# Robust and Efficient Large-Large Table Outer Joins on Distributed Infrastructures

Long Cheng[123] , Spyros Kotoulas[2], Tomas E Ward[1], Georgios Theodoropoulos[4]

[1] National University of Ireland Maynooth, Ireland
[2] IBM Research, Ireland
[3] Technische Universität Dresden, Germany
[4] Durham University, UK
long.cheng@tu-dresden.de, spyros.kotoulas@ie.ibm.com
tomas.ward@eeng.nuim.ie, theogeorgios@gmail.com

**Abstract.** Outer joins are ubiquitous in many workloads but are sensitive to load-balancing problems. Current approaches mitigate such problems caused by data skew by using (partial) replication. However, contemporary replication-based approaches (1) introduce overhead, since they usually result in redundant data movement, (2) are sensitive to parameter tuning and value of data skew and (3) typically require that one side is small. In this paper, we propose a novel parallel algorithm, Redistribution and Efficient Query with Counters (REQC), aimed at robustness in terms of size of join sides, variation in skew and parameter tuning. Experimental results demonstrate that our algorithm is faster, more robust and less demanding in terms of network bandwidth, compared to the state-of-the-art.

## 1 Introduction

Outer joins are popular in complex queries and frequently used in OLAP [1, 2] and large-scale data analysis, to name but a few applications. Unlike inner joins, the operation does not discard tuples from either relation that do not match with tuples in the other [3]. For example, for a left outer join (⟕) between two inputs $R$ and $S$ on their attributes $a$ and $b$, the following query returns not only the matched tuples in the form of <x,a,y>, but also <x,a,*null*>, when values do not match.

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

Currently, as for inner joins, implementations for distributed outer joins utilise one of two distributed patterns [4]: *redistribution-based* and *duplication-based outer joins*. To study the core performance characteristics of these approaches, we focus on analyzing the parallelism within a single outer join operation between two relations $R$ and $S$ on an $n$-node system (assuming both $R$ and $S$ are in the form of <key, value> pairs and $|R| < |S|$ in the following).

For redistribution-based approaches, parallel outer joins contain three phases: *partition*, *redistribution* and *local outer joins*. In the first phase, the relations $R_i$ and $S_i$, initially arbitrarily partitioned across each computation node $i$, are partitioned into distinct sets $R_{ik}$ and $S_{ik}$ ($k \in [1, n]$) respectively, according to the hash values of their join key attributes. Each of these sets is distributed to a corresponding remote node $k$ in the second phase. After that, the sequential outer joins of local fragments commence. This scheme can achieve near linear speed-up under ideal balancing conditions for distributed systems [5]. However, when the processed data has significant *attribute value skew*, the join performance will dramatically decrease due to the emergence of computational hot spots [6].

Duplication-based distributed outer joins differ significantly from inner joins. There are two distinct stages involved: (1) An inner join between $R$ and $S$, composed by a *duplication* and *local inner join* phase in which the former phase duplicates (broadcasts) $R_i$ at each node to all other nodes, and the latter is the same as that for sequential inner joins, formulating the intermediate results $T_i$ at each node $i$; (2) An outer join between $R$ and $T$, which is similar to the redistribution-based method described above. The *duplication* in this method can efficiently reduce hot spots resulting from *attribute value skew*. Nevertheless, this operation is costly and only suitable for small-large table outer joins. Additionally, such a scheme will still encounter performance bottlenecks when there exists *join product skew* [7], because in such scenarios the redistributed $T$ could be very large (e.g. Cartesian product) or suffer from skew itself.

As data skew occurs naturally in various applications [8], and join performance is challenged by large scale data in the era of Big Data, it is important for practical data systems to perform efficiently in such contexts. In this work, we propose a new outer join algorithm, redistribution and efficient query with counters (REQC), for robustly and efficiently processing large-large table outer joins on distributed architectures. We summarize the contributions of this paper as following:

- We apply the join geography of semijoins to parallel outer joins on distributed systems. We find that this semijoin-like scheme is better suited for skew handling in massive distributed joins.
- We further develop the semijoin-based scheme into the REQC algorithm, in order to increase performance and robustness.
- Experimental results on 192 cores and 1 billion tuples indicate that our method is both efficient and robust. Moreover, we compare our approaches with five different baselines taken from the literature which we implement on the same platform. Our findings indicate that our method is faster, more robust and requires less network communication, across a range of skew and parameter values.

The rest of this paper is organized as follows: In Section 2, we report on related work. We present our REQC algorithm in Section 3 and its detailed implementation in Section 4. We evaluate our approach in Section 5 while we conclude the paper and suggest directions for future work in Section 6.

## 2 Related Work

### 2.1 Related Work on Joins

Data skew is a significant problem for multiple communities, such as databases [9], data management [10], data engineering [11] and web data [8]. Joins with extreme skew can be found in the semantic web field (e.g. in [8], the most frequent item in a real-world dataset appeared in 55% of entries).

Research in parallel joins on shared-memory systems [9] and GPUs [12] has already achieved significant performance speedups through improvements in architecture at the hardware-level of modern processors. Nevertheless, as applications grow in scale, their associated scalability is limited by either the number of threads available or the system memory and I/O.

Various techniques have been proposed for distributed inner joins to handle skew [7, 13–15]. Often, the assumption is that inner join techniques can be simply applied to outer joins, as identified in [4]. However, applying such techniques for outer joins directly may lead to poor performance [16].

Current research on outer joins focuses on join reordering, elimination and view matching [3, 17, 18]. State-of-the-art methods designed specific for outer join implementation achieve significant performance improvements [4], however, they are based on the duplication-based method and cannot be applied to large-large table outer joins.

Distributed semijoins have been extensively studied, primarily in two domains: (1) joins in P2P systems, for reducing network communication based on the high selectivity of a join [13], such as descrbied in [19]; (2) pre-joins in distributed systems which seek to avoid sending tuples which will not participate in a join, such as the method described in [7], for a common implementation, and [20], for application to the MapReduce framework. In contrast, we apply a semijoin pattern with **full parallelism** to outer joins on a distributed architecture **directly**, and propose our efficient and robust REQC algorithm on this basis.

### 2.2 Details on the State-of-the-art

**PRPD.** Xu et al. [15] propose a hybrid distributed geography called PRPD (*partial redistribution & partial duplication*) for inner joins, by combining the two conventional patterns described. For a single skew relation $S$ (assume $R$ is uniformly distributed), the high skew tuples $S_{loc}$ of $S$ are retained locally and other tuples $S_{redis}$ are redistributed based on hashing. For $R$, the tuples $R_{dup}$ with keys contained in $S_{loc}$ are broadcast to all the nodes, and the rest $R_{redis}$ are redistributed as normal. The final joins are composed by $R_{redis} \bowtie S_{redis}$ and $R_{dup} \bowtie S_{loc}$ at each node.

As the high skew tuples of $S$ are not redistributed at all and, instead, just a small number of tuples from $R$ are broadcast, the *attribute value skew* can be highly reduced. This hybrid scheme has shown to be very efficient in processing inner joins, and could be applied to outer joins directly. Nevertheless, we have to redistribute the results of $R_{dup} \bowtie S_{loc}$ in an outer join, which could be very costly: since $S_{loc}$ is highly skewed, the cardinality of the intermediate results can be very large. This condition will be demonstrated in our evaluation in Section 5.

**DER.** Xu et al. [4] also propose another algorithm called DER (*duplication and efficient redistribution*), aimed at optimizing outer joins. This method comprises two stages: (1) $R_i$ at each node $i$ is duplicated to all the nodes to start inner joins. At this stage, not only are the matched results $T$ kept but also the ids of all *non-matched* rows in the table $R$; (2) Only the recorded ids are redistributed according to their hash values and, then, the final join results are assembled on that basis.

In fact, this optimization provides for an efficient way to extract non-matched results of an outer join. Notice that the *join* in the first stage of the conventional duplication-based scheme is an *inner join* instead of an outer join. The reason for this is that the duplication could bring either redundant or erroneous non-matched outputs. To alleviate this problem, redistributing the intermediate results is adopted. In comparison, DER uses a clever way around this: non-matched tuples of $R$ are redistributed and these tuples are indicated by a row-id from the table $R$. As such, the redistributed part is small and network communication and computational workload are greatly reduced. The experimental results show that the DER algorithm can achieve significant speedups over competing methods.

As DER must broadcast $R$, it is designed to work best for small-large table outer joins. When associated with the PRPD algorithm, the broadcasted part $R_{dup}$ is typically small. As identified by [16], we can integrate DER into PRPD to fix the cardinality problem as described for $R_{dup} \bowtie S_{loc}$ previously. The experiments in [16] have shown that this hybrid method (referred to PRPD+DER) is efficient on handling skew in large-large outer joins. Regardless, we will demonstrate that our proposed REQC algorithm can outperform this optimized technique.

**QC.** Recently, Cheng et al. [21] introduced a novel parallel join approach called *query-based distributed joins*, for handling data skew of inner joins on distributed architectures. An approach on that basis named *query with counters* (QC) [16] specified for *outer-joins* proved to be faster than the state-of-the-art in the presence of highly-skewed data. Regardless, the method performs bad when processing low-skew data. In comparison, the proposed REQC approach further refines that basic algorithm and we will show that this new method is more robust and also capable of higher performance than [16] in our evaluation in Section 5.

## 3 Our Approach

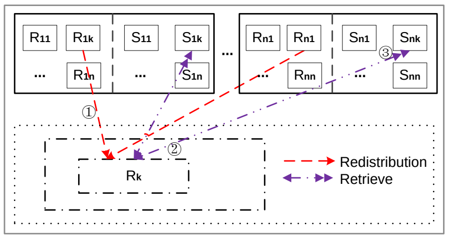
### 3.1 Semijoin for Outer Joins

The approach of semijoin-based distributed joins is shown in Figure 1(a), where the two communication patterns (*redistribution* and *retrieve*) makes it different from the conventional join approaches and the commonly-used semijoins. Under such a scheme, the implementation of the outer join in *Query 1* is organized as the following four steps:

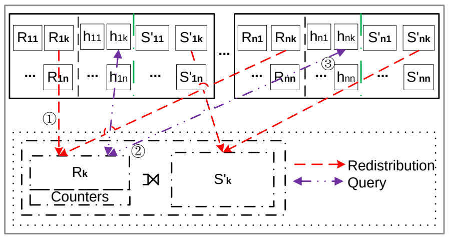
1. Tuples in $R_i$ at each node $i$ are redistributed to remote nodes based on the hash values of their attributes $a$. This process is shown as ① in the figure.
2. The *unique* keys[1] $\pi_b(S_i)$ of $S_i$ at each node $i$ are sent to the corresponding node as well, according to their hash values. This process is shown as ② in the figure.
3. All received tuples $\bigcup_{i=1}^{n} R_{ik}$ at each node $k$ probe all received keys $\bigcup_{i=1}^{n} \pi_b(S_{ik})$, organizing the matched results $T_k$ and output the non-matched results. After that, each key fragment $\pi_b(S_{ik})$ probe $T_k$ and send back the matched tuples to node $i$. The process of sending these back is shown as ③ in the figure.
4. The returned tuples join with tuples of $S_i$ at each node to produce matched results.

The final outer join results are composed from the output of the non-matched part in Step 3 and the matched part in Step 4. As we only distribute the *unique* keys of $S$, this scheme can be very efficient for handling data skew in distributed outer joins. More exactly, (1) even when $S$ is high skewed, each node will receive only one key (or maximum of $n$ keys if these tuples appear on the $n$ nodes); and (2) each transferred key is treated the same in the following look-up operations. We will exam the performance of this approach in our later evaluations.

[1] Here, we use the operator $\pi_b$ for presenting the duplicate-removing *projection* on the join attribute $b$ of the relation $S$.



(a) Semijoin-based approach



(b) REQC outer join approach

**Fig. 1.** The semijoin-based outer join approach and our proposed REQC method. The dashed square refers to the remote computation nodes and objects.

## 3.2 REQC Algorithm

To distinguish the matched and the non-matched tuples and then send the former back to the requester, we implement $\{[\bigcup_{i=1}^{n} R_{ik} \ltimes \bigcup_{i=1}^{n} \pi_b(S_{ik})] \ltimes \pi_b(S_{ik})\}$ at each node $k$ as described in the third step of the above method. This process is complex and could be time costly, since there is significant computation involved. In the meantime, when $S$ has low skew, the two-sided communication of large numbers of transferred keys and returned tuples can become costly, decreasing the join performance. To remedy these problems, we propose our REQC algorithm, shown in Figure 1(b), based on three optimizations:

1. Tuples in each $S_i$ are first divided into two parts before the joins: (a) the non- or low-skewed part $S_i'$, is hash-redistributed to all the nodes, and (b) the high-skewed part $h_i$, using the semijoin-based scheme.
2. Each received tuple fragment $S_{ik}'$ and key fragment $\pi_b(h_{ik})$ probes the received tuples $\bigcup_{i=1}^{n} R_{ik}$ at each node $k$. To identify the non-matched results, a *counter* is added to each tuple and it increases by one when this tuple is probed. Then, the non-matched tuples will be the ones with the counter value still at zero after all probings.
3. Only the retrieved *values* are sent back during the probing process of the key fragments, and the *value* is set to *null* when a key's probes have failed. The transferred keys are kept locally and the returned values follow the same sequence as these keys. Then, the <key, value> pair can be easily rebuilt based on their sequence (e.g. the index in an array), to compute the final join with $h_i$ at each node $i$.

With these optimizations, we can efficiently improve the performance of the semijoin-based approach as follows: (1) even when $S$ shows low skew, all tuples will be redistributed, avoiding the two-sided communications issue and consequently improving the robustness; (2) a simple probe operation is applied to the retrieval of the matched results for $\pi_b(h_{ik})$, which is much simpler than the previously mentioned join operations; (3) only values rather than entire tuples are returned, therefore the inter-machine communication is reduced. Though we also return the non-matched values as *nulls*, bringing additional communication, the number of $\pi_b(h_i)$ is always very small, making this effect negligible.

We refer to our algorithm as *redistribution and efficient query with counters* because (1) the process of transferring keys to remote nodes and retrieving the corresponding values looks like a *query*; (2) *counters* are used to distinguish non-matched results; and (3) only tuples corresponding to keys with high skew are processed by *querying*. We refer to the algorithm with only the latter two optimizations as QC (*query with counters*) [16]. As shown in our later evaluation, QC is always faster than the semijoin-based approach, implying that the introduction of *counters* is itself beneficial for such operations.

Moreover, compared with the state-of-art techniques PRPD+DER [4, 15] described in Section 2, our approach does not involve any redundancy in join (or lookup) operations, because our method is totally duplication-free and all nodes only receive the tuples that they will eventually use. This should make the approach more efficient, and we will conduct a detailed performance comparison in Section 5. Additionally, the join framework of our approach is more straightforward and can be applied to other kinds of joins directly (e.g. the returned *null* can be applied directly for right outer-joins and the *counters* for anti-joins etc).

## 4 Implementation

In this section, we present a detailed implementation of the proposed REQC approach. We compare our algorithm with the state-of-art techniques PRPD+DER [4, 15]. Since [4, 15] do not provide any code-level information, and in the interests of making a fair comparison, we have implemented all these methods with the parallel language X10 [22].

### 4.1 Pre-partitioning of Skew Tuples

We have to measure the local skew so as to partition the relation $S$ at each node for our algorithm as well as the PRPD+DER method. Efficient skew measurement is beyond the scope of this work. As we are more interested in a high performance in-memory implementation we add two pre-processing steps before each test: (1) for each test parameterized by $t$, each node pre-reads the keys appearing more than $t$ times into an `ArrayList` and considers these the required skew keys; and (2) Tuples in $S_i$ at each node $i$ are divided into $S'_i$ and $h_i$ based on an assessment of the skewed keys, and each of them is kept in an `ArrayList` as well. These pre-processing steps make our later performance comparison more fair and meaningful because: (1) the total join performance is very sensitive to the chosen skew keys and operations like sampling cannot guarantee the same set of keys are selected, (2) the extra time cost for skew extraction is removed, so that the focus is on the analysis of runtime performance only, and (3) in a real system, there are opportunities to perform these operations as part of other processing activities.

### 4.2 Parallel Join Processing

We describe our implementation at each node as the following four steps. As the local join process is well studied and techniques such as the sort-merge and hash joins are commonly used, we have selected the hash-join for our implementations.

**R Distribution:** As shown in Figure 2 lines 1-8, tuples of $R$ at each computation node are partitioned into $n$ chunks, and each tuple is assigned according to the hash value of its key by a hash function $h(key) = key\ mod\ n$. After that, all collected tuples in the chunk $R\_c(i)$ is transferred to the remote node $i$. Note that the term *here* means the id of current node.

**Push Query Keys:** Similar to the previous step, tuples of $S'$ are also hash-redistributed to remote nodes. For the high skewed part $h$, tuples are kept in *hashmap* and only the unique keys are pushed to remote nodes. Lines 9-21 of Figure 2 present the details of this process. There, each `HashMap` in *h_c* supports the data structure of $1 \rightarrow n$ mapping, so as to efficiently hold skewed tuples. In addition both the *h_c* and *local_key_c* are kept in memory for computing the final joins, as mentioned in Section 3.2. We synchronize the operation here to guarantee the completion of the data transfer at each node before the next phase commences.

**Return Queried Values:** This phase starts after the grouped query keys have been transferred to the appropriate remote nodes. The implementation at each node is similar to a sequential hash join. For each received tuple of $R$, as shown in lines 22-27 of Figure 2, a <*key,(value, 0)*> pair is placed in the local hash table *T*, where the 0 means the initialized $counter = 0$ of this tuple. After that, as shown in lines 28-40, the received keys start to access *T* sequentially to obtain 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 placed into an array $r\_value\_c$ so that it can be sent back to the requester(s). All these processes take place in parallel at each node, and we also synchronize the operations here.

**Result Lookup:** The join results of the high skewed tuples at each node 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. In the meantime, the received tuples of $S'$ probe the hash table $T$ to retrieve the matched results for the low skewed tuples. After that step, we can easily scan the *counter* of each tuple in $T$ to organize the non-matched results. This process is described in lines 41-61 of Figure 2. The entire join process terminates when all individual nodes terminate.

## 5 Evaluation

**Platform.** Our evaluation platform is the *High-Performance Systems Research Cluster* located at 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.

**Datasets.** Our evaluation is implemented on two relations $R$ and $S$, which are both two-column tables. We fix the cardinality of R to 64 million tuples and S to 1 billion tuples and set both their key and payload to 8-byte integers. We assume that $R$ and $S$ meet the foreign key relationship, namely every tuple in $S$ is guaranteed to find exactly one join partner in $R$ [11], and 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 0 for uniform, 1 for the low skew (top ten popular keys appear 14% of the time)

```
    R Distribution:
1:  Initialize $R\_c$:array[array[tuple]]($n$)
2:  for $tuple \in list\_of\_R$ do
3:      $des \leftarrow$ hash($tuple.key$)
4:      $R\_c(des).add(tuple)$
5:  end for
6:  for $i \leftarrow 0..(n-1)$ do
7:      Push $R\_c(i)$ to $r\_R\_c(i)(here)$ at node $i$
8:  end for

    Push Query Keys:
9:  Initialize $S'\_c$:array[array[tuple]]($n$)
               $h\_c$:array[hashmap[tuple]]($n$)
10: for $tuple \in list\_of\_S'$ do
11:     $des \leftarrow$ hash($tuple.key$)
12:     $S'\_c(des).add(tuple)$
13: end for
14: for $tuple \in list\_of\_h$ do
15:     $des \leftarrow$ hash($tuple.key$);
16:     $h\_c(des).put(tuple)$
17: end for
18: for $i \leftarrow 0..(n-1)$ do
19:     Extract unique keys of $h\_c(i)$ to $local\_key$(i)
20:     Push $local\_key(i)$ to $r\_key\_c(i)(here)$,
             $S'\_c(i)$ to $r\_S'\_c(i)(here)$  at node i
21: end for

    Return Queried Values:
22: Initialize $T$:hashmap, $r\_value\_c$:array[value]
23: for $i \leftarrow 0..(n-1)$ do
24:     for $tuple \in r\_R\_c(here)(i)$ do
25:         Put <$tuple.key$, ($tuple.value$, 0)> into $T$
26:     end for
27: end for
28: for $i \leftarrow 0..(n-1)$ do    // probing received keys
29:     for $key \in r\_key\_c(here)(i)$ do
30:         if $key \in T$ then
31:             $r\_value\_c.add(T.get(key).value)$
32:             $T.get(key).counter{+}{+}$
33:         else
34:             $r\_value\_c(i).add(null)$
35:         end if
36:     end for
37: end for
38: for $i \leftarrow 0..(n-1)$ do
39:     Push $r\_value\_c(i)$ to $value\_c(i)(here)$ at node $i$
40: end for

    Result Lookup:
41: for $i \leftarrow 0..(n-1)$ do    // joins of high skew part $h$
42:     for $value \in value\_c(here.id)(i)$ do
43:         if $value \neq null$ then
44:             Lookup corresponding $key$ over $h\_c(i)$
45:             Output matched results
46:         end if
47:     end for
48: end for
49: for $i \leftarrow 0..(n-1)$ do    // joins of low skew part $S'$
50:     for $key \in r\_S'\_c(here)(i)$ do
51:         if $key \in T$ then
52:             Output the matched result
53:             $T.get(key).counter{+}{+}$
54:         end if
55:     end for
56: end for
57: for $key \in T$ do
58:     if $T.get(key).counter == 0$ then
59:         Output non-matched results
60:     end if
61: end for
```

**Fig. 2.** Implementation of proposed REQC algorithm at each node.

and 1.25 for high skew dataset (top ten popular keys appear 52% of the time). Joins with such characteristics and workloads are common in data warehouses and column-oriented architectures as well as being prevalent in recent studies [9–11].

**Setup.** In all experiments, we only count the number of matches, we do not actually output join results. Moreover, for PRPD, PRPD+DER and our REQC algorithm, in which skewed tuples need to be pre-extracted, we implemented a test series with different parameters $t$ (recall that tuples where the key appears more than $t$ times is considered as skewed) for each dataset, as shown in Figure 4. When presenting results, we always choose the $t$ with the best runtime achieved.

## 5.1 Runtime

**Performance.** We examined the runtime performance of the six algorithms as described previously: the conventional redistribution-based algorithm (referred to *Hash*), PRPD [15], PRPD+DER [4, 15], semijoin-based outer joins (referred to as *Semijoin*), QC [16] and the proposed REQC approaches. We implement our tests using 16 nodes (192 hardware cores) of the cluster and present the results in Figure 3. It can be seen that: (1) when $S$ is uniform, the first three methods and REQC perform nearly the

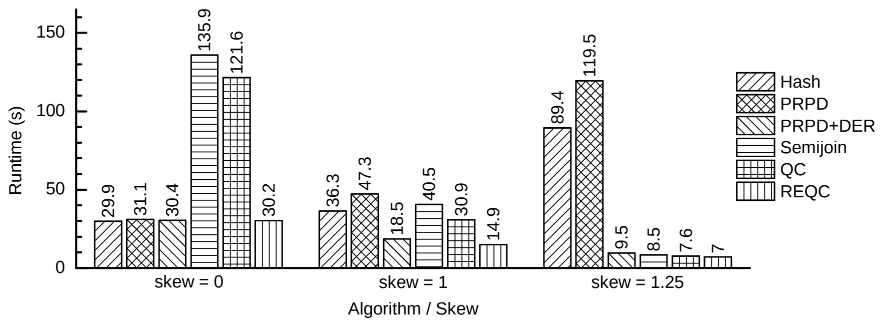


**Fig. 3.** Runtime comparison of the six algorithms under varying skew (192 cores).

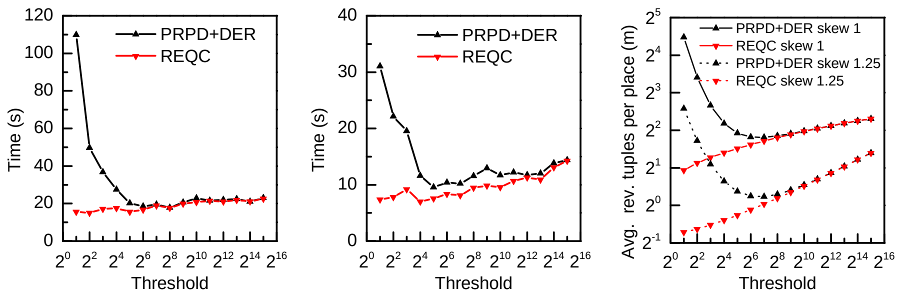
same, much faster than *Semijoin* and QC; (2) with low skew, PRPD+DER and REQC outperforms the other four algorithms; and (3) with high skew, the latter four algorithms perform much better than *Hash* and PRPD.

We can also observe that the time cost of *Hash* and PRPD increases sharply with the increase in data skew. In contrast, for the other four algorithms, it decreases. Moreover, PRPD always performs the worst, meaning that the approach for inner joins cannot be applied to outer joins directly. In the meantime, QC is always faster than *Semijoin*, demonstrating that the latter two optimizations described in Section 3.2 do improve join performance by themselves. Furthermore, runtime performance of PRPD+DER and REQC changes much more gradually than the other four algorithms with increasing skew, demonstrating their robustness under varying skew. Finally, it is also worth highlighting that our proposed REQC approach is always faster than the state-of-the-art PRPD+DER algorithm, about 24%-36% depending on skew value.

**REQC vs PRPD+DER.** We conduct a more detailed comparison of our REQC and PRPD+DER, based on a series of tests with different parameters $t$, corresponding to what the system considers a popular key. The results are presented in Figure 4(a) and 4(b). It can be seen that REQC always outperforms PRPD+DER for any given $t$. In addition, the runtime difference for different $t$ values are only minor for our REQC algorithm while those in PRPD+DER change more rapidly, demonstrating that our approach is more robust with respect to input parameters. In fact, tuning $t$ would require additional, more complex or costly operations, meaning that the performance difference between the two approaches would be even greater for applications which include these steps.

## 5.2 Network Communication and Load Balancing

**Network Communication.** We count a single key or payload as 1/2 of a tuple, and record *the average number of received tuples at each core* for each algorithm as shown in Table 1. We can see that *Semijoin* results in the highest number of tuples while the other five algorithms receive the same number of tuples when the dataset is uniform. This is expected, since (1) tuples in the first three algorithms and REQC are just simply redistributed; (2) the number of transferred keys and the payload of QC is equal to the number of tuples; and (3) *Semijoin* not only moves all the keys, but also all the retrieved tuples. With an increase in skew, the average received tuples in the *Hash* and PRPD



(a) runtime skew=1 (b) runtime skew=1.25 (c) network communication

**Fig. 4.** Runtime and network communication of PRPD+DER and REQC with increasing threshold $t$ over different skews (192 cores).

**Table 1.** Number of tuples (max/avg) received at each core using 192 cores (millions)

| Max. / Avg. | Hash | PRPD | PRPD+DER | Semijoin | QC | REQC |
|---|---|---|---|---|---|---|
| **skew=0** | 5.9 / 5.9 | 5.9 / 5.9 | 5.9 / 5.9 | 8.7 / 8.7 | 5.9 / 5.9 | 5.9 / 5.9 |
| **skew=1** | 62.4 / 5.9 | 62.4 / 5.9 | 3.5 / 3.5 | 3.0 / 3.0 | 2.1 / 2.1 | 2.3 / 2.3 |
| **skew=1.25** | 239.8 / 5.9 | 239.8 / 6.0 | 1.3 / 1.3 | 0.7 / 0.7 | 0.6 / 0.6 | 0.8 / 0.8 |

methods generally does not change. The reason is that all tuples are still redistributed in *Hash* and PRPD needs to redistribute the large number of intermediate results. In contrast, the other four show a significant decrease, as they do not move high skew tuples. In addition, our REQC algorithm transfers less data than PRPD+DER.

We also track the detailed number of received tuples for different threshold $t$ values for REQC and PRPD+DER and present the results in Figure 4(c). It can be seen that in PRPD+DER that number first decreases and then increases, showing a trade-off between the number of duplicated and redistributed tuples. For REQC, the number of received tuples is always increasing, however, it is less than PRPD for any given $t$. In our tests, the best performance achieved in REQC is always better than PRPD+DER. For example, $t = 4$ for REQC and $t = 64$ for PRPD+DER in the condition $skew = 1$. That is why REQC transfers less data than PRPD+DER in Table 1, notably 34%-38% less, under skew.

**Load Balancing.** We analyze the load balancing properties of each algorithm based on *the maximum number of received tuples at each core*. We can see that the first two algorithms encounter serious load-balancing problems when the data exhibits skew. In contrast, the latter four algorithms achieve perfect load balancing under varying skew.

## 5.3 Scalability

We finally test the scalability of our REQC algorithm. We implement our test on a distributed architecture with 2 nodes (24 cores), 8 nodes, 12 nodes and 16 nodes (192 cores) on all three datasets. The detailed time-cost is shown in Figure 5, where each step there is consistent with the implementation explained in Section 4.2.

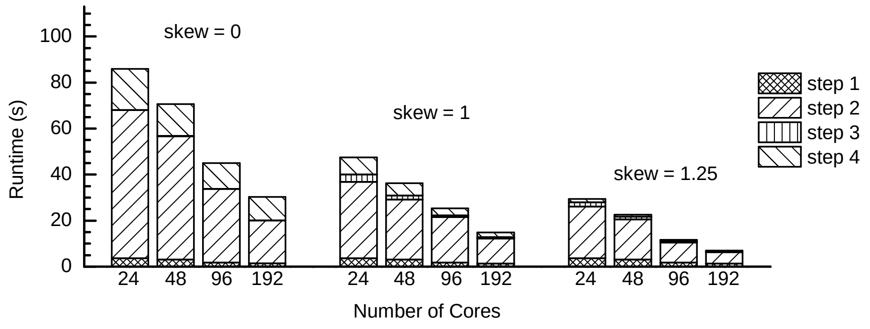


**Fig. 5.** The detailed time cost of the REQC algorithm with increasing the number of cores.

We can see that our algorithm generally scales well with the number of cores under varying skew. More specifically when data is uniformly distributed, the second and fourth step scale well and dominate the runtime. In addition, the time cost of the third step is nearly 0, the reason is that there are no *query* keys for remote nodes. With low skew, all four steps decrease with increase in the number of cores, and the second step becomes the most expensive part of the execution. Moreover, for high skew, the second step is always the dominating factor in performance. All of this demonstrates that the *query* processing of the third step in our algorithm is very lightweight, and the process in the second step (namely tuple hash-partitioning, local hash table building for high skew tuples and data transfers) has a high impact on the join performance.

## 6 Conclusions

In this paper, we have introduced a new algorithm, *redistribution and efficient query with counters*, for robustly and efficiently computing large-large table outer joins on distributed architectures. We have presented a detailed implementation of our approach and the experimental results demonstrate that our implementation is robust, efficient and scalable. Furthermore, compared to state-of-the-art techniques [4, 15], our algorithm always performs better with less network communication under skew conditions.

Data duplication is widely used in data engineering to reduce data movement and load imbalance. As our algorithm is duplication-free, we anticipate that our proposed method will not only be a supplement to existing schemes on parallel joins to minimize runtime but also for other domains. We intend to apply our approach in the semantic web domain, where workloads present very high skew [8].

## Acknowledgments

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

## References

1. Galindo-Legaria, C., Rosenthal, A.: Outerjoin simplification and reordering for query optimization. ACM Transactions on Database Systems (TODS) **22**(1) (1997) 43–74

2. Rao, J., Pirahesh, H., Zuzarte, C.: Canonical abstraction for outerjoin optimization. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. SIGMOD '04, ACM (2004) 671–682
3. Bhargava, G., Goel, P., Iyer, B.: Hypergraph based reorderings of outer join queries with complex predicates. ACM SIGMOD Record **24**(2) (1995) 304–315
4. Xu, Y., Kostamaa, P.: A new algorithm for small-large table outer joins in parallel DBMS. In: Proceedings of the 26th IEEE International Conference on Data Engineering. ICDE '10 (2010) 1018–1024
5. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6) (June 1992) 85–98
6. DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical skew handling in parallel joins. In: Proceedings of the 18th International Conference on Very Large Data Bases. VLDB '92, Morgan Kaufmann Publishers Inc. (1992) 27–40
7. Al Hajj Hassan, M., Bamha, M.: 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. HiPC '09 (2009) 350–358
8. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the data skew: distributed inferencing by speeddating in elastic regions. In: Proceedings of the 19th International Conference on World Wide Web. WWW '10, ACM (2010) 531–540
9. Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. Proc. VLDB Endow. **2**(2) (August 2009) 1378–1389
10. Blanas, S., Li, Y., Patel, J.M.: 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. SIGMOD '11, ACM (2011) 37–48
11. Cagri Balkesen, Jens Teubner, G.A., Öszu, M.T.: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: Proceedings of the 29th International Conference on Data Engineering. ICDE '13 (2013)
12. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08, ACM (2008) 511–524
13. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4) (December 2000) 422–469
14. Zhang, X., Kurc, T., Pan, T., Catalyurek, U., Narayanan, S., Wyckoff, P., Saltz, J.: Strategies for using additional resources in parallel hash-based join algorithms. In: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing. HPDC '04, IEEE Computer Society (2004) 4–13
15. Xu, Y., Kostamaa, P., Zhou, X., Chen, L.: Handling data skew in parallel joins in shared-nothing systems. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08, ACM (2008) 1043–1052
16. Cheng, L., Kotoulas, S., Ward, T., Theodoropoulos, G.: Efficient handling skew in outer joins on distributed systems. In: Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. CCGrid '14 (2014) 295–304
17. Hill, G., Ross, A.: Reducing outer joins. The VLDB Journal **18**(3) (June 2009) 599–610
18. Larson, P.Å., Zhou, J.: View matching for outer-join views. The VLDB Journal **16**(1) (2007) 29–53
19. Koloniari, G., Pitoura, E.: Peer-to-peer management of XML data: issues and research challenges. ACM Sigmod Record **34**(2) (2005) 6–17
20. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in MapReduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM (2010) 975–986
21. Cheng, L., Kotoulas, S., Ward, T., Theodoropoulos, G.: QbDJ: A novel framework for handling skew in parallel join processing on distributed memory. In: Proceedings of the

15th IEEE International Conference on High Performance Computing and Communications. (2013) 1519–1527

22. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: 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. OOPSLA '05, ACM (2005) 519–538