of two input relations R and S follow the foreign key relationship, and we keep the primary keys in R as unique while adding skew to the corresponding foreign keys in S. Meanwhile, 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. For the skewed ones, the **unique** keys of tuples are uniformly distributed and each of them has a match in R[6]. We list the input of S in the Table 1 in bold font indicating default values.

For the Zipf distribution, the skew factor is set to 0 for uniform, 1 for low skew (top ten popular keys appear 14% of the time) and 1.4 for high skew (top ten popular keys appear 68% of the time). For the linear distribution case, we use the function $f(r)$ to describe the key distribution, where r is the rank of a key, according to its popularity. For example, $f(r) = 46341 - r$ means that the most popular key appears 46341 times, the second one appears 46340 times etc. This data set can be considered as low-skewed. Meanwhile, $f(r) = 23170$ is a dataset, in which keys are uniformly distributed but highly repetitive. Both these two datasets contain 1B tuples with 46341 unique keys.

Moreover, to conduct more complete performance comparison in the presence of different workloads, we distribute all the tuples in R evenly to all computing nodes while we use both evenly and sort-range methods for S. The former method guarantees that the number of skewed tuples will be the same on each computation node. In the latter method, all tuples are first sorted according to key popularity, and then partitioned in equal-sized chunks and assigned to each node sequentially. This means that the number of skewed tuples can have great variation between computing nodes.

5.3 Setup

In all experiments, we only count the number of matches, but do not actually output join results. Moreover, for PRPD and PRPQ, we implemented a test series with different t for each data set, as shown in Figure 6. When we present the results in other figures or tables, we always choose the point t with the best achieved run time from Figure 6.

5.4 Runtime

We consider the runtime of the four algorithms[7] the hash-based algorithm (referred to as *Hash*), PRPD [24], PRPQ and the basic query approach [7] (referred to as *Query*). We implement these tests using 16 nodes (192 hardware cores) of the cluster on the default datasets.

[5]Throughout the paper, when referring to tuples, M=2^{20} and B=2^{30}.

[6]The join selectivity factor would be relatively high following the literature, and the query-based part of our PRPQ could get very small profits from this kind of setting (we do not consider output materialization). Regardless, for low selectivity, as mentioned, our PRPQ can be transferred to PRPS if needed. Thus, we do not conduct additional test results for datasets with different selectivity factors in the following.

[7]Recall again that we focus on performance issues over different join patterns. The semijoin-based method will have the same characterization as the query-based method, and the detailed comparisons of such method will be beyond the scope of this paper.

Table 1: Details of the test datasets

S	Key distribution	Partition	Size
Zipf	$skew = \mathbf{0}, \mathbf{1}, \mathbf{1.4}$	**evenly**,	512M,
Linear	$f(r) = 46341 - r$, 23170	sort-range	**1B**, 2B

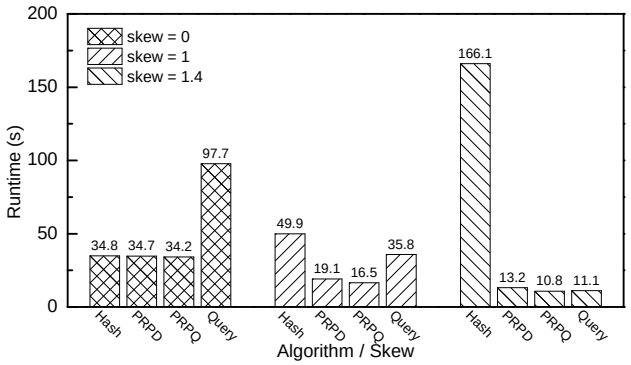

Figure 5: Runtime of the four algorithms.

5.4.1 Performance

The results in Figure 5 illustrate that: (1) when S is uniform, the *Hash*, PRPD and PRPQ algorithms perform nearly the same and much better than the *Query* implementation; (2) with low skew, PRPD and PRPQ is comparatively faster that the other two approaches; and (3) with high skew, *Hash* is the worst while the other three perform much better, demonstrating their capacity to handle skew.

It can be also seen that with increasing data skew, the time cost of *Hash* increases sharply while that of *Query* decreases. This demonstrates that *Query* is more suitable for processing highly skewed datasets. Moreover, PRPD and the PRPQ algorithm change much more smoothly compared to the two basic approaches and their time cost decreases with increasing skew, demonstrating their robustness against skew. Furthermore, for the high skew case, we note that PRPQ outperforms *Query* with threshold $t = 32$, which implies that those tuples with keys appearing less than 32 times perform better in *Hash* than in *Query*.

5.4.2 PRPQ *vs* PRPD

Figure 5 also shows that the best performance achieved by PRPQ is better than PRPD under different skew scenarios. To conduct a more detailed comparison, we implemented a series of tests on different datasets and with different partitioning strategies. The threshold t ranges between values that enable us to always capture the skew keys[8] and present the results in Figure 6 where: (1) *evenly* refers to S being evenly distributed to all the nodes; (2) *range* refers to the sort-range partitioning; (3) *Linear 0* means that S follows

[8]We evaluate all possible configurations for replicating the skewed part. Figure 6 shows all meaningful values for the *threshold*: the maximum frequency in the dataset for 6(e) is 240, so, for a threshold of 241, both approaches degrade to a simple hash-join. It is similar for other settings.

Table 2: Speedup achived by PRPQ over PRPD with varying the size of inputs (using 192 cores)

Skew	1			1.4		
Scale	0.5	1	2	0.5	1	2
Speedup	1.42	1.16	1.20	1.44	1.22	1.48

Table 3: Detailed number of received tuples at each core (millions)

Skew/	0		1		1.4	
Algo.	Max.	Avg.	Max.	Avg.	Max.	Avg.
Hash	5.94	5.94	62.40	5.93	347.76	5.94
PRPD	5.94	5.94	3.53	3.51	1.16	1.13
PRPQ	5.94	5.94	2.65	2.64	0.53	0.52
Query	5.94	5.94	2.12	2.12	0.43	0.43

the linear distribution $f(r) = 23170$ while *Linear 1* refers to $f(r) = 46341 - r$; (4) the first two numbers in the parenthesis indicate the value of t for which the best performance achieved by PRPD and PRPQ respectively while the third one demonstrates the relative speedups of PRPQ over PRPD based on their best runtime.

We can see that, for any given t, PRPQ always performs better than PRPD. Looking at the detailed figures, PRPQ can achieve 16% - 176% performance improvement over PRPD. The maximum achieved speedup of 2.67× happens in the case of *Linear 0 evenly* dataset. This is due to the fact that the number of picked skew keys is always large and this case, the key distribution at each node follows $f(r) = 23170/192 = 121$, namely each key appears 121 times. Thus, when $t < 121$, all the 46341 keys at each node will always be processed as skew keys, which makes the time difference between PRPQ and PRPD large. This also appears in the cases (a),(b) and (e): with a small t at the beginning, a large number of skewed keys leads to a large difference. With increasing t, the difference decreases to almost 0, as the number of picked skew keys becomes smaller and smaller.

Finally, the variations of the results achieved for different t values are only minor for the PRPQ algorithm while those in PRPD change more sharply, demonstrating that our algorithm is less affected by the input parameters (i.e. tuning). Defining the t in a range that achieves better performance would require additional, more complex or costly operations, therefore, we can expect that our algorithm could profit more on performance than PRPD in real applications.

5.4.3 *Cardinality experiments*

We also examine the speedup by varying the cardinalities of the two input relations. For the Zipf distribution, we create data sets in which both relations are half the default size (scale 0.5, namely $32M \bowtie 512M$) and double the size (scale 2, namely $128M \bowtie 2B$). We vary the threshold and record the best achieved runtime. Table 2 shows the results, which demonstrate that our algorithm can achieve higher performance irrespective of the input size.

5.5 Network Communication

Communication costs are evaluated through measuring the number of received tuples at each core. The average

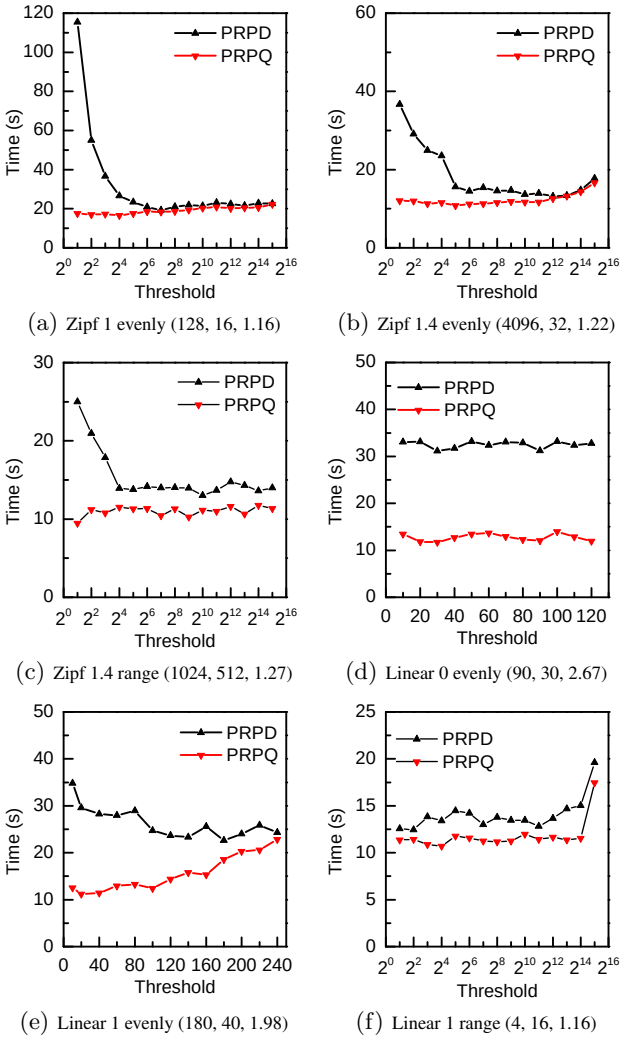

Figure 6: Runtime of PRPD and PRPQ with increasing threshold t (within meaningful range) over different datasets ($64M \bowtie 1B$ with 192 cores).

number of received tuples is presented in Table 3. We can see that all the four algorithms receive the same number of tuples when the dataset is uniform. This is reasonable, since there is no skew and all the tuples of PRPD and PRPQ are only processed by the *partial redistribution* while the number of *Query* keys and returned values (both considered 1/2 tuple) is equal to the number of total tuples in *Query*. With increasing the skew, the number of the received tuples in *Hash* does not change, as all tuples still need to be redistributed. In contrast, the other three methods show a significant decrease, as a large number of skewed tuples are not transferred in PRPD and PRPQ while *Query* groups the skewed tuples and only transfers the unique keys (also responsible values).

For PRPD and PRPQ, we also track the number of received tuples for different threshold t values and present the results in Figure 7. It can be seen that in PRPD that number first decreases and then increases, showing a trade-off between the number of duplicated and redistributed tuples. For PRPQ, the number of received tuples is always increas-

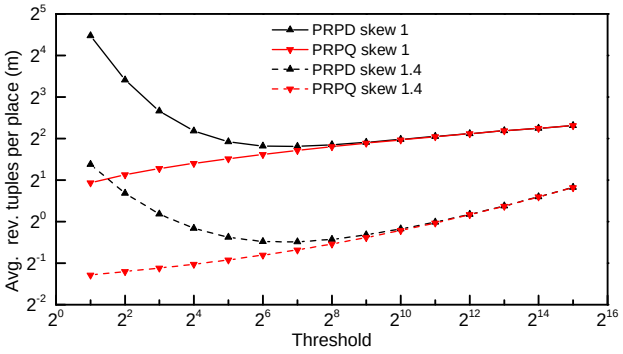

Figure 7: Average number of received tuples at each core by varying the threshold ($64M \bowtie 1B$ with 192 cores).

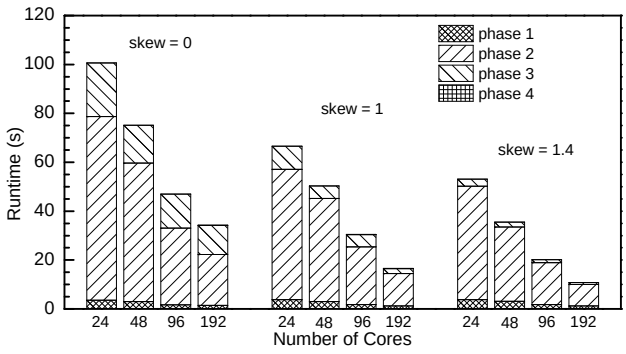

Figure 8: The runtime breakdown of PRPQ under different skews by increasing the cores ($64M \bowtie 1B$).

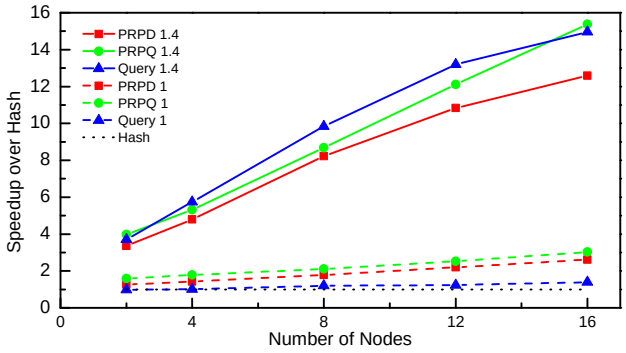

Figure 9: Speedup ratio over the hash algorithm under different skews by varying the number of nodes ($64M \bowtie 1B$).

ing. However, is is less than PRPD for each given t, showing the advantage of *Query* in such a aspect. Combining this with the value where best performance is achieved, t is set to 2^7 and 2^{14} for PRPD, values that are greater than the values of 2^4 and 2^5 for PRPQ respectively. This is the reason why PRPQ clearly transfers less data than PRPD in Table 3, notably $24\% - 54\%$ less under the skews.

5.6 Load-balancing

We also analyze the load balancing properties of each algorithm based on the number of received tuples. The values for the maximum and average number of received tuples at each core are shown in Table 3 as well. We can see that all four algorithms achieve perfect load balancing when the data set is uniform. With increasing skew, the difference between the value of the maximum and the average for *Hash* increases, indicating poor load balancing in the presence of skew. In comparison, PRPD and PRPQ have more tolerance, showing their ability for handling the skew. In the meantime, the basic *Query* algorithm is always balanced, showing its special characteristic on this metric.

5.7 Scalability

We evaluate the scalability of our PRPQ implementation by varying the number of processing cores on the three default datasets, from 24 cores (2 nodes) up to 192. Results are presented in Figure 8, and each phase there is consistent with the implementation explained in Section 4.2.

It can be seen that PRPQ generally scales well under different skews. Notably, the relative speedup achieved between 48 and 96 cores is close to the ideal 2x, which is obviously greater than that between other settings. This could be attributed to the network overhead, in that inter-machine communication is more quickly extended at the beginning. For 192 cores, the data set becomes comparably small for the underlying system and coordination overhead becomes more significant.

Examining the details for each phase, under low skew, phase 2 and 3 scale well and are the dominating factor for the runtime. In the case of high skew, the third phase becomes comparably much smaller and the second phase start to dominate the performance, which decreases with increasing the number of cores. As the second phase mainly focuses on data transfer and the third, on join operations, the network load has a higher impact on the join performance than on the computation workload. For example, in the case of 192 nodes and high skew, the second phase takes 8.812 secs while the third takes only 0.739 secs (note that this includes the time to push back the returned values). Finally, we note that the cost of the fourth phase is extremely small and can therefore be ignored. The reason is that both the size of hash tables in T and the number of looked up elements (returned values) at each core relies only on the number of picked skew tuples, which is small in our tests, resulting in a final lookup cost in the order of tens of milliseconds.

5.8 Comparison with Hash-based Joins

We conclude our analysis with the presentation of speedup using the very popular *Hash* algorithm as a baseline[9], by analyzing the performance improvement achieved for *joins* in each algorithm for different numbers of nodes.

Figure 9 presents the speedup ratio of PRPD, PRPQ and the *Query* algorithm over the basic hash method with increasing number of nodes from 2 (24 cores) to 16 and for skew values 1 and 1.4 respectively. Since results are normalized to the value for the hash method, we present only a single line for it. All three algorithms consistently achieve speedups, demonstrating their ability to handle skew on distributed architectures. Furthermore, their speedup generally increase with increasing the number of nodes as well

[9]Recall again that we do not compare with the duplication-based method here, as it is seldom adopted.

as with increasing the degree of skew for a fixed number of nodes. Furthermore, for high skewed data, PRPQ achieves nearly linear speedup while PRPD and *Query* do not. This can be attributed to the following reasons: (1) For *Query*, the frequency of each element at each core decreases with increasing cores, namely the ratio of low frequency elements increases. This in turn has a negative effect on speedup, as *Query* is not good at processing such low frequent data. (2) In comparison, via the variable t, PRPQ always processes *high-frequency elements using Query and low-frequency elements using Hash*. This presents an optimal way to process the data and achieves better speedups. (3) The broadcast cost increases with increasing the number of nodes, which results in scalability loss in PRPD.

6. CONCLUSIONS

In this paper, we have introduced a new approach for parallel joins, called PRPQ (*partial redistribution & partial query*). The approach has been devised specifically to target joins with skew in shared-nothing architectures. Our experimental results demonstrate the scalability and robustness of PRPQ joins against skew. PRPQ achieves significant speedups over the conventional hash approach in the presence of skew and outperforms the state-of-art PRPD algorithm [24].

Data duplication is widely used in data engineering to reduce data movement and load imbalance. As our algorithm is duplication-free, we anticipate that our proposed method will not only be a supplement to existing schemes on parallel joins to minimize runtime but also for other domains. We intend to apply our approach in the semantic web domain, where workloads present very high skew [20].

Acknowledgments. This work is supported by the Irish Research Council and IBM Research Ireland.

7. REFERENCES

[1] M. Al Hajj Hassan and M. Bamha, "An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems," in *HiPC*, 2009, pp. 350–358.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *PVLDB*, vol. 5, no. 10, pp. 1064–1075, Jun. 2012.

[3] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *SIGMOD*, 2011, pp. 37–48.

[4] 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 *SIGMOD*, 2010, pp. 975–986.

[5] G. A. Cagri Balkesen, Jens Teubner and M. T. Öszu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *ICDE*, 2013, pp. 362–373.

[6] 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," in *OOPSLA*, 2005, pp. 519–538.

[7] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. QbDJ: A novel framework for handling skew in parallel join processing on distributed memory. In *HPCC*, 2013, pp. 1519–1527.

[8] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Efficiently handling skew in outer joins on distributed systems. In *CCGrid*, 2014, pp. 295–304,.

[9] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Robust and efficient large-large table outer joins on distributed infrastructures. In *Euro-Par*, 2014, pp. 258–369.

[10] 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.

[11] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *VLDB*, 1992, pp. 27–40.

[12] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner, "Spinning relations: high-speed networks for distributed join processing," in *DaMoN*, 2009, pp. 27–33.

[13] R. Goncalves and M. Kersten, "The data cyclotron query processing scheme," *ACM Trans. Database Syst.*, vol. 36, no. 4, pp. 27:1–27:35, Dec. 2011.

[14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD*, 2008, pp. 511–524.

[15] K. Imasaki and S. P. Dandamudi. An adaptive hash join algorithm on a network of workstations. In *IPDPS*, 2002.

[16] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "GPU join processing revisited," in *DaMoN* ACM, 2012, pp. 55–62.

[17] 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," *PVLDB*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009.

[18] G. Koloniari and E. Pitoura, "Peer-to-peer management of XML data: issues and research challenges," *ACM SIGMOD Record*, vol. 34, no. 2, pp. 6–17, 2005.

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

[20] S. Kotoulas, E. Oren, and F. van Harmelen, "Mind the data skew: distributed inferencing by speeddating in elastic regions," in *WWW*, 2010, pp. 531–540.

[21] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, 2005, pp. 829–840.

[22] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *SIGMOD*, 1989, pp. 110–121.

[23] C. B. Walton, A. G. Dale, and R. M. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins," in *VLDB*, 1991, pp. 537–548.

[24] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *SIGMOD*, 2008, pp. 1043–1052.