



Research Repository UCD

| | |
|-------------------------------------|---|
| Title | Minimizing Network Traffic for Distributed Joins Using Lightweight Locality-Aware Scheduling |
| Authors(s) | Cheng, Long, Murphy, John, Liu, Qingzhi, et al. |
| Publication date | 2018-08-31 |
| Publication information | Cheng, Long, John Murphy, Qingzhi Liu, and et al. "Minimizing Network Traffic for Distributed Joins Using Lightweight Locality-Aware Scheduling." Euro-Par, August 31, 2018. https://doi.org/10.1007/978-3-319-96983-1 . |
| Conference details | The 24th International European Conference on Parallel and Distributed Computing (EURO-PAR 2018), Turin, Italy, 27-31 2018 |
| Publisher | Euro-Par |
| Item record/more information | http://hdl.handle.net/10197/10051 |
| Publisher's version (DOI) | 10.1007/978-3-319-96983-1 |

Downloaded 2025-12-04 22:54:20

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)



© Some rights reserved. For more information

Minimizing Network Traffic for Distributed Joins Using Lightweight Locality-Aware Scheduling

Long Cheng^{1,2}, John Murphy¹, Qingzhi Liu², Chunliang Hao³ and Georgios Theodoropoulos⁴

¹ PEL, University College Dublin, Ireland

² Eindhoven University of Technology, The Netherlands

³ Institute of Software, CAS, China

⁴ Southern University of Science and Technology, China
`long.cheng@ucd.ie`

Abstract. Large computing systems such as data centers are becoming the mainstream infrastructures for big data processing. As one of the key data operators in such scenarios, distributed joins is still challenging current techniques since it always incurs a significant cost on network communication. Various advanced approaches have been proposed to improve the performance, however, most of them just focus on data skew handling, and algorithms designed specifically for communication reduction have received less attention. Moreover, although the state-of-the-art technique can minimize network traffic, it provides fine-grained optimal schedules for all individual join keys, which could result in obvious overhead. In this paper, we propose a new approach called LAS (Lightweight Locality-Aware Scheduling), which targets reducing network communication for large distributed joins in an efficient and effective manner. We present the detailed design and implementation of LAS, and conduct an experimental evaluation using large data joins. Our results show that LAS can effectively reduce scheduling overhead and achieve comparable performance on network reduction compared to the state-of-the-art.

1 Introduction

To cope with the growing Big Data from various domains, large systems such as data centers have been built across the globe to support high-performance data processing. As one of the core tasks in such scenarios, efficient execution of distributed data operators such as joins is still challenging current techniques and systems. The main reason is that these operators are always expensive, in terms of both network resource consumption and network communication time. In fact, in recent years, the performance of CPUs has grown much faster than network bandwidth and, as such, the network becomes a performance bottleneck to computation [1, 2]. Therefore, effective strategies on the execution of distributed data operators, which can reduce data communication time, becomes increasingly desirable.

In this work, we focus on one of the most challenging operators – distributed joins, which is used to facilitate combination of two relations based on a common

key. More specifically, we focus on reducing its network traffic. The main reason is that any communication reduction in a distributed join will be directly translated to faster execution, for both low-end and high-end platforms [3]. Moreover, data systems would also benefit from our design in terms of energy consumption, since data centers could consume obvious energy on communication links, switching and aggregation elements.

A typical distributed join implementation contains a data redistribution process, which always incurs large amounts of data transferring over networks [4]. Various advanced join approaches have shown that they can effectively reduce network communication [4, 5]. However, they mainly focus on data skew handling, i.e., communication reduction is only considered as a byproduct of their designs. In comparison, the state-of-the-art *track* approach [3] is designed specifically for minimizing network traffic. It provides a fine-grain granularity optimum on data locality for all input tuples, and thus minimal data communication can always be achieved. Moreover, the experiments have shown that the method can significant speed up conventional join approaches. However, the scheduling process of *track* is relatively complex, which would make the scheduling itself costly and thus bring in obvious overhead for the final joins, especially when the number of keys is large (details see Section 2).

To reduce network traffic for distributed joins in an efficient and effective manner, in this paper, we present a novel algorithm called LAS (**L**ightweight **L**ocality-**A**ware **S**cheduling). We provide the detailed design and implementation of LAS in a distributed computing environment and conduct a performance evaluation using maximal 100GB data over up to 128 computing cores (32 nodes). We summarize the contributions of this work as following:

- We introduce the state-of-the-art scheduling approach for distributed joins and analyze its possible performance issues in the presence of big datasets.
- We propose LAS, for minimizing network traffic in distributed joins, by incorporating efficient and effective strategies on data locality exploration.
- Our experimental results demonstrate that LAS is obviously lightweight and can achieve comparable performance on network reduction compared to the state-of-the-art. Moreover, join implementations based on LAS can significantly outperform the conventional approaches for large datasets.

The rest of this paper is organized as follows: In Section 2, we introduce some current distributed join approaches and analyze their possible performance issues. We describe our new method and its implementation in Section 3, and present the experimental evaluation in Section 4. We report on related work in Section 5 while we conclude the paper in Section 6.

2 Background

In this section, we briefly introduce two basic join approaches and two advanced techniques. Moreover, we also discuss about their possible performance issues in terms of network communication.

2.1 Basic Approaches

The redistribution-based and duplication-based joins are the two conventional distributed join methods. In the former approach, tuples on each node are firstly partitioned into distinct sets. Then, each set is transferred to a remote node for final local joins [4]. As the partitioning is usually based on the hash values of join keys, we refer the method to Hash in the following. An example of the scheme between two relations R and S is shown in Fig. 1(a). There, tuples are in the form of $\langle k, v \rangle$ pairs, where k is join key and v is payload. Assuming each hash value is calculated based on the modulus of 5, then all the tuples with key 3 will be transferred to node 3. If we quantify the cost of network traffic by the number of tuples transferred to remote nodes, then the cost of Hash is 13. Similarly, as shown in Fig. 1(b), for a duplication-based case, all the 4 tuples in the small relation R is duplicated to all the remote nodes, and thus its cost is 16.

From a scheduling perspective, the two approaches have not used any related techniques: the destination node(s) of a tuple only depends on the hash value of its join key or the size of the relation it belongs to. These kinds of straightforward processing make the two methods far from optimal in terms of network communication, because transferring all input tuples or broadcasting a relation over networks is always expensive. Additionally, as a well known issue, the performance of Hash will dramatically decrease in the presence of significant data skew [4]. We will focus on Hash in the following, as the approach is widely used in various data applications. In contrast, the duplication-based method is relatively seldom adopted, except for small-large joins [4].

2.2 Skew Handling Methods

To improve the performance of the conventional approaches, various advanced techniques have been proposed, and one of them is PRPD (partial redistribution & partial duplication) [5]. Its main idea is to distinguish skew and non-skew tuples and handle them in different ways: non-skew tuples are processed by Hash while the skew ones are based on duplication. As shown in Fig. 2(a), assume the tuples with key 3 are the skew ones, then the two tuples $\langle 3, 10 \rangle$ and $\langle 3, 11 \rangle$ of R at node 2 will be broadcast, and the tuples $\langle 5, 13 \rangle$ and $\langle 8, 12 \rangle$ will be redistributed. Then, the network cost of PRPD is 10, which is smaller than Hash. Moreover, it should be noticed that all the skew tuples in S are kept locally rather than being redistributed, and thus the data skew problem in Hash is well addressed.

In terms of data locality scheduling, PRPD only needs to identify the skew keys of input tuples. This process would be simple and can be done in a quick way, such as *sampling*, while the cost of which is always negligible for join time [5]. Because of the efficiency of PRPD, the method has been widely used and studied in various large data systems. However, if we look at the details of PRPD in Fig. 2(a), we can find that PRPD is not optimal in communication reduction in two perspectives: (1) part of data transferring is actually unnecessary. For example, the tuples $\langle 5, 13 \rangle$ and $\langle 8, 12 \rangle$ do not make any contributions to the join, i.e., their join results are actually empty, but they are scheduled to be

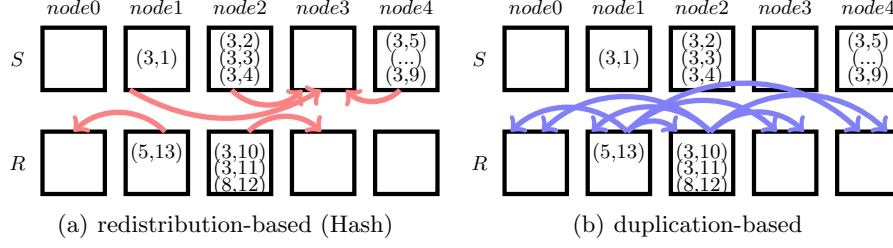


Fig. 1. Two basic data movement approaches in a distributed join.

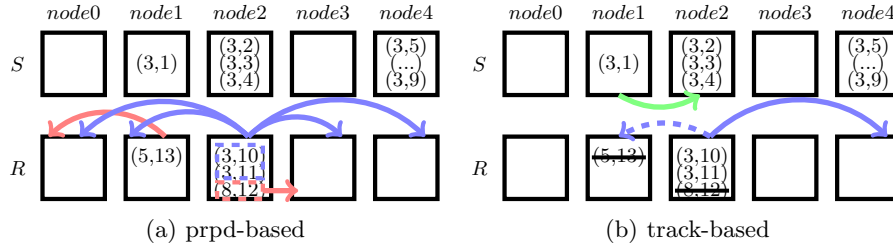


Fig. 2. Two advanced data movement approaches in a distributed join.

transferred; and (2) some nodes may receive some tuples that they do not really need. For instance, S does not have any tuples on node 0 and 3, but the two tuples $\langle 3, 10 \rangle$ and $\langle 3, 11 \rangle$ are scheduled to be duplicated to the two nodes.

2.3 The State-of-the-art

Compared to PRPD, the *track* approach [3] (referred to Track) is able to minimize network traffic for a distributed join, which can be considered as the state-of-the-art. The method is based on the complete knowledge of occurrence frequency of each key on each node. As illustrated in Fig. 2(b), based on a global statistic, Track knows that none of the five nodes has tuples with keys 5 or 8 in S , thus the tuples with these two keys in R , i.e., $\langle 5, 13 \rangle$ and $\langle 8, 12 \rangle$, will be scheduled to keep locally (and be ignored in later join execution). Moreover, for the case with key 3, Track only broadcasts the two tuples $\langle 3, 10 \rangle$ and $\langle 3, 11 \rangle$ to the nodes with matching keys in S , i.e., node 1 and 4. This process is called as *select broadcast*, which is different from PRPD that broadcasting to all the nodes (i.e., *full broadcast*). After that, Track uses a very smart way, i.e., *migration*, to further explore the possibilities on reducing communication traffic. For example, Track will check whether the cost is decreased, when moving $\langle 3, 1 \rangle$ from node 1 to node 2 with removing the duplication of $\langle 3, 10 \rangle$ and $\langle 3, 11 \rangle$ from node 2 to node 1. In this case, the traffic is indeed decreased, and consequently the schedule plan will be updated. Following this processing, the final network cost of Track is 3, which is much smaller than the above approaches.

Track is actually an approach which has extensively used the philosophy of moving *small* data chunks instead of *large* data chunks in a distributed environment to minimize network traffic. More specifically, it employs of a fine-grained multi-phase scheduling algorithm to explore data locality in a superlative way, i.e., *per distinct key*¹. In fact, the global statistic of each key is done in a way which is very similar to a distributed join over all unique keys. This could be costly, when the number of unique keys is large. Moreover, different from a simple join, to record the frequency and location of each distinct key, the pre-join has to aggregate all the keys in the form of (key, (list[N_r], list[N_s])), the computing of which could be expensive. In addition to that, the scheduling using *select broadcast and migration* (SBM) has to be performed on all the aggregated (list[N_r], list[N_s]), which would be also time costly if the number of nodes is large. All of these could result in obvious overhead for the join execution of Track. In comparison, as we will show later that LAS is significantly lightweight and can achieve comparable performance to Track on network reduction.

3 Our Approach

In this section, we present LAS and its implementation in detail. Additionally, we also discuss about its advantages by the comparison with current techniques.

3.1 The LAS Method

Based on the analysis of the approaches described above, we can see that there is actually a trade-off between communication reduction and scheduling overhead. Namely, scheduling leading to less network traffic could have a heavier overhead in data locality exploration, and vice versa. To reduce network traffic as much as we can and also in a quick manner for big data joins, we have one core design principle for LAS: fine granularity optimal scheduling such as Track should only be applied to a small number of keys which can greatly reduce network traffic, rather than all the distinct keys.

Following the above principle, we can use a hybrid way to explore data locality for input tuples, i.e., applying Track on the skew tuples and Hash on the rest ones. The main motivation is that the number of skewed tuples could be huge, but the number of their unique keys is normally small, and thus they can be scheduled in a quick way. Moreover, although the number of non-skew keys is large, Hash can always handle them quickly. This means that such a hybrid scheduling will be very lightweight. However, since the communication reduction ability of Hash is weak, the network traffic brought by the hybrid scheduling could be still high. Taking the case in Fig. 2(b) for example, the non-skew tuples $\langle 5, 13 \rangle$ and $\langle 8, 12 \rangle$ will be transferred, which is actually not necessary.

¹ Note that here *per distinct key* means that even though a key in R and a key in S have the same value, they will be distinguished, i.e., each of them is identified by a tag like R or S . Later, we will use the term *per join key* to remove this distinction.

To improve the above problem, we use a relatively fine-grained method for the non-skew data from the idea of Track: exploring data locality in a *per-key* level. The main difference is that we perform our scheduling based on *per join key* rather than *per distinct key*. Namely, when we count the key appearing frequency, a key in R and a key in S will be treated as a same key, if they have the same value (i.e., matching). For the detailed scheduling, we do not distinguish keys from R and S , thus we do not need to perform any SBM operations. Instead, for each join key, we just set its destination to a node, which contains the most number of tuples having the key. We call this processing as a *locality-aware* operation (LA). Obviously, LA can greatly simplify the statistic and data locality exploration operations on each node, compared to Track. In general, the proposed LAS can be divided into the following three main phases from a global point of view:

- *phase 1*: on the basis of identified skew keys, group all input tuples of R and S into two parts, the skew ones (R', S') and non-skew ones (R'', S'') .
- *phase 2*: minimize the network traffic for (R', S') use Track [3], i.e., exploring data locality for *each distinct key* using *select broadcast and migration*.
- *phase 3*: minimize the network traffic for (R'', S'') by exploring data locality for *each join key* with a *locality-aware* way.

Based on LAS, the non-skew tuples $\langle 5, 13 \rangle$ and $\langle 8, 12 \rangle$ in Fig. 2(b) will not be transferred. The reason is that node 1 has the most number tuples with key 5 and node 2 with 8, and thus the destination nodes of the two tuples will be assigned to node 1 and node 2 respectively. In this case, the network cost of LAS is 3, which is much smaller than Hash and PRPD.

3.2 Comparison with Current Approaches

From a scheduling granularity viewpoint, we summarize the differences of the four techniques Hash, PRPD, Track and LAS, in Table 1. It can be seen that LAS is based on per-key level scheduling, thus it will be more powerful than Hash and PRPD on communication reduction. More specifically, LAS has inherited the advantages from Track so that it can avoid any redundant duplication for skew data, compared to PRPD. Moreover, for non-skew tuples, LAS also has tried to explore data locality, instead of just simply redistributing them.

Table 1. A General Comparison of Different Approaches

| Data/Alg. | Hash | PRPD [5] | Track [3] | LAS |
|---|-----------|------------------|-----------------------|-----------------------|
| skew | chunk, FR | per join key, FB | per distinct key, SBM | per distinct key, SBM |
| non-skew | chunk, FR | chunk, FR | per distinct key, SBM | per join key, LA |
| FR: <i>full redistribut.</i> , FB: <i>full broadcast</i> , SBM: <i>select broadcast & migration</i> , LA: <i>locality-aware</i> | | | | |

Compared to Track, LAS does not distinguish keys with a same value in R and S for the non-skew inputs. The used LA operation is much simpler than SBM, and thus LAS will be more lightweight than Track, especially when the

Algorithm 1 Implementation of LAS, parallel processing on each node i

```
Input:  $R_i, S_i, skew$ 
Output: schedule plan  $L_i(key, src, des)$ 

step 1: key statistics
1: initialize  $dk$  and  $jk$  in  $\text{Array}[\text{Map}[\text{int}, \text{int}]](n)$ 
2: for  $key \in R_i, S_i$  do //record frequency
3:    $h \leftarrow \text{hash}(|key|)$ 
4:   if  $key \in skew$  then
5:     increase freq of  $key$  in  $dk(h)$  by 1
6:   else
7:     increase freq of  $|key|$  in  $jk(h)$  by 1
8:   end if
9: end for
10: for  $j \leftarrow 0..(n-1)$  do //send to node  $j$ 
11:   send  $dk(j), jk(j)$  to  $dk.r(i), jk.r(i)$  at node  $j$ 
12: end for

step 2: locality exploration
13: initialize  $dk\_aggr$  in  $\text{Map}[\text{int}, \text{List}[\text{Triple}]]$ 
    and  $jk\_aggr$  in  $\text{Map}[\text{int}, \text{List}[\text{Pair}]]$ 

//aggregation on key
14: for  $j \leftarrow 0..(n-1), st \in dk.r(j)$  do
15:   put  $(|st.k|, (st.k, j, st.v))$  in  $dk\_aggr$ 
16: end for
17: for  $j \leftarrow 0..(n-1), st \in jk.r(j)$  do
18:   put  $(st.k, (j, st.v))$  in  $jk\_aggr$ 
19: end for
20: for  $entry \in dk\_aggr$  do //apply Track
21:   apply SBM on  $entry.v$ 
22:   for  $tr \in entry.v$  do
23:     put  $(tr.1, tr.2, \text{SBM}(tr.1))$  in  $L_i$ 
24:   end for
25: end for
26: for  $entry \in jk\_aggr$  do //apply LA
27:   get pair  $P$  with max.  $pair.v$  in  $entry.v$ 
28:   for  $pair \in entry.v$  do
29:     put  $(entry.k, pair.k, P.k)$  in  $L_i$ 
30:   end for
31: end for
32: collect each  $L_j$ 
```

number of unique keys is large. We have an additional skew quantification process in LAS. Nevertheless, as we have described in Section 2.2, the cost (of sampling) will be very small. Moreover, although we have used a relatively coarse way to schedule the non-skew tuples, our approach is still based on per-key, and thus network traffic can still be effectively reduced. In fact, as we will show in our later evaluation, LAS can achieve comparable performance on communication reduction to Track, and much better than other methods.

3.3 Parallel Implementation

LAS schedules each key independently, therefore it can be implemented in parallel in a distributed computing environment. The parallel implementation of LAS on each node i is given in Algorithm 1. We assume that we have obtained the skew keys. Then, the inputs of LAS are the two relations and the skew, and the output is a schedule plan including the source node and destination node(s) of each key. In the local statistic process, we count the appearing frequency of keys in skew and non-skew tuples in a separate way (lines 2-9). The recorded information is in the form of $\langle key, freq \rangle$ pairs, and each pair is collected by a specified bucket, based on the hash value of the key. We have to distinguish the keys with the same value but from different relations for the skew data (i.e., per distinct key). For simplicity, we have added a negative sign to each key from S , when we read the relation (assume all the input keys are positive integers). In this condition, when we count the appearing number for the non-skew keys, we just simply used their absolute values (lines 6-7). After that, similar to a hash redistribution, each bucket is pushed to the assigned remote node for further processing (lines 10-12).

Based on the received pairs from each node, the detailed scheduling of LAS is presented in lines 13-31 of Algorithm 1. All the keys are firstly aggregated, so

that we can get the frequency information of each key on each node. Then, we perform the SBM operations over the skew data (lines 20-25). For each key in the non-skew part, we simply scan the aggregated entries and search the node with the maximal value on *freq*. Then, the found node will be added into schedule plan as the destination node for the key (lines 26-31). All the scheduling process will be ended when the destination(s) of each key from each node is assigned. For the latter join executions, input tuples will be partitioned based on their destination nodes and then transferred to remote nodes for local joins.

4 Experimental Evaluation

4.1 Experiment Setup

We evaluate our approach over a cluster located at SUSTech in Shenzhen City in China. We use 4 CPU cores running at 2.80 GHz for each computing node with 64GB of RAM. The nodes are connected by Infiniband, and the operating system is Linux kernel version 2.6.32-431.

We compare LAS with the widely used Hash and the state-of-the-art Track, and have implemented the three methods using the programming language X10 [6], with version 2.3, compiling to C++ with gcc version 4.4.7. The evaluation is implemented on joins between two relations R and S . We fix the cardinality of R to 64 million and S to 1 billion records. Following a general way, we set the data format to $\langle key, payload \rangle$ pairs, where each *key* is an integer. We assume that R and S meet the foreign key relationship, and thus we only add skew to S , following the Zipf distribution. As listed in Table 2, besides a uniform distributed dataset, we have generated another three sets with different degrees of skew, by varying the Zipf factor from 0.8 to 1.1. It should be noted that joins with such characteristics and workloads are common in data warehouses [4].

Table 2. Details of Test Datasets and Meanings of used Parameters

| data set | skew | # unique | top1 | top10 | data set | skew | # unique | top1 | top10 |
|----------|------|-------------|------|-------|----------|------|------------|-------|-------|
| DS1 | 0 | 250,000,000 | 0.0% | 0.0% | DS3 | 1 | 46,947,295 | 4.7% | 13.8% |
| DS2 | 0.8 | 136,137,483 | 0.3% | 1.1% | DS4 | 1.1 | 19,966,276 | 10.7% | 28.7% |

X: the number of selected top skew keys, **Y**: the size of the payload for each tuple

In all our experiments, we set the system parameter **X10_NPLACES** to the number of cores. This lets us be able to focus on analyzing the performance of our approach in distributed computing environments rather than computing with multiple thread parallelism (as each *place* in X10 can be considered as a logical node). To capture the precise characteristics of LAS, we manually set the first top **X** keys as skewed keys and do not take the actually skew quantification time into account in our results. As a default, we set **X** to 4000, and we use 64 cores (16 nodes) and the data set DS3.

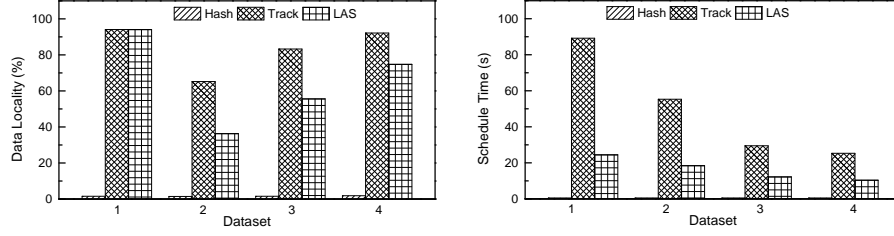


Fig. 3. Data locality and scheduling time by varying data skew over 64 cores (16 nodes).

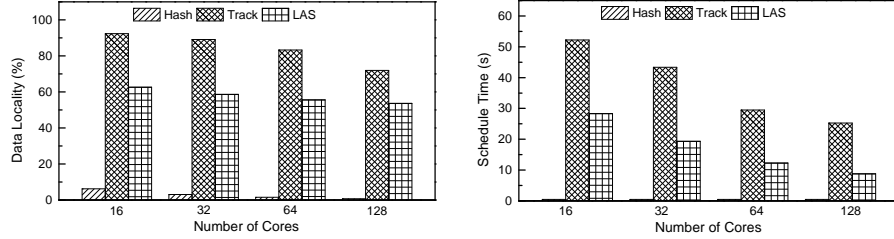


Fig. 4. Data locality and scheduling time by varying number of cores over $skew=1$.

4.2 Experimental Results

We measure the efficiency and effectiveness based on three metrics: data locality, scheduling time and join runtime. The first metric indicates the volume of network traffic, a high data locality indicates a light traffic load on a network.

Vary data skew. We run our tests using 64 cores (16 nodes) over the four different datasets. Fig. 3 shows the results of data locality and scheduling time of each algorithm. There, LAS and Track achieve much higher data locality than Hash in all the cases, demonstrating the effectiveness of the two approaches on reducing data communication. Specifically, their data locality is around 92% when the data is uniform distributed. This means the per-key level strategies can effectively explore data locality for non-skew tuples, and Hash or current skew handling techniques such as PRPD [5] have not considered such an optimization. Moreover, we can observe that the data locality of LAS is generally lower than Track. However, their results are still in a comparable range. In fact, as shown in our later results in Fig. 5, this kind of data locality difference can be decreased by increasing X in LAS. For scheduling time, it can be seen that LAS is always more lightweight than Track. Moreover, their scheduling time decreases with the increasing of data skew, the reason is that the number of unique keys decreases.

Vary number of nodes. We also test the three approaches over the system by varying the number of cores from 16 (4 nodes) to 128 over the default dataset. As shown in Fig. 4, the data locality of LAS and Track are always much higher than Hash. Although Track transfers less data than LAS, their difference is decreasing with increasing the number of cores. The possible reason is that the distribution of keys becomes sparse, and part of skew keys could become not so

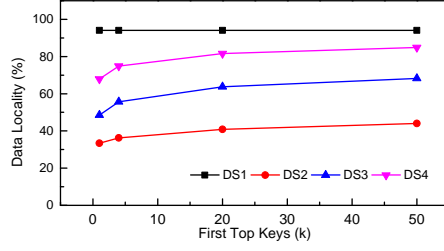


Fig. 5. Data locality with different \mathbf{X} .

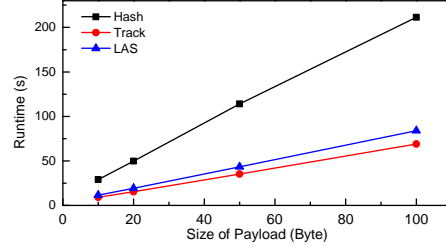


Fig. 6. Join comparison with different \mathbf{Y} .

skew, and thus the LA operation in LAS starts to perform similarly to the SBM in Track. For scheduling time, both the LAS and Track decrease with increasing the number of cores, showing the good scalability of the two algorithms.

Vary number of skew keys. LAS treats skew and non-skew keys in different ways. To show the impacts of the selected skew keys in our implementations, we vary the value of \mathbf{X} from 1000 to 50000 over the four datasets and present the results in Fig. 5. There, the data locality for DS1 keeps consistence, due to the dataset is uniform distributed. For the skew cases, the data locality is increasing with increasing the number of selected top skew keys. This is because the SBM operation used for skew keys can provide a more fine-grained control on reducing network traffic than the LA. In this experiment, we find that the scheduling time over each data set has only slight changes (in 1 sec) with increasing \mathbf{X} . The reason is that the number of selected keys is still much smaller compared to the whole unique keys. This also means that we can actually set \mathbf{X} to 50K rather than 4K as a default at least, to keep the data locality differences between LAS and Track smaller in the results presented in Fig. 3 and Fig. 4. For example, the data locality of LAS can be increased from 55% to 68% for DS3 by changing \mathbf{X} from 4K to 50K, without any increase on the scheduling time.

Join performance. We finally compare the join performance of the three approaches by varying the size of payload \mathbf{Y} for all tuples, from 10 Bytes to 100 Bytes. This means that the maximal dataset in size is around 100GB. For local join execution, we select the commonly used hash joins, i.e., hash table building and probing [4]. For each join, we only count the number of matches rather than materializing the output. Additionally, to avoid the network congestion in data transferring, we use a simple and efficient round-robin communication pattern in the joins [4]. The results are shown in Fig. 6. There, joins using LAS and Track perform better than Hash. However, we notice that when the dataset is small (e.g., $\mathbf{Y}=10$), the runtime difference between Hash and LAS (and Track) is only 20 secs. Considering the scheduling time of LAS is 12 secs and Track is 29 secs, we believe Hash could be the best choice for joins, when the number of keys is huge but the whole dataset in size is small. With the increase of \mathbf{Y} , the advantages of LAS and Track become obviously, indicating that these two approaches would be more suitable for large datasets (in size). Moreover, we can see that LAS can always achieve similar performance with Track, and their

difference is only 15 secs when the data reaches 100GB. Since LAS is around 17 secs faster than Track on scheduling for the case, we believe that LAS would be a better solution for moderate size datasets (e.g., 100GB or smaller). It should be noticed that, in real cases, a system optimizer will be able to get the possible cost of scheduling and network communication for each approach, and consequently to choose the best plan for executions. Nevertheless, detailed discussion on how to compute and compare the cost is outside the scope of this paper.

Brief summary. In general, LAS has applied per-key strategies on communication reduction, thus it always transfers much less data over networks than Hash. Compared to the state-of-the-art Track, LAS has adopted a relatively coarse-grained operation (i.e., LA) to large number of non-skew keys, and thus its scheduling is more lightweight, especially when the number of unique keys is large. From above results, we can see the LAS can always achieve comparable performance to Track, in terms of data locality and join runtime. In such scenarios, we believe that LAS can be considered as a new and efficient solution for distributed joins in large-scale distributed scenarios.

5 Related Work

Research towards optimizing main-memory joins has already achieved significant performance speedups through optimizations over modern processors. Nevertheless, as applications grow, join performance would be limited by the available computing cores or system memory [4]. The two conventional Hash and duplication-based methods offer the potential scalability on processing big data. However, they are far from network-optimal, because transferring all input tuples or broadcasting a single relation would incur a heavy time-cost. Moreover, Hash could meet serious load balancing issues when input data is skew.

As data skew is quite common in data applications, various advanced algorithms have been proposed to against join skew [4, 5]. The main idea of these approaches is keeping large number of skew tuples locally instead of transferring them over networks. This leads to obvious network traffic reduction in their join executions. However, all these methods focus on skew handling rather than reducing network traffic, and thus they are still not optimal. For example, although the work [4] proposes a *fetch on demand* method to process skew tuples, similar to PRPD [5], it has not explored data locality issues for non-skew data yet. In comparison, LAS provides a more fine-grained scheduling for all input tuples, and thus it can perform better on communication reduction.

To maximize data-locality, different data partitioning techniques have been proposed to avoid remote join operations for queries [7]. More generally, various advanced data placement and replication strategies have been proposed for data center storage systems to reduce the network overhead for particular workloads [8, 9]. Different from them, we focus on exploring data locality using online scheduling rather than pre-processing. On the other hand, although the state-of-the-art Track [3] is able to minimize network traffic, it applies complex schedules to all join keys, which could lead to heavy overhead in the presence of big data.

In comparison, LAS has used a simpler but effective way to handle input data, and thus LAS is more lightweight. More important, as we have shown in our experiments, LAS can reduce the network traffic significantly, and also can achieve comparable performance to Track.

Recent work has tried to optimize network time for distributed joins [10]. However, the optimization problem is NP-complete, making the technique can not be applied to per-key [3]. Although an efficient heuristic has been proposed for the optimization [11], its scheduling still performs based on data chunks (i.e., partitions) rather than individual keys. In contrast, LAS uses linear scheduling applied to each join key and thus it is more powerful on communication reduction. On the other hand, LAS can be used in conjunction with these techniques to optimize network communication time at a more fine-grained granularity.

6 Conclusions

In this paper, we focus on effective and efficient scheduling techniques to reduce network traffic for distributed joins. We have discussed the possible performance issues of current approaches and proposed the LAS algorithm on that basis. We have described the detailed design and implementation of LAS, and experimentally shown that LAS is lightweight and can achieve comparable performance on communication reduction, compared to the state-of-the-art. Moreover, we have also shown that LAS can obviously outperform the conventional approaches such as Hash in both communication reduction and join runtime. Our future work mainly lies in extending the proposed scheduling approach in more complex environments such as mobile and cloud computing systems [12, 13].

Acknowledgments. Part of this work was supported by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 799066. The computations were performed on the Inspur TS10K Cluster at the High Performance Computing Center in SUSTech.

References

1. Greenberg, A., Hamilton, J.R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D.A., Patel, P., Sengupta, S.: VL2: a scalable and flexible data center network. *Communications of the ACM* **54**(3) (2011) 95–104
2. Cheng, L., Wang, Y., Pei, Y., Epema, D.: A coflow-based co-optimization framework for high-performance data analytics. In: *ICPP*. (2017) 392–401
3. Polychroniou, O., Sen, R., Ross, K.A.: Track join: distributed joins with minimal network traffic. In: *SIGMOD*. (2014) 1483–1494
4. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Improving the robustness and performance of parallel joins over distributed systems. *Journal of Parallel and Distributed Computing* **109** (2017) 310–323
5. Xu, Y., Kostamäa, P., Zhou, X., Chen, L.: Handling data skew in parallel joins in shared-nothing systems. In: *SIGMOD*. (2008) 1043–1052

6. 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. *ACM SIGPLAN Notices* **40**(10) (2005) 519–538
7. Zamanian, E., Binnig, C., Salama, A.: Locality-aware partitioning in parallel database systems. In: *SIGMOD*. (2015) 17–30
8. Yang, Z., Hoseinzadeh, M., Andrews, A., Mayers, C., Evans, D.T., Bolt, R.T., Bhimani, J., Mi, N., Swanson, S.: AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In: *IPCCC*. (2017) 1–8
9. Yang, Z., Wang, J., Evans, D., Mi, N.: AutoReplica: automatic data replica manager in distributed caching and data processing systems. In: *IPCCC*. (2016) 1–6
10. Rödiger, W., Muhlbauer, T., Unterbrunner, P., Reiser, A., Kemper, A., Neumann, T.: Locality-sensitive operators for parallel main-memory database clusters. In: *ICDE*. (2014) 592–603
11. Cheng, L., Li, T.: Efficient data redistribution to speedup big data analytics in large systems. In: *HiPC*. (2016) 91–100
12. Mao, Y., Wang, J., Sheng, B.: Mobile message board: Location-based message dissemination in wireless ad-hoc networks. In: *ICNC*. (2016) 1–5
13. Wang, J., Yao, Y., Mao, Y., Sheng, B., Mi, N.: Fresh: Fair and efficient slot configuration and scheduling for Hadoop clusters. In: *CLOUD*. (2014) 761–768