Efficiently Handling Skew in Outer Joins on Distributed Systems

Long Cheng*†, Spyros Kotoulas†, Tomas E Ward*, Georgios Theodoropoulos‡

*National University of Ireland Maynooth, Ireland
†IBM Research, Ireland
‡Durham University, UK
{lcheng, tward}@eeng.nuim.ie, spyros.kotoulas@ie.ibm.com, theogeorgios@gmail.com

***Abstract*—Outer joins are ubiquitous in databases and big data systems. The question of how best to 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. Conventional approaches to this problem such as ones based on hash redistribution often lead to load balancing problems while duplication-based approaches incurs significant overhead in terms of network communication. In this paper, we propose a new algorithm, query with counters (QC), for directly handling skew in outer joins on distributed architectures. We present an efficient implementation of our approach based on the asynchronous partitioned global address space (APGAS) parallel programming model. We evaluate the performance of our approach on a cluster of 192 cores (16 nodes) and datasets of 1 billion tuples with different skew. Experimental results show that our method is scalable and, in cases of high skew, faster than the state-of-the-art.**

***Keywords*-Distributed join; parallel join; outer join; data skew; X10**

I. 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.

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 [1] [2]. 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 [3] [4], as in shared-memory architectures, when the data reaches very large scale, the memory and I/O eventually will become the bottleneck. As a consequence, efficient parallelisation of joins on distributed memory machines becomes increasingly desirable.

Although distributed join algorithms have been widely studied [5] [6] [7], there has been relatively little done on the topic of outer joins a surprising fact given that outer joins are common in complex queries and widely used such as in OLAP applications. 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 [8]. As a result the final join contains not only the matched part but also the non-matched part.

As data skew occurs naturally in various applications, it is important for practical data systems to perform efficiently in such contexts. Similarly to inner joins, the conventional hash-based and duplication-based methods when used for outer joins precipitates performance degradation when data skew is encountered as follows: (1) the former method suffers from poor load balancing; and (2) the latter induces costly network communication. While many algorithms have been designed for handling skew for inner joins [9] [10], 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 [11]. However, as shown in our evaluations later in this manuscript, applying such techniques for outer joins directly may lead to poor performance. Although many systems can convert outer joins to inner joins [12], providing an opportunity then to use inner join techniques, this approach necessitates rewriting mechanisms, which may prove complex and costly. Finally, methods which have been designed specifically for outer joins achieve significant performance improvements [11] over the aforementioned approaches. We will later see though that these state-of-the-art methods are design variations of the two conventional approaches described earlier (namely hash distribution and duplication), making them only applicable in small-large table outer joins.

In this paper, we propose a new approach which we have called *query with counters* (QC), to directly and efficiently handle data skew in massively parallel outer joins on distributed systems. We implement our method using

the asynchronous partitioned global address space (APGAS) model-based programming language - X10 [13]. We conduct a performance evaluation on an experimental configuration consisting of 192 cores (16 nodes) and large datasets of 1 billion tuples with a range of different skews. We summarize the contributions of this paper as below:

- We present a new algorithm (QC) for directly and efficiently handling skew in parallel outer joins.
- We analyze the performance of two state-of-art techniques: (1) PRPD [10], for handling skew in inner joins; and (2) DER [11], for optimizing inner 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, and our experimental results confirm this expectation.
- We compare QC with the PRPD+DER method and show experimentally that our algorithm outperforms PRPD+DER in the presence of high skew. Moreover, the results demonstrate that our method is scalable and maintains load balanced under skew.

The rest of this paper is organized as follows: In Section II, we present background on outer join algorithms and current techniques. We present our *query with counters* algorithm in Section III and its detailed implementation in Section IV. We provide a quantitative evaluation of our algorithm in Section V while we conclude the paper and suggest future work in Section VI.

II. Background

In this section, we describe the two conventional outer join approaches, hash-based and duplication-based outer joins, and discuss their performance. 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 simply focus on this kind 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 $R ⟕ S$.

```
select R.x R.a S.y
from R left outer join S  on R.a = S.b
```

A. Conventional Approaches

Distributed outer joins can be broadly composed as a distribution stage followed by a local join process. This latter process is well studied and techniques such as the sort-merge and hash joins are commonly used. We have selected the hash-join as the local join process for our analysis. Moreover, to capture the core performance of queries, we focus on exploiting the parallelism within a single join operation between two relations R and S. We assume that both relations are in the form of <*key, value*> pairs, where *key* is the join attribution. Additionally, we also assume that R is uniformly distributed and S is skewed.

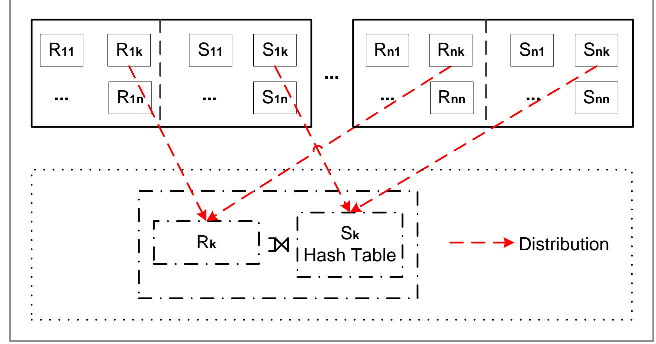

Figure 1. Hash-based Distributed Outer Joins. The initially partitioned relation R_i and S_i at each node are firstly partitioned and distributed to all nodes based on the hash values of join attributes, and then the outer joins are implemented in parallel at each node. The dashed square refers to the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

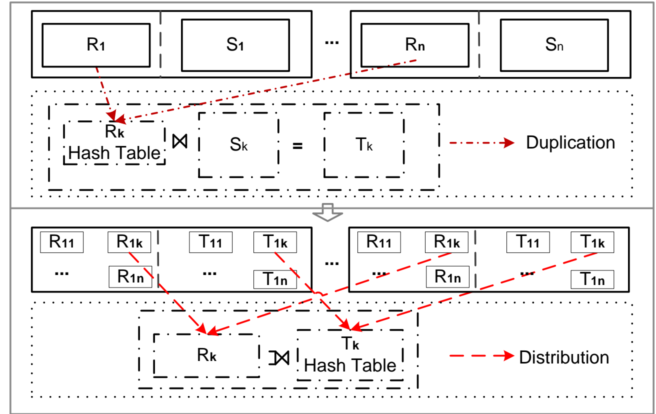

Figure 2. Duplication-based Distributed Outer Joins. The relation R at each node is simply duplicated to all the nodes and then inner joins commence in parallel at each node (above). After that, the intermediate results T implements outer joins with R through the hash-based way (below).

For hash-based approaches, as shown in Figure 1, the parallel outer joins contain four phases, which is similar to the case for inner joins: *partition*, *redistribution*, *build* and *probe*. In the first phase, 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. 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 outer join of local fragments commence. In the build phase, the relation S_k composed by the redistribution at each node (namely $S_k = \bigcup_{i=1}^{n} S_{ik}$) will be scanned, and an in-memory hash table will be created with the join key attributes in the interim. The final probe phase scans each tuple of the relation R_k ($R_k = \bigcup_{i=1}^{n} R_{ik}$) to check whether the join key is in the hash table. The combination of tuples from R and S will be output in the case of a match, otherwise, the output is composed by the tuple from R and the value *null*.

The duplication-based distributed outer join approach is shown in Figure 2. Its implementation includes two main stages, which makes the approach significantly different from that for inner joins. (1) The inner join between R and S. It includes three phases: *replication*, *build* and *probe*. The replication phase just duplicates (broadcast) R_i at each node to all other nodes. This means that, after the replication, the relation R_k will be equal to the full input R, namely, $R_k = \bigcup_{i=1}^{n} R_i = R$. The following two phases are the same as for sequential inner joins, i.e. local lookups for S_k will commence once the in-memory hash table of R_k is created, and consequently outputs the results T_k. (2) The outer join between R and the intermediate join results T_k. This process is the same as the hash-based method described above.

B. Performance Issues

Since 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 nodes, both distributed schemes show the potential for scalability in terms of processing massively parallel outer joins. 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 [6], when the data to be processed has significant skew the performance of such parallel algorithms is dramatically decreased [14]. 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 join attribute will be transferred to the same remote node. When the input is skew, the popular keys will flood into a small number of nodes and cause hot spots. These result in performance bottlenecks due to two reasons: (1) the high time-cost of communication: large numbers of tuples are transferred to hot spots through the network, and (2) load imbalance: a large number of hash table lookups are implemented at hot spots in the probing phase. 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 contrast, duplication-based methods can reduce hot spots, nevertheless, when R is large, the broadcast of every R_i to all the nodes is time-cost heavy and the building of a large hash table based on $\bigcup_{i=1}^{n} R_i$ at each node during the first stage impacts performance as well due to the associated memory- and lookup-cost. Further, even if R is small, the cardinality of the intermediate join results will be large when S is high skewed, which makes the second stage costly and consequently decreases the whole performance.

C. Dealing with Skew

As there are no specific methods for handling skew in single parallel outer join, here, we just discuss two typical techniques used for inner joins - one implements load assignment by *histograms* [15] and the other is the state-of-art PRPD [10] method. We will apply the later approach in outer joins and evaluate its performance later.

Histograms: Hassan et al. [15] employ a method based on distributed histograms, which can be divided into two parts: (1) histograms for R, S and $R \bowtie S$ are built at each node, in either local or global view or both, and (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. [10] propose an algorithm named PRPD (*partial redistribution & partial duplication*) for inner joins. For a single skew relation S (assuming R is uniformly distributed), S is partitioned into two parts: (1) a locally-retained part S_{loc}, which comprises high skew items and which is not involved in the redistribution phase, and (2) the redistributed part S_{redis} which comprises the tuples with low frequency of occurrence and is redistributed using a common hash-based implementation. The relation R is also divided into two parts: (1) the duplicated part R_{dup}, which contain the keys in S_{loc}, which will be broadcast to all other nodes, and (2) the redistributed part R_{redis} - the remaining part of R that is to be redistributed 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 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 a small number of tuples containing the same keys from R are broadcast. The results for this approach show significant speedup in the presence of data skew. In fact, PRPD is a hybrid method combining both the hash-based and duplication-based join scheme. Therefore, we can simply use the $R_{redis} ⟕ S_{redis}$ and $R_{dup} ⟕ S_{loc}$ to replace the corresponding inner joins in the scenarios of outer joins. Even so, 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 $R_{dup} ⟕ S_{loc}$ will be large because the S_{loc} here is high skewed, and this will

bring in significant time-costs. We will exam this in our evaluations in Section V.

D. Outer Joins Optimization

Xu et al. [11] 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 stages. (1) They duplicate R_i to all the nodes and then implement the inner joins. In contrast to the conventional approach, they record the ids of all non-matched rows of R at this stage. (2) They do not redistribute any tuples in the second stage, instead, they just redistribute the recorded ids according to their hash values and then simply organize the non-match join results on that basis. The final output is the union of the inner join results in the first stage and the non-matched ones in the second stage.

In fact, this method shows an efficient way to extract non-matched results. Notice that the *join* in the first stage of the conventional duplication-based method is a inner join but not a outer join, the reason is that the outer join brings either redundant or erroneous non-matched output. For a two-node system for example, if the output of the duplicated tuple {1,a} is {1,a,null} on both nodes, there is no match for this tuple in S and there is a redundant output. In the meantime, if the {1,a,null} appears only on one node, there is a match on the other node, output {1,a,null} will result in error. The conventional approach to alleviate this problem is by redistributing the intermediate 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 more ingenious 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 their experimental results demonstrate that the DER algorithm can achieve significant speedups over competing methods.

As DER must broadcast R_i, it is designed to work best for small-large table outer joins. In this scenario, since R is small, the redistributed part in the second stage will remain small even when S is skew. This is because DER only processes the non-matched part, the number of which is always less than $|R|$ at each node. 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 skew problem as described for $R_{dup} ⟕ S_{loc}$ 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 exam its performance in Section V as well. For outer joins implemented directly by PRPD (namely the part $R_{dup} ⟕ S_{loc}$ is implemented by the conventional duplication-based outer join method), we refer to this approach as PRPD+Dup.

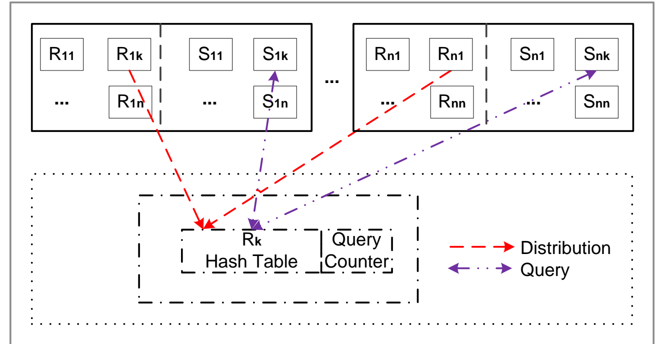

Figure 3. The Query with Counters approach for outer joins. The dashed square refers to the remote computation nodes and objects.

III. Our Approach

In this section, we first introduce our *query with counters* approach and its detailed work flow. Then we analyze how this scheme can *directly* and *efficiently* handle data skew in outer joins. We also present an account of the advantages and disadvantages of the approach compared to current techniques in this domain.

A. The QC Algorithm

Assuming the input relations are R and S, where S is skew, there are N computing nodes, and before the join operations the ith node has a subset of both relations R_i and S_i. 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. We divide its detailed work flow into the following four steps.

R Distribution: The relation R is processed the same way as the hash-based implementations, in that 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 jth node.

Push Query Keys: In this phase, we scan each tuple in the relation S at each node and insert them in a set of local hash tables T_i (the number of hash tables is N). The tuple assignment is according to $h(k) = k \ mod \ N$ as well, such that the tuples having the hash value j are put into the jth hash table T_{ij}. The structure of the hash tables are shown as Figure 4(a). It supports the $1 \rightarrow n$ mappings, such that tuples with the same keys will be stored in the same bucket. After that, iterations on each hash table commence and all keys in each hash table are picked up and kept sequentially in memory. Finally, we push the keys from the hash table T_{ij} to the j node, where these keys are called the *query keys* of the node j in our approach.

Count Matches and Return Queried Values: In this step, we first build a local hash table T'_i with the data structure

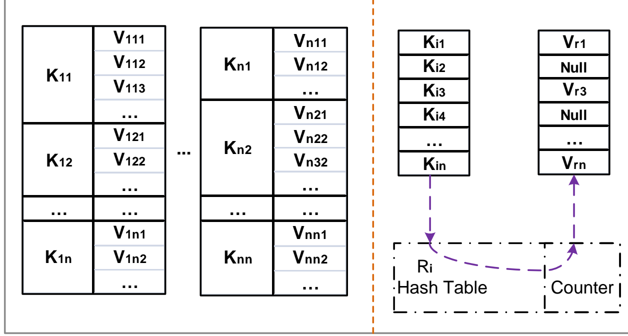

Figure 4. The data structure used in QC algorithm: (a) the local hash tables of S, the tuples are distributed to a set of hash tables according to the hash values of their keys and the tuples with the same keys are inserted into the same bucket (left), and (b) the query keys of a remote node and its corresponding returned values (right).

<*key, (value, counter)*> at each node, in which the *key* and *value* are the received tuples from the first phase while the *counter* is an integer and initialized as 0. After that, we look up each of the received query keys in T_i' and output either a matched value or `Null`. All these values are also kept sequentially as well as the corresponding query keys, and we called them *returned values*, because we push these values back to the nodes where the query keys originally come from after finishing the lookups. The detailed process can be seen in Figure 4(b). (1) If a match exists, the returned value will be the matched value, meanwhile, we also increase the corresponding *counter* by one. (2) If there is no match in R_i, the returned value will be set to `Null`.

Result Lookup: After receiving sets of returned values from remote nodes, we can formulate the final join results. We divide this process into two kinds of lookup: (1) matched result lookup, through scanning the received values at each node. Take a node i for example, for the returned values from jth node, we first check whether the value is null. If the value is null, we continue scanning the next value. If it is not, it means that there is a match between R and S. The reason is that each query key is extracted from S, and a non-null returned value means that this key exists in R as well. Therefore, we look up the corresponding query key in the corresponding hash table T_{ij} and output the join results. (2) non-matched result lookup, through checking whether the *counter* is 0. We iterate all the keys in the hash table T' and check the corresponding *counter*. For each $counter = 0$, we output the non-matched result of the corresponding key directly. The reasons are: (1) the query is based on the hash-based implementation, and (2) the key in R with $counter = 0$ means that this key has never been matched with the query keys, also means it has no match in S. The join operation ends with the output of all the results.

B. Handling Data Skew

Though S is skewed, we do not transfer any tuples of this relation in our framework. Instead, we just transfer the keys of S. More exactly, we only distribute the **unique** keys of S on the basis of $1 \to n$ structure of hash tables T_i.

Assuming that there exist skew tuples, 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 lookup 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 following look up operations.

C. Comparison with other Approaches

In addition to efficient handling of data skew, compared with the conventional approaches, our scheme still has two other advantages: (1) network communication can be highly reduced, because we only transferred the unique keys of S, and their corresponding returned values, and (2) computation can be decreased when S is high skew, because (a) though we have two lookup operations on T_i and T_i', the hash tables in T_i will be very small, (b) skew tuples will be looked up only once instead of checking all of them, and (c) lookup operations for the tuples that are not participating in the join results are removed by just checking whether the returned value is null or not.

Taking a higher level comparison with the *histograms* [15] and the two PRPD-based [10] methods as described in Section II, there are two other advantages to our approach in aspect of handling skew: (1) we do not need any global knowledge of the relations in the presence of skew while [15] and [10] require a global statistic to quantify the skew, and (2) our approach does not involve redundancy of join (or lookup) operations while the other two have, because each node in our method is just *query what I need*, while [15] and [10] have *broadcast behavior*, such that some nodes may receive some tuples what they do not really need. Furthermore, by using a local query counter, we can directly identify the non-matched results while [15] and [10] needs more complex pre-distribution or redistribution operations. In the meantime, although the DER [11] 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.

In our method, we have to build local hash tables 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 as well, and the two-sided communication will 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 V.

IV. Implementation

We present a detailed implementation of the QC algorithm using the X10 framework. 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, therefore, we have also implemented the PRPD+Dup and PRPD+DER algorithms in X10.

A. An overview of X10

X10 [13] is a new multi-paradigm programming language developed by IBM. It supports the asynchronous partitioned global address space (APGAS) model and is specifically designed to increase programmer productivity, while being amenable to programming shared memory and distributed memory supercomputers. It uses the concepts of `place` and `activity` as the kernel notions to exploit parallelism in the available hardware. A place is a logical abstraction of the underlying heterogeneous processing element in the hardware, such as cores in a multi-core architecture, GPUs, or a whole physical machine. Activities are light-weight threads that run on places. X10 schedules activities on places to best utilize the available parallelism. The number of places is constant through the life-time of an X10 program and is initialized at program startup. Activities on the other hand can be forked at program execution time. Forking an activity can be blocking, wherein the parent returns after the forked activity completes execution, or non-blocking, wherein the parent returns instantaneously, after forking an activity. Furthermore, these activities can be forked locally or on a remote place.

X10 provides a data structure called distributed array (`DistArray`) for programming parallel algorithms. One or more elements in the `DistArray` can be mapped to a single place using the concept of points [13]. The following three X10 primitives are critical in understanding the pseudocode given in the following sections:

- `at(p) S`: this construct executes statement `S` at a specific place `p`. The current activity is blocked until `S` finishes executing on `p`.
- `async S`: a child activity is forked by this construct. The current activity returns immediately (non-blocking) after forking `S`.
- `finish S`: this construct is used to block the current activity and wait for all activities forked by `S` to terminate.

There are a number of advantages using the X10 language, and in turn the APGAS model, to implement our algorithm: (1) flexible and efficient scheduling. APGAS, like PGAS, separates tasks from the underlying concurrency model, thereby allowing one to implement an efficient scheduling strategy irrespective of the number of tasks forked using `async`; (2) APGAS, being derived from both MPI and OpenMP programming models, extracts parallelism at both the distributed and single machine hierarchies; (3) the abstract programming model supports the development of succinct code which is easier to debug and maintain.

Algorithm 1 R Distribution

1: `finish async at` $p \in P$ {
2: Initialize *R_c*:array[array[tuple]](N)
3: **for** *tuple* $\in list_of_R$ **do**
4:     *des*=hash(*tuple.key*)
5:     *R_c*$(des).add($*tuple*$)$
6: **end for**
7: **for** $i \leftarrow 0..(N-1)$ **do**
8:     Serialize *R_c*(i) to *ser_R_c*(i)
9:     Push *ser_R_c*(i) to *r_R_c*$(i)(here)$ at place i
10: **end for**
11: }

Algorithm 2 Push Query Keys

1: `finish async at` $p \in P$ {
2: Initialize *T*:array[hashmap[key,ArrayList(value)]](N)
3: **for** *tuple* $\in list_of_S$ **do**
4:     *des*=hash(*tuple.key*);
5:     **if** *tuple.key* $\notin T(des)$ **then**
6:         $T(des).put(tuple.key, tuple.value)$
7:     **else**
8:         $T(des).get(tuple.key).value.add(tuple.value)$
9:     **end if**
10: **end for**
11: **for** $i \leftarrow 0..(N-1)$ **do**
12:     Extract keys in $T(i)$ to *local_key_c*(here)(i)
13:     Serialize *local_key_c*$(here)(i)$ to *ser_key*(i)
14:     Push *ser_key*(i) to *remote_key_c*$(i)(here)$ at place i
15: **end for**
16: }

B. Parallel Implementation

R Distribution: We are interested in high performance distributed memory join algorithms, therefore, we first read all the tuples in `ArrayList` at each node, and commence the distribution of R. The pseudocode of this 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. After that, the grouped items are serialized and sent to the corresponding remote place. This process is done in parallel, and we use the `finish` predicate to guarantee the completion of the tuple transfer in each place before pushing query keys.

Push Query Keys: The detailed implementation of the second step is given in Algorithm 2. A set of `hashmap` is initialized at each place. Each `hashmap` collects tuples of

S according to their hash values. If the key of a tuple has already been in the `hashmap`, then only the value part of the tuple will be added in the hash table. 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 *local_key_c*, and then serialized and pushed to the assigned place for further processing.

Both the `array[hashmap]` and *local_key_c* are `DistArray` objects, which are kept in memory for the subsequent result lookups, as mentioned in Section III-A. The serialization/deserialization process is used only when the push array objects are neither `long`, `int` nor `char`, otherwise we directly deploy the `array.asycCopy` method to transfer the data. We use the *finish* operation in this part to guarantee the completion of the data transfer at each place before the next phase commences.

Count Matches and Return Queried Values: This phase starts after the grouped query keys have been transferred to the appropriate remote places. The implementation at each place is similar to a sequential hash join. The received serialized tuple and key arrays, representing the distributed R and grouped query keys respectively, are deserialized. For the tuples, all the <*key,value*> pairs and the an initialized counter are placed in the local hash table T'. The query keys are used to access this hash table sequentially to get their values. In this process, if the mapping of a key already exists, its value is retrieved and the counter is increment by one, otherwise, the value will be 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). All these processes take place in parallel at each place, and we use the `finish` operation for synchronization. The details of the algorithm are given in Algorithm 3.

Result Lookups: The join results at each place 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 matched join results as shown in Algorithm 4. In the meantime, we scan the *counter* of each key in the local hash table T' and output the corresponding non-matched results. The entire outer join process terminates when all individual activities terminate.

C. The PRPD-based Methods using X10

For our purposes, the X10 implementation of the two algorithms PRPD+Dup and PRPD+DER are as described in the previous section. Additionally, for the part of PRPD implementation we add a *key sampling* process on S to measure the skew, 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.

Algorithm 3 Count Matches and Return Queried Values

1: `finish async at` $p \in P$ {
2: Initialize T':hashmap, *value_c*:array[value]
3: **for** $i \leftarrow 0..(N-1)$ **do**
4:     Deserialize *r_R_c*$(here)(i)$ to tuples
5:     Put all <*tuple.key, (tuple.value, 0)*> into T'
6: **end for**
7: **for** $i \leftarrow 0..(N-1)$ **do**
8:     Deserialize *remote_key_c*$(here)(i)$ to *key_c*
9:     **for** *key* $\in$ *key_c* **do**
10:         **if** *key* $\in T'$ **then**
11:             *value_c*.*add*(T'.*get(key).value*)
12:             T'.*get(key).counter++*
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 place i
18: **end for**
19: }

Algorithm 4 Results Lookups

1: `finish async at` $p \in P$ {
2: **for** $i \leftarrow 0..(N-1)$ **do**
3:     Deserialize *r_value_c*$(here)(i)$ to *local_value_c*
4:     **for** *value* $\in$ *local_value_c* **do**
5:         **if** *value* $\neq$ *null* **then**
6:             Look corresponding *key* in $T(i)$
7:             Output matched results
8:         **end if**
9:     **end for**
10: **end for**
11: **for** $key \in T'$ **do**
12:     **if** T'.*get(key).counter* $== 0$ **then**
13:         Output non-matched results
14:     **end if**
15: **end for**
16: }

V. EVALUATION

In this section, we present the results of our experimental evaluation on a commodity cluster and a comparison with the state-of-the-art.

A. Platform

Our evaluation platform is 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 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.

B. Datasets

The evaluation is implemented on two relations R and S, which are both two-column tables that are populated with random data. The key and payload are both set to 8-byte integers. The cardinality of R is set to 64M tuples[1] and S to 1B tuples. Joins with such characteristics are common in data warehouses, column-oriented architectures and non-relational stores (e.g. see [17]) .

Three key distributions are examined in our tests: uniform, low skew and high skew. Skew is only present on S and follows a Zipf distribution. The skew tuples are evenly distributed on each computing node and the skew factor is set to 1 for the low skew (top ten popular keys appear 14% of the time) and 1.4 for the high skew dataset (top ten popular keys appear 68% of the time). For the two skewed two datasets, the selectivity factor for joins is set to 100% as default.

C. Setup

We set the `X10_NPLACES` to the number of cores and `N_Thread` to 1, namely one place for a single activity, which avoids the overhead of context switching at runtime. The parameter *sample rate* is set to 10%, and the *threshold* is set to a reasonable number 100 based on preliminary results. In all experiments, we only count the number of join results, but do not actually output them. Moreover, we record the mean value based on five measurements and we empty the file system cache between tests to minimize the effects of caching by the operating system.

D. Runtime

We examined the runtime of four algorithms: the basic hash-based algorithm (referred as Hash), PRPD+Dup, PRPD+DER and our QC approach. We implement these tests using 16 nodes (192 cores) of the cluster on the default datasets with different skew.

1) Performance: The results in Figure 5 show that: (1) when S is uniform, the first three algorithms perform nearly the same and much better than our QC implementation; (2) with low skew, PRPD+DER becomes the fastest and our approach is better than the other two methods; and (3) with high skew, our approach outperforms the other three. In this process, the method PRPD+DER performs very well under skew, which confirms our expectation in Section II-D. At the same time, the PRPD+Dup implementation shows the worst poor performance under skew, even worse than Hash, which means that skew handling techniques designed for inner joins can not always be applied for outer joins directly.

[1]Throughout this paper, when referring to tuples, M=2^{20} and B=2^{30}

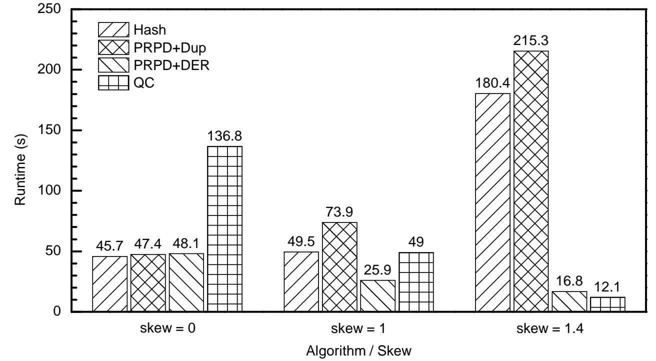

Figure 5. Runtime comparison of the four algorithms under different skews (with selectivity factor 100% over 192 cores).

We also observe that with increasing of data skew, the time cost of Hash increases sharply while our scheme decreases sharply, which indicates that our QC approach has total opposite properties compared with the commonly used hash-based join algorithm. 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. QC performs the best under high skew conditions, where conventional methods fail. As such, our method can be considered as a supplement for the existing schemes. In fact, the optimizer in a system could pick the correct implementation based on the skew of the input so as to minimize runtime.

2) Selectivity Experiments: We also examine how join selectivity affects the performance for each algorithm. For both the low skew and high skew distributions, we created two different S that have the same cardinality as the default dataset but only 50% and 0% of the tuples join with a tuple in R.

The results for the low skew dataset are presented in Figure 6. There, the PRPD+Dup algorithm shows lower runtime with decreasing selectivity, and the runtime of the other three methods does not change or slightly decreases. This is reasonable, (1) PRPD+Dup has to process the intermediate matched join results, the number of which depends on the join selectivity; (2) the transfer and join operations in Hash remain the same with different selectivity; (3) though the number of the non-matched results increases with decreasing selectivity, PRPD+DER only needs to redistribute the non-matching row-ids for $R_{dup} ⟕ S_{loc}$, which remains small because R_{dup} is always small; and (4) the number of operations on *counters* and the final *result lookups* decreases with decreasing selectivity, leading to slightly performance

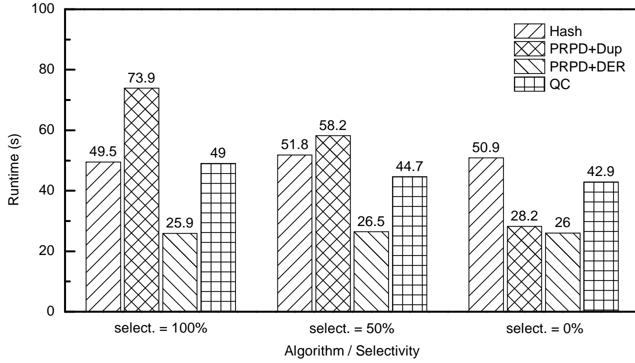

Figure 6. Runtime of the four algorithms under low skew by varying the join selectivity factor ($skew = 1$ over 192 cores).

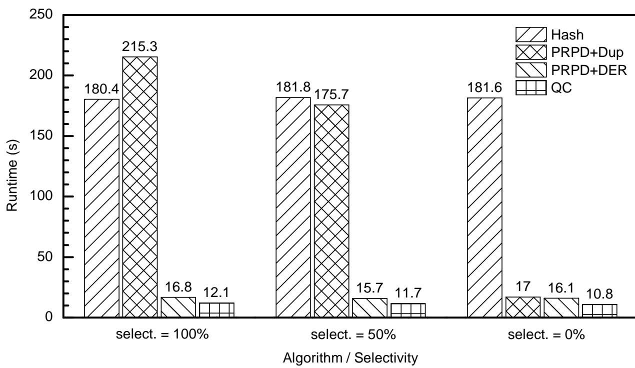

Figure 7. Runtime of the four algorithms under high skew by varying the join selectivity factor ($skew = 1.4$ over 192 cores).

improvement in our QC algorithm. These also appear when the dataset is highly skewed as shown in Figure 7. There, PRPD+Dup changes sharply, showing its sensitivity to the join selectivity. In contrast to this, our QC algorithm is robust and also outperforms the other three methods, demonstrating its strong ability to handling high skew in outer joins again.

E. Network Communication

Performance regarding communication costs is evaluated by measuring the number of received tuples. We implement our test on 192 cores, and collect the received tuples (keys) at each place by inserting counters. The results of the average number of received tuples for each place are shown in Figure 8.

We can see that all the four algorithms receive the same number of tuples 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 and the number of query keys is equal to the number of total keys in our QC algorithm. With the increase in skew, the received tuples in Hash and PRPD+Dup does not change. In contrast, PRPD+DER and our method show a significant decrease, demonstrating they can handle the skew

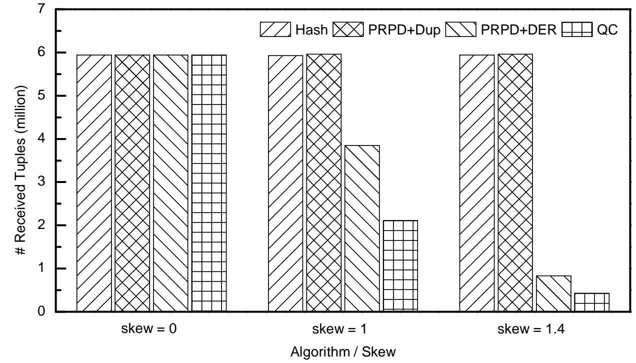

Figure 8. The average number of received tuples (or keys) for each place under different skews (with selectivity factor 100% over 192 cores).

Table I
THE NUMBER OF RECEIVED TUPLES AT EACH PLACE (MILLIONS)

Algo.\skew	**0**		**1**		**1.4**	
	Max.	**Avg.**	**Max.**	**Avg.**	**Max.**	**Avg.**
Hash	5.94	5.94	62.40	5.93	347.78	5.94
PRPD+Dup	5.94	5.94	62.43	5.96	347.80	5.96
PRPD+DER	5.94	5.94	3.95	3.85	0.92	0.84
QC	5.94	5.94	2.12	2.12	0.43	0.43

effectively. In addition, our method transfers much less data than PRPD+DER. All of this shows that our implementation can reduce the network communication more efficiently than other approaches under skew.

F. Load Balancing

We analyze the load balancing of each algorithm based on the metric: *number of received tuples (keys) at each place*. The reason is that this number can indicate both the communication and computation time cost. The more tuples a place receives, the more time will be spent on data transfer and join (build and probe) operations at this place. Though we have to push the values back and implement the *result lookups* in our QC algorithm, the number of returned values is the same as the received keys, which has the same effect for load balancing.

As the place that receives the maximum number of tuples dominates the final runtime, we just report results of the maximum and average numbers for this metric, which is shown in Table I. We can see that all four algorithms achieves perfect load balancing when the dataset is uniform. As the skew increases, the load balancing of the hash-based algorithm and PRPD+Dup becomes much worse. In the meantime, though PRPD+DER shows much improvement for that condition, our QC approach still much better than PRPD+DER.

G. Scalability

We test the scalability of our implementation by varying number of processing cores under skew, from 24 cores (2

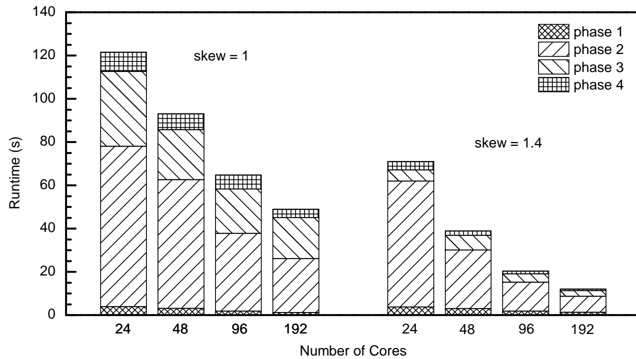

Figure 9. The runtime breakdown of the QC algorithm under skews by varying number of cores (with selectivity factor 100%).

nodes) up to 192. The results are shown in Figure 9. Each phase is consistent with the implementation explained in Section IV-B.

We can see that the implementation generally scales well with the number of cores. Doubling number of cores brings in 1.30x - 1.44x speedup for the low skew dataset and 1.68x - 1.92x for high skew. In detail, phases 1, 2 and 4 scale well and phase 3 is slightly affected by increasing number of cores. The reason would be that the number of received query keys at each place does not obviously change with increasing number of cores in phase 3. This leads in little change in the time cost on the corresponding operations such as hash table lookups and pushing returned values. For example, if there are 2M skew tuples with the same key {1} at each place over 48 cores, then the first place will receive 48 keys {1} in phase 3. Although the number of tuples with key {1} decreases to 1M at each place when using 96 cores, the received query key {1} at the first place will increase to 96, greater than the previously 48. And such increase will be leveraged by the decrease of the non-skewed query keys received at this node. Finally, we note that cost of the fourth phase is very small under high skew. The reason is that both the size of hash tables in T and the number of looked up elements (returned values) at each place is very small in our tests, resulting in a final lookup cost in the order of seconds.

VI. Conclusions

In this paper, we have introduced a new outer joins algorithm, *query with counters*, which specifically targets processing outer joins with high skew. We have presented an implementation of our approach using the X10 system. Our experimental results show that our implementation is scalable and can efficiently handle skew. Compared to the state-of-art PRPD+DER techniques [10] [11], our algorithm is faster with less network communication under high skew.

We will combine our method with approaches that partition data according to key skew in the future, so as to achieve more robustness and higher performance on outer joins. In addition, we will investigate extensions to handle nonuniform network throughput (e.g. outer joins across racks). Finally, we intend to apply our approach in the semantic web domain, where workloads present very high skew [17].

Acknowledgments

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

References

[1] 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.
[2] 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.
[3] 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.
[4] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *DaMoN*, 2012, pp. 55–62.
[5] 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.
[6] 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.
[7] 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 *HPDC*, 2004, pp. 4–13.
[8] G. Bhargava, P. Goel, and B. Iyer, "Hypergraph based reorderings of outer join queries with complex predicates," in *SIGMOD*, 1995, pp. 304–315.
[9] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, Dec. 2000.
[10] 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.
[11] Y. Xu and P. Kostamaa, "A new algorithm for small-large table outer joins in parallel dbms," in *ICDE*, 2010, pp. 1018–1024.
[12] B. Glavic and G. Alonso, "Perm: Processing provenance and data on the same data model through query rewriting," in *ICDE*, 2009, pp. 174–185.
[13] 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.
[14] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *VLDB*, 1992, pp. 27–40.
[15] 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.
[16] 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.
[17] 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.