QbDJ: A Novel Framework for Handling Skew in Parallel Join Processing on Distributed Memory

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*—The performance of parallel distributed data management systems becomes increasingly important with the rise of Big Data. Parallel joins have been widely studied both in the parallel processing and the database communities. Nevertheless, most of the algorithms so far developed do not consider the data skew, which naturally exists in various applications. State of the art methods designed to handle this problem are based on extensions to either of the two prevalent conventional approaches to parallel joins - the hash-based and duplication-based frameworks. In this paper, we introduce a novel parallel join framework, query-based distributed join (QbDJ), for handling data skew on distributed architectures. Further, we present an efficient implementation of the method 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 skews. The results show that the method is scalable, and also runs faster with less network communication compared to state-of-art PRPD approach in [1] under high data skew.**

***Keywords*-Distributed join; parallel join; data skew; high performance; X10**

I. Introduction

The $join$ is a critical operation widely used in various data management systems. It 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 $R.a = S.b$. This operation is associated with a large time cost and improved the efficient implementation of such an operation can have a significant impact in improving the performance of database queries.

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 [2] [3]. Nevertheless, as applications grow in scale, the associated scalability is bounded by the limit on the number of threads available and the availability of specialized hardware predicates. Furthermore, when the data reaches very large scale, memory and I/O eventually become the bottleneck. For this reason, an efficient parallelism of join on multiple machines becomes increasingly desirable.

Various distributed join algorithms have been proposed previously [4] [5] [6], all of which can be considered extensions to either of the two conventional join frameworks: hash-based and duplication-based join.

For hash-based frameworks, as shown in Figure 1, the parallel joins contain three phases: redistribution, build and probe. For the redistribute phase, using the same hash function, both the relation R_i and S_i at each node are partitioned into distinct sets R_{ik} and S_{ik} according to the hash values of their join key attributes, and then each set of tuples is distributed to a corresponding remote node. Next, the sequential join of local fragments commences. In the build phase, assuming $|R| < |S|$, the relation R_k composed by the redistribution at each node (namely $R_k = \cup_{i=1}^{n} R_{ik}$) 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 S_k ($S_k = \cup_{i=1}^{n} S_{ik}$) to check whether the join key is in the hash table, and the output will be created in the case of a match.

The duplicated-based distributed join framework is shown in Figure 2. This join implementation also includes three phases: replication, build and probe. The replication phase just simply 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, $R_k = \cup_{i=1}^{n} R_i = R$. 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 in the above frameworks is implemented in parallel among each computing node, and the number of execution units can be increased by employing new nodes, both distributed schemes show the potential for scalability in terms of processing massively parallel joins. However, though researchers have shown that implementations on the hash-based scheme can achieve near linear speed-up on parallel systems under ideal balancing conditions [5], when the processed data has significant skew, the performance of such parallel algorithms are dramatically decreased [7]. The duplication-based methods can handle the skew, but the

broadcast of each R_i to all the nodes is always time-cost heavy and the building of a large hash table based on $\cup_{i=1}^{n} R_i$ at each node has detrimental impact on performance due to the associated memory- and lookup-cost [8].

As mentioned since data skew occurs naturally in variously applications, it is important for practical data systems to perform efficiently in such contexts and consequently different techniques and algorithms have been proposed to handle the join skew [9] [10] [11] [1], but all of them so far rely on the conventional frameworks already described.

In this paper, we propose a novel framework as an alternative to the conventional approaches, called *query-based distributed join* (QbDJ), for efficiently handling data skew in massively parallel joins on distributed systems. From this basis we develop an efficient distributed join algorithm and implement our parallel joins using the asynchronous partitioned global address space (APGAS) model-based programming language - X10 [12]. We evaluate performance on an experimental configuration consisting of 192 cores (16 nodes) and large datasets of 1 billion tuples with different skews. Moreover, we also have a performance comparison with the basic hash-based implementation as well as the state-of-art technique for efficiently handling data skew - the PRPD method presented in [1].

From these results, our main conclusions are that the proposed framework is: (a) *robust against data skew*, showing excellent load balancing, (b) *scalable*, speedup achieved with increments in the number of nodes (threads), (c) *highly efficient*, since we can process the join $256M \bowtie 1B$ with high skew in only 13 seconds, which is magnitudes faster compared with the conventional hash-based implementation, and also outperforms the state-of-art PRPD algorithm, and (d) *novel*, can be considered as a new approach and alternative to the two conventional frameworks commonly used.

The rest of this paper is organized as follows: In Section II, we present the background to the problem of data skew in parallel joins. We present our *query-based distributed join* framework 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 point to directions for future work in Section VI.

II. Data Skew in Parallel Joins

In this section, we first present an overview of data skew in parallel joins and the challenges it introduces. Then we discuss two common algorithms for shared-nothing architectures that can handle the data skew problem relatively efficiently. As the duplication-based approach is seldom adopted, except for some work on its variants [8] [13], which highly rely on underlying high-speed networks, we just focus on the hash-based frameworks.

A. Skew in the Join

In a common parallel database management system (PDBMS) under the hash-based framework, the redistribution of tuples in relation R and S deeply relies on the hash function, and all the tuples having the same join attribute will be transferred to the same remote node.

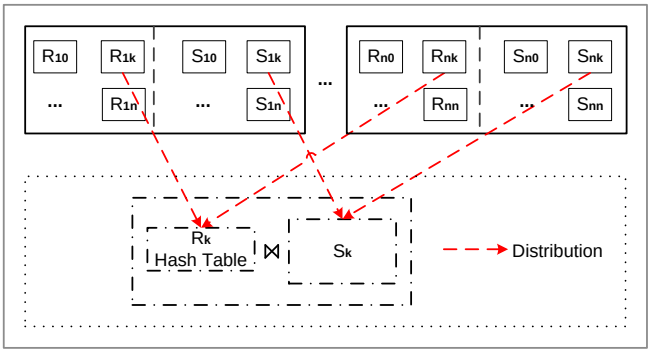

Figure 1. Hash-based Distributed Join Framework. The dashed square refers to the remote computation nodes and objects.

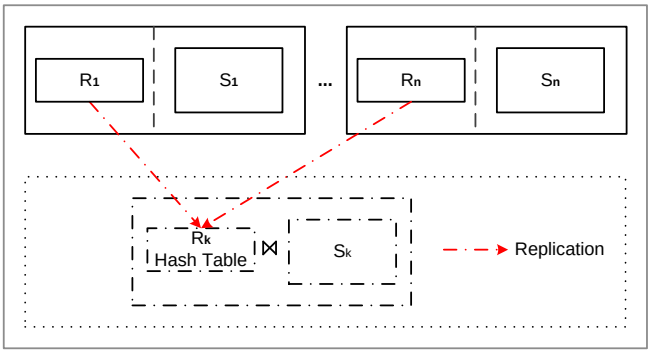

Figure 2. Duplication-based Distributed Join Framework. The dashed square refers to the remote computation nodes and objects.

Assuming tuples in R and S are simply <*key, payload*> (or <*key, value*> in follows) pairs. If there are many tuples in S that have the same key but with different payload, then S is considered as skew data. And during the redistribution of S, all 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 number 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 because the skew tuples will be still distributed to the same nodes. Therefore, an efficient approach to handle this kind of skew becomes critical for the performance of PDBMS.

B. Two Efficient Approaches

Different techniques, such as DHT [14], dynamic scheduling [15] and statistically based methods [16] etc., have been applied in the implementation of joins to handle the skew issue. Here, we discuss two typical methods - one implements load assignment by *histograms* while the other

one is the state-of-art PRPD method. We describe each in turn.

Histograms: Hassan and Bamha et al [16] [17] focus on improving the redistribution plan to process data skew. They mainly use distributed histograms in their method, 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 redistribute plan to balance the workload for each node is formulated.

Their experimental results show that this method is efficient and scalable in presence of data skew, nevertheless, 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 communications, 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.

state-of-art PRPD: Xu *et al* [1] proposes 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 a single skew relation S (assuming R is uniform distributed), they partitioning S into two parts: (1) local 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 is redistributed as 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 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, they just broadcast a small number of tuples contains the same keys from R. Their results show that this algorithm can achieve significant speeds up in the presence of data skew. Even so, we notice that: (1) their implementation is based on the assumption that they have knowledge of the data skew, which means that global statistical operations for R and S are required initially, and (2) the tuples of the duplicated part are processed by broadcasting, which is good for load-balancing but could bring in significant time-cost.

III. Query-based Distributed Join

In this section, we first introduce our *query-based distributed join* framework and its detailed work flow. Then we analyze how this scheme can efficiently handle data skew. Furthermore, we also discuss its advantages and disadvantages compared with current approaches.

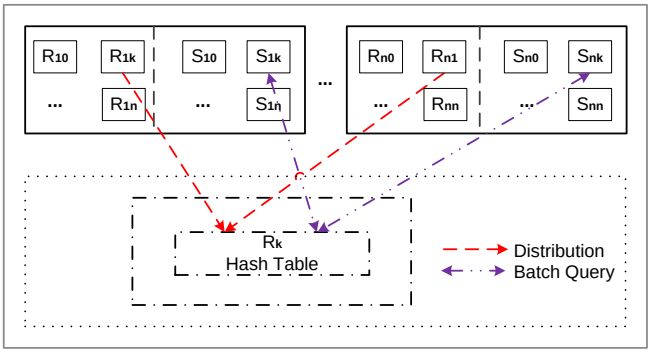

Figure 3. Query-based Distributed Join Framework. The dashed square refers to the remote computation nodes and objects.

A. Framework

Assuming the input relations are R and S, where $|R| < |S|$ and 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 framework has two different communication patterns - distribution and query, between local and remote nodes, which obviously makes it different from the conventional hash-based and duplication-based frameworks. 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 $h_1(k) = k \ mod \ N$. After that, all the chunks R_{ij} will be transferred to the jth node. There are two reasons to do so: (1) R is relatively small such that we can afford the distribution cost, and (2) R can be considered as a uniform distributed data set, as adding skew to the relation R would violate the primary key constraint [18].

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 $h_1(k) = k \ mod \ N$ 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.

Return Queried Values: In this step, we first build a local hash table T_i' at each node, based on the received tuples from the first phase. After that, we look up each

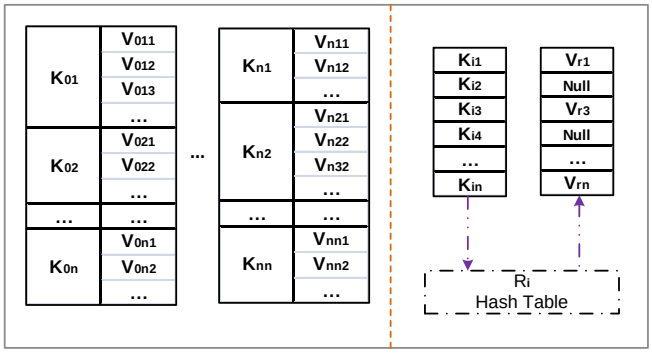

Figure 4. The data structure used in query-based distributed join: (a) the local hash tables of S, the tuples are distributed to a set of hash tables according to the hash values of their keys and the tuples with the same keys are inserted into the same bucket (left), and (b) the query keys of a remote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

of the received query keys in T_i' and output the matched values. If there is no matching keys, the value will be set to `Null`. All these values are also kept sequentially as well as the corresponding query keys. This process can be seen in Figure 4(b), where all the values are called *returned values*, because we push these values back to the nodes where the query keys originally come from after finishing the lookups.

Result Loopups: After receiving sets of returned values from remote nodes, we start to scan these 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. 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 $1 \rightarrow n$ structure of hash tables T_i.

Assuming that there exists 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 $h_1(k_s)$-th node, which results a hot spot both in communication and the following probing operations. By comparison, our framework 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 frameworks, our scheme still has two other advantages: (1) network communication can be highly reduced, because we only transferred parts of keys in 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 [16] and PRPD [1] method described in § II-B there are two other advantages to our approach: (1) we do not need any global knowledge of the relations in the presence of skew while [16] and [1] 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 [16] and [1] have "*Broadcast behavior*", such that some nodes may receive some tuples what they do not really need.

In our framework, 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 uncompetitive 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 § V.

IV. Implementation

In this section, we present a detailed implementation of the proposed query-based distributed joins using the X10 framework. We compare our algorithm with the hash-based and the state-of-art PRPD algorithms [1]; since the latter does not provide any code-level information, we have also implemented PRPD in X10.

A. An overview of X10

X10 [12] is a new multi-paradigm programming language supporting 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

Algorithm 1 R Distribution

```
1: finish async at p ∈ P {
2: Initialize R_c:array[array[tuple]](N)
3: for tuple ∈ list_of_R do
4:     des=hash(tuple.key)
5:     R_c(des).add(tuple)
6: end for
7: for i ← 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: }
```

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 [12]. 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.

B. Parallel Join Processing

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 then start to distribute the relation 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.

Algorithm 2 Push Query Keys

```
1: finish async at p ∈ P {
2: Initialize T:array[hashmap[key,ArrayList(value)]](N)
3: for tuple ∈ list_of_S do
4:     des=hash(tuple.key);
5:     if tuple.key ∉ 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 ← 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: }
```

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 § 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.

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 are placed in the local hash table *T'*. The 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, 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

Algorithm 3 Return Queried Values

```
1: finish async at p ∈ P {
2: Initialize T':hashmap, value_c:array[value]
3: for i ← 0..(N − 1) do
4:     Deserialize r_R_c(here)(i) to tuples
5:     Put all <tuple.key,tuple.value> into T'
6: end for
7: for i ← 0..(N − 1) do
8:     Deserialize remote_key_c(here)(i) to key_c
9:     for key ∈ key_c do
10:        if key ∈ T' then
11:            value_c.add(T'.get(key).value)
12:        else
13:            value_c.add(null)
14:        end if
15:    end for
16:    Push value_c(i) to r_value_c(i)(here) at place i
17: end for
18: }
```

Algorithm 4 Results Lookups

```
1: finish async at p ∈ P {
2: for i ← 0..(N − 1) do
3:     Deserialize r_value_c(here)(i) to local_value_c
4:     for value ∈ local_value_c do
5:         if value ≠ null then
6:             Look corresponding key in T(i)
7:             Output join results
8:         end if
9:     end for
10: end for
11: }
```

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 join results as shown in Algorithm 4. The entire join process terminates when all individual activities terminate.

C. PRPD *using X10*

For our purposes, the X10 implementation of the PRPD algorithm is as described in the previous section. Additionally, 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. As we also need to broadcast the skew keys as well as the duplication part of R, we choose the `x10.util.Team` API for efficient multi-point communication [19] instead of using loops on all places.

V. Evaluation

In this section, we present the results of our experimental evaluation on a commodity cluster. We conduct a quantitative evaluation of our implementation and compare them to the results obtained by other algorithms.

A. Platform

Our evaluation platform is the *Exascale 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. We fix the cardinality of R to 256 million tuples and S to 1 billion tuples. Join with such characteristics are common in data warehouses and column-oriented architectures.

Three key distributions are examined in our tests: uniform, low skew and high skew. We only add skew to S, following the 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). Again, highly skewed datasets are very common in a variety of settings in data warehouses and also in non-relational stores (e.g. see [20]).

C. Setup

We set the `X10_NPLACES` to the number of cores and `N_Thread` to 1, namely one place for one 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 1000 based on preliminary results. In all experiments, we only count the number of matches, but do not actually output join results. Moreover, we record the mean value based on ten measurements and we empty the file system cache between tests to minimize the effects of caching by the operating system.

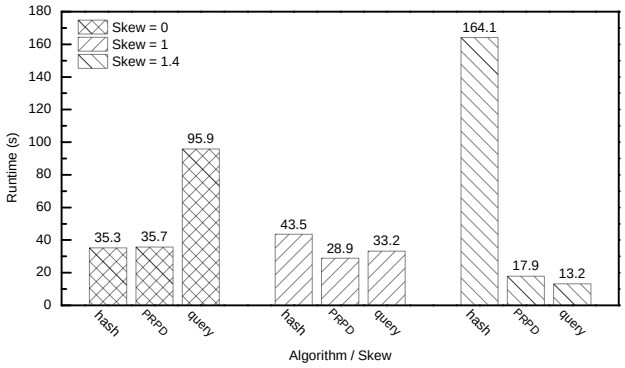

Figure 5. Runtime comparison of the three different algorithms. The join is implemented on $256M \bowtie 1B$ with different skew by using 192 cores.

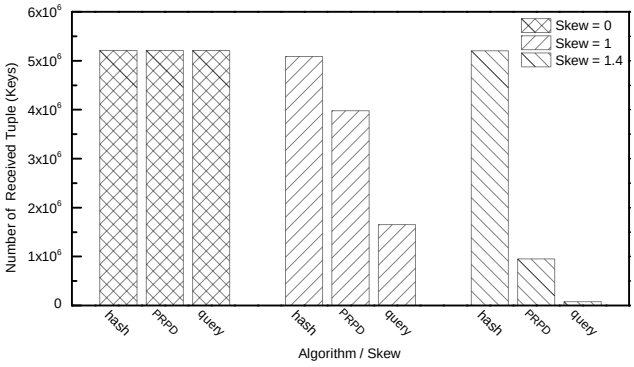

Figure 6. The average number of received tuples (or keys) for each place of the three different algorithms.

D. Runtime

We examined the runtime of three algorithms: conventional hash-based algorithm, PRPD [1] and our query-based approach. We implement these tests using 16 nodes (192 hardware cores) of the cluster on the datasets with different skews, and present the results in Figure 5. We can see that each algorithm has its strengths and weaknesses: (1) when the distribution is uniform, hash and PRPD perform nearly the same and much better than our query-based implementation, (2) with low skew, PRPD becomes the faster with our approach being slightly slower, and (3) with high skew, our approach outperforms the other two and the hash-based implementation shows very poor performance.

In the meantime, we also observe that with the increase of the data skew, the time cost of hash method increases sharply while our scheme decreases sharply, which means that our framework has total opposite properties compared with the commonly used hash-based join framework. In the meantime, PRPD is a hybrid method, still in the scope of the conventional approaches, so it has reasonable robustness against skew. Our method performs best under high skew conditions, so our new join framework can be considered as a supplement for the existing schemes. In fact, a system could pick the correct implementation based on the skew or the input so as to minimize runtime.

We have examined the time breakdown on each phase (not shown in the figure) and found that the time cost of our *push query keys* and *return queried values* phase is about three times more than the *S redistribution* and *build & probing* phases of the hash-based implementation respectively. This has corroborated our expectation mentioned in § III-C.

E. Network Communication

The number of received tuples (or query keys in our algorithm) for each place indicates both network load and load balancing. As R is uniformly distributed, we only show the part of transferred tuples (keys) of S in each algorithm. 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 in each algorithm is shown in Figure 6.

We can see that the three algorithms receive the same number of tuples when the dataset is uniform. This is reasonable, since the partial redistribution of PRPD is ineffective as there is no skew and the number of query keys is equal to the number of total keys in our approach. With the increase in skew, the received tuples in the hash-based method does change. In contrast, PRPD and our method show a significant decrease, as they are grouping skewed results more effectively. In addition, our method transfers much less data than PRPD. 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) of S at each place*. We have three reasons to do so: (1) R is uniform distributed that has no effect for the balance at each place, and the broadcast part of R in PRPD does not weaken the balancing as well, (2) the number can indicate the communication and computing time cost, the more tuples (keys) a place receives, the more time will be spent on data transferring and join (lookup) operation at this place, and (3) we have to push the values back and implement the *results lookups* in our query-based algorithm, however, (a) the number of returning values is the same as the received keys, which has the same effect for load balancing, and (b) the final lookups take only a very small part of the whole runtime that can even be neglected.

As the place that receives the maximum number of tuples dominates the final runtime, we just report results of the maximum and average number of the metric, which is shown in Table I. We can see that all three algorithm achieves perfect load balancing when the dataset is uniform. With the skew increase, the load balancing of hash-based algorithm

Table I
THE NUMBER OF RECEIVED TUPLES OR KEYS (IN MILLIONS)

Algo.\Skew	0		1		1.4	
	Max.	Avg.	Max.	Avg.	Max.	Avg.
hash-based	5.21	5.21	57.68	5.20	324.23	5.21
PRPD	5.21	5.21	6.73	3.98	3.62	0.95
query-based	5.21	5.21	1.68	1.65	0.09	0.08

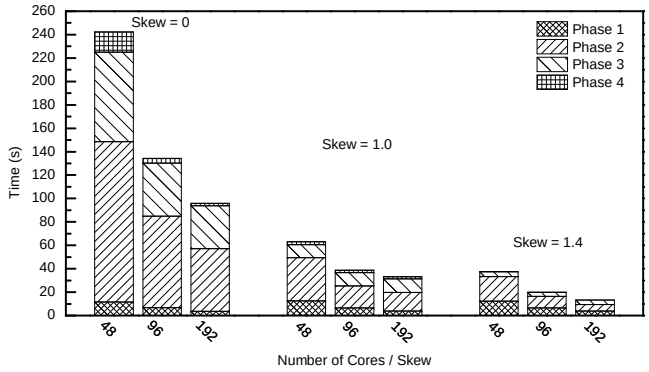

Figure 7. The detailed time cost of query-based approach on different key distributions by increasing number of cores.

becomes much worse. In the meantime, though PRPD has much improvement for that condition, our query-based approach still much better than PRPD, which nearly has not been effected by the data skew.

G. Scalability

We test the scalability of our implementation by varying the number of processing cores on all the three dataset. We start our test with 4 nodes (48 cores), 8 nodes (96 cores) and 16 nodes (192 cores). The detailed time-cost of each phase is shown in Figure 7.

We can see that the implementation generally scales well with the number of cores. In detail, when the dataset is uniformly distributed, all four phases (referred as phase 1 etc. according to § IV-B) scale well and the time-cost in the second and third step dominates the whole performance. When the distribution is skewed, we observe that phase 1 and phase 2 still scale well while phase 3 is slightly effected by increasing the number of cores, and the time-cost of phase 4 becomes extremely small. This is reasonable: (1) in *phase 1 & phase 2*, the operations are relying on the cardinalities of R and S at each node, but not the skew. (2) in *phase 3*, tuples are evenly distributed, which leads to the number of received query keys at each node not obviously changing when increasing then number of cores. Take the tuples with the same key k_1 for example, the $h_1(k_1)$-th node will always receive one k_1 from each node. It means that this node first receives 48 k_1 and then 96 k_1 when increasing the number of cores to 96. In the meantime, this increase will be leveraged by the decrease of the non-skewed query keys received at this node. (3) in *phase 4*, the size of the hash tables at each place built for S will decrease with the increment of the cores and the skew. That is why the time is only in the order of tens of *ms* when the skew is 1.4.

VI. CONCLUSIONS

In this paper, we have introduced a new framework for parallel joins, the *query-based distributed join*, which specifically targets joins with very high skew. We have presented an implementation of the framework in the APGAS programming model using the X10 system. Our experimental results show that our implementation is scalable, faster and results in less network communication compared to the state-of-art PRPD algorithm [1], in the presence of high skew.

Future work lies in combining our method with approaches that partition data according to key skew, such as PRPD, so as to efficiently execute joins with less skew. In addition, we will investigate the effect of varying join hit-rates on our framework and further investigate extensions to handle non-uniform network throughput (e.g. joins across racks). Finally, we intend to validate our approach on real-world workloads that present very high skew, such as the ones found in the Semantic Web field [20].

ACKNOWLEDGMENTS

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

REFERENCES

[1] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. ACM, 2008, pp. 1043–1052.

[2] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009.

[3] G. A. Cagri Balkesen, Jens Teubner and M. T. Öszu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *Proceedings of the 29th International Conference on Data Engineering*, ser. ICDE '13, 2013.

[4] C. B. Walton, A. G. Dale, and R. M. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins," in *Proceedings of the 17th International Conference on Very Large Data Bases*, ser. VLDB '91. Morgan Kaufmann Publishers Inc., 1991, pp. 537–548.

[5] 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.

[6] 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 *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '04. IEEE Computer Society, 2004, pp. 4–13.

[7] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *Proceedings of the 18th International Conference on Very Large Data Bases*, ser. VLDB '92. Morgan Kaufmann Publishers Inc., 1992, pp. 27–40.

[8] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner, "Spinning relations: high-speed networks for distributed join processing," in *Proceedings of the 5th International Workshop on Data Management on New Hardware*, ser. DaMoN '09. ACM, 2009, pp. 27–33.

[9] X. Zhou and M. E. Orlowska, "Handling data skew in parallel hash join computation using two-phase scheduling," in *Proceedings of The 1st International Conference on Algorithms and Architectures for Parallel Processing*, ser. ICAPP '95, vol. 2. IEEE, 1995, pp. 527–536.

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

[11] M. Bamha and G. Hains, "Progress in computer research," F. Columbus, Ed. Nova Science Publishers, Inc., 2001, ch. Frequency-adaptive join for shared nothing machines, pp. 227–241.

[12] 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 *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. ACM, 2005, pp. 519–538.

[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] G. S. Manku, "Routing networks for distributed hash tables," in *Proceedings of the 22nd annual Symposium on Principles of Distributed Computing*, ser. PODC '03. ACM, 2003, pp. 133–142.

[15] K. Imasaki and S. Dandamudi, "Performance evaluation of nested-loop join processing on networks of workstations," in *Proceedings of the 7th International Conference on Parallel and Distributed Systems*, ser. ICPADS '00. IEEE Computer Society, 2000, pp. 537 – 544.

[16] M. Al Hajj Hassan and M. Bamha, "An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems," in *Proceedings of The 16th annual IEEE International Conference on High Performance Computing*, ser. HiPC '09, 2009, pp. 350–358.

[17] M. Bamha, "An optimal skew-insensitive join and multi-join algorithm for distributed architectures," in *Proceedings of the 16th International Conference on Database and Expert Systems Applications*, ser. DEXA '05. Springer-Verlag, 2005, pp. 616–625.

[18] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '11. ACM, 2011, pp. 37–48.

[19] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat, "A performance model for x10 applications: what's going on under the hood?" in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, ser. X10 '11. ACM, 2011, pp. 1:1–1:8.

[20] S. Kotoulas, E. Oren, and F. van Harmelen, "Mind the data skew: distributed inferencing by speeddating in elastic regions," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. ACM, 2010, pp. 531–540.