FastJoin: A Skewness-Aware Distributed Stream Join System

Shunjie Zhou[†], Fan Zhang[†], Hanhua Chen[†], Hai Jin[†], Bing Bing Zhou[‡]
[†]National Engineering Research Center for Big Data Technology and System
Cluster and Grid Computing Lab
Service Computing Technology and System Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
[‡]School of Information Technologies
The University of Sydney, NSW 2006, Australia
Emails:{zhoushunjie, zhangf, chen, hjin}@hust.edu.cn, bing.zhou@sydney.edu.au

***Abstract*—In the bigdata era, many applications are required to perform quick and accurate join operations on large-scale real-time data streams, such as stock trading and online advertisement analysis. To achieve high throughput and low latency, distributed stream join systems explore efficient stream partitioning strategies to execute the complex stream join procedure in parallel. Existing systems mainly deploy two kinds of partitioning strategies, i.e., random partitioning and hash partitioning. Random partitioning strategy partitions one data stream uniformly while broadcasting all the tuples of the other data stream. This simple strategy may incur lots of unnecessary computations for low-selectivity stream join. Hash partitioning strategy maps all the tuples of the two data streams according to their attributes for joining. However, hash partitioning strategy suffers from a serious load imbalance problem caused by the skew distribution of the attributes, which is common in real-world data. The skewed load may seriously affect the system performance.**

In this paper, we carefully model the load skewness problem in distributed join systems. We explore the key tuples which lead to the heavy load skewness, and propose an efficient key selection algorithm, *GreedyFit* to find out these key tuples. We design a lightweight tuple migration strategy to solve the load imbalance problem in real-time and implement a new distributed stream join system, *FastJoin*. Experimental results using real-world data show that FastJoin can significantly improve the system performance in terms of throughput and latency compared to the state-of-the-art stream join systems.

***Index Terms*—distributed stream join system; data skew; dynamic load balancing; load migration**

I. Introduction

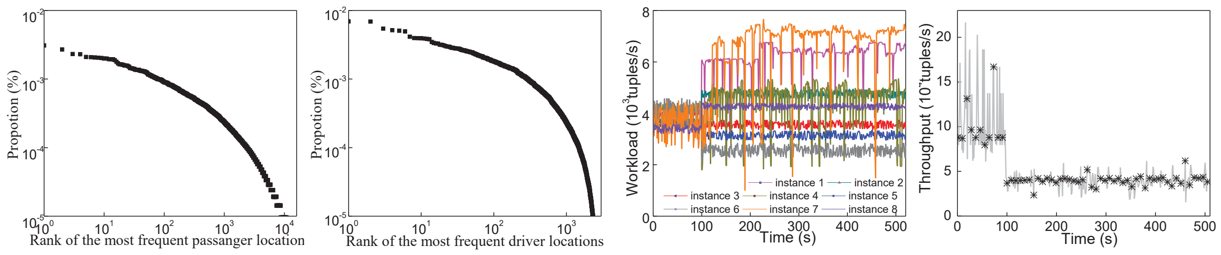
Many applications, such as advertisement analyzing [2], on-demand ride-hailing [33], and high frequency algorithmic trading [29], require real-time process and analysis of large-scale datasets which are generated in the form of high-speed data streams [12, 13, 17, 20]. In these applications, *stream join* is one of the most basic and important operations. For example, Google uses Photon [2] for advertisement analysis by joining two data streams, a query stream which is generated when users perform search on Google website, and an advertisement click stream which is generated when users click an advertisement provided by Google. For another example, DiDi company builds an on-demand ride-hailing platform [33] for real-time taxi order dispatching, which keeps joining the order stream of passengers and the driving track steam of taxis.

Efficient stream join is a more challenging task compared to traditional join operations since it faces continuous unbounded data, and the operation is often required to be performed in real, or near real time. Performing efficient stream join has the following basic requirements: 1) low processing latency and high throughput – these are critical requirements for most real-time data analysis systems; 2) memory efficiency and scalability – stream join should be processed in memory. However, memory size is limited but the data are unbounded. This requires the design of highly efficient and scalable join algorithms; 3) completeness, that means each pair of tuples from two streams that are matched for join must be joined exactly once. This requires a proper stream data scheduling algorithm.

Existing parallel or distributed stream join algorithms [4, 27, 30] commonly try to distribute stream input data evenly among processing nodes and guarantee that each pair of tuples from different streams are joined exactly once. This kind of simple data partitioning strategy works well for non-hash joins. However, it is inefficient for hash based join operations. This is because the associated tuples are likely to be placed on different processing nodes, which needs excessive additional computation and communication costs for performing a join operation. Hash-join is widely used in real applications such as order-matching [33], Google advertising [2], log analyzing [14]. Such applications need to join two streams by the same key of a tuple. To perform hash-join operation, a better strategy is to use *hash partitioning* so that the associated tuples will be given the same hash or key value and directly placed on the same processing node for joining [16, 22]. Though hash partitioning can reduce a large amount of unnecessary computation and communication operations, it may lead to data skew and imbalanced workloads among the processing nodes can result in unacceptable high processing latency and low system throughput.

* The Corresponding Author is Hanhua Chen (chen@hust.edu.cn).

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00111



(a) Skewness of the location distribution in the passenger order stream (b) Skewness of the location distribution in the taxi track stream (c) The workload of each join instance in BiStream (d) The system throughput of BiStream

Fig. 1: The skewed data distribution leads to highly load imbalance and low throughput in BiStream

Data skew is a notorious problem in hash partition based distributed systems as well as the distributed stream join system in this work. In the following, we conduct an experiment to reveal how data skew leads to performance degradation in distributed stream join system. We examine the on-demand ride-hailing application deployed on a distributed stream join system following the hash join. In the experiment, we use the real-world dataset collected from DiDi Chuxing in Chengdu, China in November 2016 [1]. The dataset consists of two kinds of stream data, the query orders of passengers and the driving tracks of taxis. The order is dispatched to the nearest taxis.

Figure 1(a) and Figure 1(b) depict the key distributions of the passenger order stream and the taxi track stream. We can see that both distributions are highly skewed. About 20 percent of the locations occupies 80 percent of all the passenger orders, while about 24 percent of the locations occupies 80 percent of all the taxi driving tracks. In Figure 1(c), different lines represent the workloads of different stream join instances created by the system. We can see that in the very beginning the workloads of all join instances are similar, but soon later significant workload differences appear across join instances. The result indicates the system convergences to the status with serious load imbalance. Figure 1(d) shows the overall system throughput. We can see that the more serious the degree of load imbalance is, the lower overall throughput the system can provide. Hence, we claim that the existing hash based stream join systems suffer highly load imbalance and low throughput in the present of data skew of real world systems.

Therefore, an interesting question is how to effectively deal with the problem of data skew, or load imbalance when using the hash partitioning. In this paper, we introduce a novel skewness-aware distributed stream join system, called FastJoin. FastJoin is based on the BiStream [16], the state-of-the-art distributed join processing system. Compared to previous parallel architectures for stream joining, it is more scalable in storage and computation. By leveraging increasing amount of storage and computation resources in cluster nodes, a distributed stream join system has the potential to support near full-history join (by storing one of the streams among multiple nodes in the cluster).

To deal with the load imbalance problem which occurs in BiStream frequently, FastJoin adopts a dynamic load balancing technique to effectively detect data skew and migrate the data from heavily loaded processing nodes to lightly loaded ones while keeping the overhead as low as possible. We implement FastJoin on Apache Storm [31], one of the most popular stream processing systems, and use large-scale real-world datasets to evaluate the effectiveness of our system. The experimental results show that, compared to the state-of-the-art system BiStream, our FastJoin system can increase the system throughput by 25.2 percent, and decrease the average processing latency by 17.6 percent.

The paper is organized as follows. Section II discusses the related work about stream join systems. Section III introduces the detailed design of our FastJoin system. Section IV gives the theoretical analysis of our design. Section V presents the details of the FastJoin system implementation. Section VI evaluates the performance of this design. Section VII concludes the paper.

II. Related Work

Parallel/distributed stream join algorithms/systems have recently attracted much research interest. Teubner et al. [30] introduce a parallel join algorithm called Handshake Join. In this algorithm two streams flow across all the processing nodes in opposite directions. The join operations of tuple pairs are performed when they arrive at the same processing node. Daniele Buono et al. [4] propose a promotion algorithm to accelerate the join processing based on NUMA-like shared-memory parallel architectures. It organizes all workers, which are implemented as threads, as a join-matrix. Workers in the same row can share the memory of the core and can execute the join processing without replication of streams. Another algorithm called SplitJoin is proposed by Najafi et al. [22]. This algorithm broadcasts the tuples in the same order to all the processing nodes, and each node keeps a specific part of tuples and performs join operations independently. Existing designs are efficient when processing fixed-size window all-pair join operations. However, they may not be efficient for hash stream join. Existing designs are also not easy to scale and difficult to tolerate node failure [15].

Google designs a distributed stream join processing system called Photon for advertisement analysis [2]. Photon can be deployed in a system with nodes placed at geographically different locations and is also easily scalable to deal with data of large-scale. Photon tried to solve the problem of load imbalance which occurs in the hash-join by simply replaying

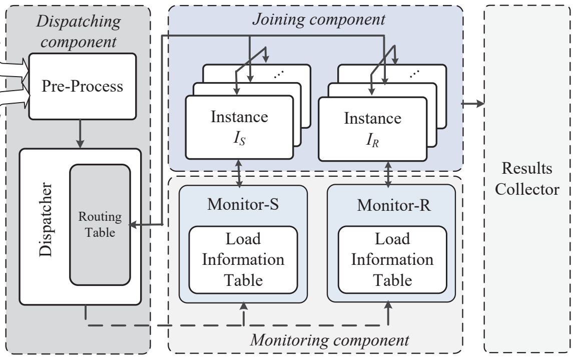


Fig. 2: The structure of FastJoin

the streams. However, it cannot guarantee that all the relevant tuples are joined exactly once.

Elseidy et al. [8] propose a distributed stream join system SQUALL, which is based on a join-matrix model. In this design, all the processing nodes are organized in the form of a matrix. For one stream, a tuple is dispatched and replicated to all the nodes in a specific row. On the other hand, each tuple of the other stream is dispatched and replicated to all the nodes in a specific column. In this way all the tuple pairs can be joined exactly once. However, such a design generates a huge amount of data replications. It is memory inefficient and thus hard to scale.

Lin et al. [16] introduce a space-efficient and scalable distributed stream join system, BiStream. BiStream leverages a join-biclique model in which the tuples in each stream are partitioned and stored once without replication. A hash partitioning strategy is used for low-selectivity join. This strategy places the tuples of the same hash value on the same processing nodes to make the join operations more efficient. As discussed in the previous section hash partitioning has a potential problem of load imbalance. To tackle the load balancing problem BiStream adopts a so-called hybrid routing strategy, ContRand. However, it is essentially a simple static load distribution strategy and thus not very effective in dealing with load imbalance since workloads on different processing nodes vary dynamically during the computation and are hard to predict.

Load balancing is also an important issue in common distributed stream processing system. However, existing designs in common distributed stream processing system can only cope with simple operations such as selection in stream scenario [5, 9, 11, 23, 24, 28, 32] while they are not applicable to the complex stream join operation. Some other work for non-stream scenario [3, 21, 25, 26] is also not suitable for hash-join operation in stream scenario.

III. System Design

In this section, we first briefly describe system architecture of our FastJoin system following the framework of the join-biclique model of BiStream [16]. Then, we present the dynamic load-balancing strategy by introducing a load quantification model, a key selection algorithm GreedyFit for migration, and the overall migration procedure adopted in FastJoin for dynamic load balancing. Finally, we describe the implementation of the window-based join in our FastJoin model. Table I lists the notations used in this design.

A. System Overview

Efficient distributed stream join systems need to have the following basic requirements: 1) low processing latency and high throughput, 2) memory efficiency and scalability, and 3) completeness. Our design follows the join-biclique model proposed by Lin et al. [16]. The join-biclique model with hash partitioning can achieve completeness and scalability in previous section. Hence, in this design, we mainly focus on solving the problem of performance degradation due to load imbalance among the worker instances in distributed stream join system (shown in Fig. 1).

The join-biclique model separates join operation component into two symmetry sub-components. It organizes all worker instances as bipartite graph, where each part of the graph only stores the tuple from one stream. The working principle of both parts of join-biclique are the same, so we just specify one of the two parts. Suppose there are two joining streams named R and S. The basic idea of join-biclique is as follow. When the system partitions $r \in R$, the dispatcher first uses some partitioning method to send tuple r to some worker instance which only store tuples of stream R. Then, it will send tuple r to some worker instances which store stream S to execute join processing. After joining all tuples, the tuple r will be discarded in worker instances which store stream S. The join-biclique is memory efficient because it nearly has no replications (actually, the replication just exists in a very short period of time which can be ignored).

Based on the join-biclique model, we design our skewness-aware distributed stream join system, FastJoin. FastJoin detects the computational workloads of the join instances and perform data migration for dynamic load-balancing in real time. FastJoin consists of three main components and its architecture is depicted in Figure 2. The first one is a *dispatching*

TABLE I: List of Notations

Notations	Description
R, S	Two data streams for joining
I_{R-i}, I_{S-i}	The i-th join instance which stores tuples from *R*, *S* and joins tuples from *S*, *R*
R_i, S_i	The set of stream tuples stored in I_{R-i}, I_{S-i}
$\lvert R_i\rvert$, $\lvert S_i\rvert$	The number of tuples in R_i, S_i
$\lvert R_{ik}\rvert$, $\lvert S_{ik}\rvert$	The number of tuples with key *k* in R_i, S_i
φ_{ri}, φ_{si}	The queue length of the tuples from R, S
K	The number of keys in an instance
SK	The set of keys for migration
L_i	The load of I_{R-i}
LI	The degree of load imbalance
Θ	The load imbalance threshold

component which collects input tuples from the two streams, partitions and distributes the tuples to the most suitable join instances. The second one is the *joining* component which consists of two groups of join processing instances to perform tuple store and join operations. The third is the *monitoring* component which detects the load imbalance among different join instances and makes migration decisions.

The dispatching component has three units. A *pre-processing unit* collects the input tuples of both streams and performs some pre-processing operations such as ordering or certain user-defined functions. The pre-processed tuples will be sent to a *dispatcher*. The dispatcher first partitions tuples according to a data partitioning strategy and then sends each tuple partition to a specific join processing instance. Since we mainly consider how to solve the load imbalance problem for low-selectivity stream join, in this paper we assume the hash partitioning strategy is used, that is, a hash function is performed on each tuple and tuples with the same key are dispatched to the same join instance. After tuples associated with the same keys are migrated from one join instance to another, the dispatcher records the migration information in a *routing table*. The dispatcher checks the routing table to dispatch the tuples to the right join instances.

The joining component contains a set of *join instances* which are divided into two groups, each responsible for storing tuples from one stream and performing join operations for the other stream, respectively. For dynamic load balancing each join instance maintains two counters in memory, one counting the number of stored tuples from one stream and the other recording the number of incoming tuples from the other stream for joining. The information will periodically be sent to the monitoring component. When receiving the instruction for data migration from the monitor, the join instance pairs will communicate with each other to send/receive the tuples for load balancing. After that, the instance will notify both the monitor and the dispatcher to update their load information table and routing table accordingly.

There are two *monitors* in the monitoring component, each being for one group of join instances. The monitors periodically receive the load statistics from the join instances to detect the load imbalance. The load information is stored in a *load information table*. When the load imbalance occurs, the monitor will determine which join instances should offload/upload tuples to/from which join instances and inform those instances for load migration. After the migration, the monitor updates its load information table accordingly.

B. Load Quantification Model

For dynamic load balancing it is important to accurately quantify the workload of each join instance so that the monitor can precisely determine which instances are heavily loaded and which ones are lightly loaded. The main workload of an instance is essentially store and join operations on tuples from different streams. To simplify the discussion, in the following we only discuss the workload on the first group of join instances which store tuples of stream *R*, and perform join operation on tuples of stream *S*. The discussion for the other group of join instances is the same and thus omitted.

On a join instance I_{R-i} (i.e., the i-th join instance in the first group), each time when a tuple of stream S arrives, it should be compared with all the tuples of stream *R* stored in I_{R-i} and join with the tuples which have the same key. Thus, the current workload L_i on instance I_{R-i} can be written as the product of the total number of tuples stored in I_{R-i} (denoted as $|R_i|$), and the queue length of the tuples from stream S (denote as φ_{si}), that is,

$$L_i = |R_i| * \varphi_{si} \tag{1}$$

Each time when the monitor receives the statistics of all the join instances, it can immediately find the instance with the heaviest workload as well as the one with the lightest workload. (Note we assume that there is a loosely synchronized clock across the join instances and the accuracy is not an issue since the migration can never take place frequently.) To determine if the load migration should take place, we define a degree of load imbalance LI which is the ratio of the heaviest workload to the lightest workload as follow,

$$LI = \frac{L_{heaviest}}{L_{lightest}} \tag{2}$$

The degree of load imbalance is always greater than or equal to one. The larger LI is, the more severe is the load imbalance. When the value LI exceeds a certain threshold Θ, the monitor will notify the instances with the heaviest/lightest workloads to perform load migration.

For each key k on instance I_{R-i} we denote the number of the corresponding tuples in stream R as $|R_{ik}|$, and the number of tuples with the same key from stream S as φ_{sik}. Then $|R_i|$ and φ_{si} can be quantified by,

$$|R_i| = \sum |R_{ik}| \tag{3}$$

$$\varphi_{si} = \sum \varphi_{sik} \tag{4}$$

If we migrate all the tuples whose key is k from instance I_{R-i} to I_{R-j}, after the migration, the workloads of instances I_{R-i} and I_{R-j} will become,

$$\begin{aligned} L_i' &= (|R_i| - |R_{ik}|) * (\varphi_{si} - \varphi_{sik}) \\ &= L_i - |R_{ik}| * \varphi_{si} - |R_i| * \varphi_{sik} + |R_{ik}| * \varphi_{sik} \end{aligned} \tag{5}$$

$$\begin{aligned} L_j' &= (|R_j| + |R_{ik}|) * (\varphi_{sj} + \varphi_{sik}) \\ &= L_j - |R_{ik}| * \varphi_{sj} - |R_j| * \varphi_{sik} + |R_{ik}| * \varphi_{sik} \end{aligned} \tag{6}$$

It is interesting to see from the above two equations that the decreased load from instance I_{R-i} may not be equal to the increased load on I_{R-j}. That means if we randomly select some keys and migrate the corresponding tuples, we could potentially bring more workload for instance I_{R-j} than the workload we saved for instance I_{R-i}. To deal with this issue we design a key selection algorithm, GreedyFit. It is discussed in the following subsection.

C. GreedyFit Algorithm

The key selection algorithm is to choose a set of keys and the tuples with the keys in this key set will be migrated from one instance with the heaviest load to the one with the lightest load. The goal of the key selection algorithm is to balance workloads on these two instances by decreasing the load of the heaviest instance as much as possible, while increasing the load of the lightest instance as little as possible.

Assume the number of keys on instance I_{R-i} is N. Theoretically there are exponential kinds of possible combinations to form the key set for migration. Actually, the key selection problem can be modeled as a kind of 0-1 knapsack problem: choose a subset of keys $SK = \{k\}$ from a universe key set, and fill the gap between the load of the heaviest instance and the lightest instance (i.e., $L_i - L_j$) as much as possible and at the same time, the number of the corresponding tuples to be migrated is as small as possible. It is well known that the 0-1 knapsack problem is essentially an NP-complete problem [10] so that it is hard to find an optimal solution.

In practice, when selecting keys for migration, an instance I_{R-i} must stop executing the store and join operations. Moreover, it takes longer communication time to migrate more tuples between the instances. These may result in longer processing latency. To solve the problem, we design a greedy algorithm, GreedyFit. The main idea of GreedFit is to order all the keys by their importance for migration, and select the keys based on this order.

To quantify the importance of each key for migration, we consider two aspects. Firstly, if we migrate a key, say key k from instance I_{R-i} to I_{R-j}, the difference of the workloads between the two instances (i.e., $L'_i - L'_j$) after the migration should be less than the difference before the migration (i.e., $Li-Lj$). Secondly, the tuples with key k stored in the heaviest instance (i.e., $|R_{ik}|$) should not be very large. Accordingly, we give the following definitions.

Definition 1 (Migration benefit) *The migration benefit of a key is the difference of the workloads which can be reduced by migrating all the tuples associated with key k from instance I_{R-i} to I_{R-j}.*

The migration benefit can be represented as,

$$F_k = (L_i - L_j) - (L'_i - L'_j) \tag{7}$$

Combining Eq. (7) with Eqs. (5) and (6), we obtain,

$$F_k = (|R_i| + |R_j|) * \varphi_{sik} + (\varphi_{si} + \varphi_{sj}) * |R_{ik}| \tag{8}$$

Definition 2 (Migration key factor) *The migration key factor of a key is the ratio of its migration benefit to the number of its corresponding tuples, i.e., $\frac{F_k}{|R_{ik}|}$.*

It is not difficult to see that the key with a larger migration key factor are more worth migrating than the key with a smaller one. With such a quantification, our GreedyFit algorithm is designed as follows.

The source instance I_{R-i} collects the statistics of the target instance I_{R-j} (i.e., $|R_j|$ and φ_{sj}), computes both the migration benefit and migration key factor for each key originally dispatched to instance I_{R-i}, and sorts all the keys with respect to their migration key factors in descending order, and finally tries to add each key into the selection key set in turn.

To judge whether a key should be selected for migration, GreedyFit considers the following condition: the workload of the target instance will not exceed the source target after the migration. This condition can be formalized as,

$$\begin{aligned}
\Delta L &= L'_i - L'_j \\
&= (|R_i| - \sum_{t=1}^{|SK|} |R_{ik}|) * (\varphi_{si} - \sum_{t=1}^{|SK|} \varphi_{sik}) \\
&\quad - (|Rj| + \sum_{t=1}^{|SK|} |R_{ik}|) * (\varphi_{sj} + \sum_{t=1}^{|SK|} \varphi_{sik}) \\
&= |R_i| * \varphi_{si} - |R_j| * \varphi_{sj} - \sum_{t=1}^{|SK|} ((|R_i| + |R_j|) \\
&\quad * \varphi_{sik} + (\varphi_{si} + \varphi_{sj}) * |R_{ik}|) \\
&= L_i - L_j - \sum_{t=1}^{|SK|} F_t > 0
\end{aligned} \tag{9}$$

GreedyFit checks ΔL when trying to add a key to the selection key set and terminates the process when ΔL becomes zero, or all the keys have been checked. The details of GreedyFit is shown in Algorithm 1.

Algorithm 1: GreedyFit Algorithm

```
1  Input: $|R_i|$, $|R_j|$, $\varphi_{si}$, $\varphi_{sj}$, $\theta_{gap}$
2  Output: A key set for migration
3  $FArray \leftarrow$ An initial empty vector;
4  $SK \leftarrow$ An initial empty set;
5  $Gap \leftarrow |R_i| * \varphi_{si} - |R_j| * \varphi_{sj}$;
6  for each key $k$ do
7      $F_k \leftarrow (|R_i| + |R_j|) * \varphi_{sik} + (\varphi_{si} + \varphi_{sj}) * |R_{ik}|$;
8      $FArray$.add($k$, $F_k$);
9  end
10 Sort $FArray$ by $\frac{F_k}{|R_{ik}|}$ in descending order;
11 for each element $e$ in $FArray$ do
12     if Gap $> e.F_k$ and $e.F_k \geq \theta_{gap}$ then
13         $Gap \leftarrow Gap - e.F_k$;
14         $SK$.add($e.k$);
15     end
16 end
17 return $SK$.
```

D. Migration Procedure

When instances receive the signals from the monitor for migration, the source instance stops store and join operations, and starts GreedyFit algorithm to find the key set for migration. Then the source instance will notify the target instance and the two instance starts to send/receive the tuples with the corresponding keys.

The source instance also notifies the dispatcher to update its routing table. This is performed at the end of the migration procedure. During the load migration, the source instance may still receive tuples with the selected keys from the dispatcher. It temporally stores the tuples of both streams *R* and *S* in a queue and then send them to the target instance after the dispatcher updated its table and notified the source instance. This is to guarantee the completeness of join. We might choose to notify the dispatcher to modify the routing table as soon as the instance completes the GreedyFit algorithm. However, during the load migration, the dispatcher may send the newly arrived tuples of stream *S* with a certain key *k* to the target instance, while the tuple of stream *R* with key *k* are still on the way to the target instance. This will result in an incomplete join. The detailed migration procedure is shown in Algorithm 2.

E. Window-based Join

FastJoin can be adjusted to support window-based join. In order to implement window-based join semantic, several components need to be modified, monitor component and joining component need to be modified at the same time.

Monitor component needs to record the historical accumulation of one storing stream $|R|$ $(or\ |S|)$ and the input rate of another joining stream φ_S $(or\ \varphi_R)$ of all worker instances. As for recording φ_S $(or\ \varphi_R)$, there is no need to be modified anywhere because we have considered the φ_S $(or\ \varphi_R)$ will fluctuate. When the tuples of one joining stream are discarded, we can still record the φ_S $(or\ \varphi_R)$. However, we have to change the data structure which records $|R|$ $(or\ |S|)$. For each worker instance, we use a fixed-size vector, which can be seen as a window, to record $|R|$ $(or\ |S|)$. The size of vector indicates the size of joining window. Every element in the vector means $|R|$ $(or\ |S|)$ in sub-window. When the expired tuples are removed in worker instance, the head of vector (early sub-window) would be popped out so that we can record the load of each worker instance in window-based join semantic.

Joining component needs to record local statistic. Like monitor component, joining component also needs to record the historical accumulation of one storing stream and the input rate of another joining stream of itself. Unlike monitor component, however, we do not need to change the data structure. We just need to decrease the value which stores $|R|$ $(or\ |S|)$ when the expired tuples are removed.

Algorithm 2: Migration

```
1  Tupleset ← An initial empty set;
2  $SK$ ← GreedyFit();
3  for each tuple $t$ in stream R storage do
4      if $t.key \in SK$ then
5          Tupleset.add($t$);
6          Remove $t$ from Stream R storage;
7      end
8  end
9  Send a migration start signal to $I_{R-j}$;
10 Send Tupleset to $I_{R-j}$;
11 Send a migration signal to the dispatcher;
12 Send ($SK$, $I_{R-j}$) to the dispatcher;
13 Send a migration end signal to $I_{Rij}$;
```

IV. Analysis

In this section, we analyze the efficiency and effectiveness of our GreedyFit algorithm and the scalability of FastJoin.

A. Discussion about the Key Selection Algorithm

As we have discussed in Section III-C, the key selection problem can be modeled as a 0-1 knapsack problem. There are many algorithms for solving the 0-1 knapsack problem. For example, dynamic programming is one of the most efficient technique which can find the optimal result in $O(KC)$ time, where K is the number of keys, and C is the capacity of the knapsack. In our stream join problem, the capacity of the knapsack is equal to $L_i - L_j$, which can be a very large value. The branch-and-bound algorithm [7] can also be used to solve this problem. However, the computational complexity may be as large as $O(2^K)$ in the worst case, which is not suitable for real-time stream processing.

In our GreedyFit algorithm, we need to sort the keys with respect to migration key factors and then check each key in turn. The time for sorting is $O(K \log K)$ and the time for computing the migration key factors and selecting keys is $O(K)$. Thus the total computational complexity is $O(K \log K)$. For recording the migration key factors, we need to use a vector to store the migration key factors for each key. The space complexity is then $O(K)$, where K is the number of keys in a join instance.

Both the time and space complexity of our GreedyFit algorithm are much lower than that of the regular join operation. In the regular join operation, each time when a tuple arrives, if it is from stream *R*, it is stored directly with a computational complexity of $O(1)$. If it is from stream S, it should be checked against all the tuples stored in this instance, and thus the computational complexity is $O(|R_i|)$. The instance mainly stores the tuples of stream R. Thus, the space complexity is also $O(|R_i|)$. Since $|R_i|$ is always much larger than K, GreedyFit algorithm will not incur significant overhead.

It is well known that greedy algorithms may not find a near optimal solution. For some applications that the processing latency is not critical, we can also use some advanced algorithms for obtaining a potentially better result. There are also some heuristic algorithms such as simulated annealing [6] which can be used for achieving near optimal solutions. The simulated annealing is essentially a random search technique. It starts a random initial solution, and iteratively finds a new solution based on the current solution. The simulated annealing is proved to converge, and the quality of the solution depends on the number of iterations. There is always a trade-off between the processing time and the solution quality.

For the key selection problem, we also design a SAFit algorithm using simulated annealing. There are four important parameters in a simulated annealing algorithm: the initial temperature *T*, the iteration times on each temperature *L*, the attenuation coefficient of the temperature a, and the termination temperature T_{min}. We start the algorithm with an initial solution by randomly picking up keys. We give each key a *flag* to sign whether the key is selected or not. Each time

Algorithm 3: SAFit Algorithm

```
Input: $|R_i|$, $|R_j|$, $\varphi_{si}$, $\varphi_{sj}$, $T$, $T_min$, $a$, $L$
Output: A key set for migration
$SK_{old} \leftarrow$ An initial empty set;
for each key $k$ do
    Randomly set $k.flag$ from {0,1};
    if $k.flag$=1 then
        $SK_{old}$.add($k$);
        if Benefit($SK_{old}$) $> L_i - L_j$ then
            $SK_{old}$.delete($k$);
            $k.flag$=0;
            Break;
        end
    end
end
$SK_{best} \leftarrow SK_{old}$ ;
$SK_new \leftarrow SK_{old}$;
while $T > T_{min}$ do
    for $i = 1$ to $L$ do
        Randomly choose one key $k$;
        $k.flag$ = $1 - k.flag$;
        Update $SK_{new}$ by adding/deleting $k$;
        if Benefit($SK_{new}$)$\leq L_i - L_j$ then
            if Value($SK_{new}$)> Value($SK_{old}$) then
                $SK_{old} \leftarrow SK_{new}$;
                if Value($SK_{old}$)> Value($SK_{best}$) then
                    $SK_{best} \leftarrow SK_{old}$;
                end
            else if Rand()¡$P(SK_{new}, SK_{old}, T)$ then
                $SK_{old} \leftarrow SK_{new}$;
            else
                $k.flag$ = $1 - k.flag$;
                $SK_{new} \leftarrow SK_{old}$;
            end
        else
            $k.flag$ = $1 - k.flag$;
            $SK_{new} \leftarrow SK_{old}$
        end
    end
    $T = T * a$;
end
return $SK_{best}$;
```

when we find a new solution, we randomly pick up one key and change its flag, i.e., if it is selected (or unselected) in the current solution, we unselect (or select) it. We only consider the new solution that satisfy the conditions that formalized in Eq. (9). Thus we will first check the sum of migration benefit $Benefit(SK)$=$\sum_{k \in SK} F_k$. Then we will compare the value of the new solution with the current solution, by using the following quantification function,

$$Value(SK) = \frac{\sum_{k \in SK} F_k}{\sum_{k \in SK} |R_k|} \tag{10}$$

If the new solution has a higher value, we change the current solution with the new solution directly. Otherwise, we accept the solution with a probability $P(SK_{new}, SK_{old}, T)$, which is computed by,

$$P(SK_{new}, SK_{old}, T) = e^{\frac{Value(SK_{new}) - Value(SK_{old})}{T}} \tag{11}$$

Such a computation is commonly used in a simulated annealing algorithm which follows a MetropolisHastings Algorithm [19]. We repeatedly try to find new solutions iteratively, and decay the temperature $T = T * a$ after each L iterations. The algorithm terminates when the temperature T is less than T_{min}. We show the details of SAFit in Algorithm 3. We compare the performance of SAFit algorithm and our GreedyFit algorithm in Section VI.

B. Effectiveness of GreedyFit

Through a greedy algorithm, GreedFit can always find an approximate result which can guarantee to reduce the degree load imbalance in the most cases.

Let L_o be the second heaviest workload on a join instance and L_p be the second lightest load on a join instance before the migration. Since $L'_i < L_i$ and $L'_j > L_j$ after the migration, the new heaviest load among the join instances is either L'_i or L_o, and the lightest load is either L'_j or L_p. The new degree of load imbalance LI' after the migration is then equal to $LI' = \frac{max(L'_i, L_o)}{min(L'_j, L_p)}$. As a result, $LI' < \frac{L_i}{L_j} < LI$.

C. System Scalability

We measure scalability by two properties following the discussion of Lin et al. [16]. One property is the flexibility of adding or removing the join instances [18] while the other one is effectiveness of utilizing the memory of newly added instances, which can be measured by *scaling gain ratio* (SGR) [16].

For the first property, since our FastJoin design depends on the join-biclique model, it will be similar with BiStream. As for the second property of memory utilization, we show that the SGR value of FastJoin is very close to one, which means almost 100% of memory can be shared to store new tuples when new join instances are added.

Compared with BiStream, FastJoin may need some additional memory space. During the load migration the lightest instance needs to store tuples of the other stream with the selected keys temporally. FastJoin also needs extra memory to store the statistics about $|R|$ and φ_s. Suppose the size of each tuple is χ_t, the size of each item which stores key statistics is χ_k where $\chi_t > \chi_k$, the number of tuples to be stored is $|R|$ and the number of keys is K. Then we obtain SGR as,

$$SGR = \frac{\chi_t * |R|}{\chi_t * |R| + \chi_k * K} \tag{12}$$

We can assume $|R| = c * K$, where c is the average number of tuples each key has ($c \geq 1$). The above equation can be rewritten as,

$$SGR = \frac{\chi_t * c}{\chi_t * c + \chi_k} \tag{13}$$

In real-world data the number of tuples is always far larger than the number of keys. (In the aforementioned on-demand ride-hailing datasets, for example, the value of c in the passenger stream is 14, and that in the taxi tracking stream is more than 10 thousands). From Eq. (13), when c is larger than 10, the value of SGR is larger than 0.9, which means more than 90 percent of memory can be shared to store new tuples when new join instances are added. Therefore, just like BiStream, when we add more join instances to expand the memory capacity of the whole stream join system, FastJoin can almost fully utilize the newly added memory space.

V. Implementation

We implement FastJoin[1] on top of BiStream[16], which is based on Apache Storm [31], the most popular distributed stream system. There are three major concepts about Storm application, including spout, bolt, and topology. Spout is in charge of ingesting data from data source and sending data to downstream to process. Each bolt has its special logic function depending on users' definition. It processes data from upstream.

First, we use the KafkaSpout as data spout of our stream join topology to obtain input data from Kafka cluster. We can adjust the number of instances of KafkaSpout, so that we can adjust the input rate of stream. In practice, the administrator of FastJoin can choose a reasonable configuration of number of KafkaSpout instance to maximize the input rate of stream under certain resource constraints.

The main structure of the FastJoin stream can be divided into two parts, the router part and the joiner part. The router part consists of two components: shuffler and dispatcher. We implement the shuffler bolt and the dispatcher bolt for shuffler and dispatcher respectively. Shufflers are pre-processing units in charge of assigning timestamp to tuples. Dispatchers distribute tuples to some worker instances accroding to distribution strategy. In FastJoin, dispatchers need to maintain a HashMap to store some keys whose tuples have been migrated to a new worker instance. We implement the joiner bolt for joiner part. In order to obtain the statistical information of the whole system such as throughput and laytency, we implement the statistic bolt.

Specially, FastJoin has a new component, monitor component. We implement monitor bolt for monitor component. There are totally two monitor bolts which will be set up in FastJoin. Each monitor bolt is responsible for the statistics of each side of join-biclique.

VI. Evaluation

We implement FastJoin system on top of Apache Storm [31], and evaluate the performance of FastJoin by comparing with BiStream and BiStream-ContRand [16], which are state-of-the-art distributed system for hash stream join. We conduct experiments with large-scale real-world dataset as well as synthetic datasets.

[1] https://github.com/CGCL-codes/FastJoin

A. Methodology

The experiments are conducted on a cluster of 30 nodes. Each node is equipped with an octa-core 2.4GHz Xeon CPU, 64GB RAM, and a 1000Mbps Ethernet interface card.

In the experiment, we use real-world data to examine the skew distribution of keys. Specifically, we use the on-demand ride-hailing dataset provides by DiDi Chuxing GAIA Initiative [1]. The dataset includes two kinds of data, one is the query orders of passengers, the other is the driving track of taxis. All the data are collected in Chengdu, China during November 2016. We use the location of passengers and taxis as key for joining since the order should always be dispatched to the nearest taxi. The passenger order stream contains seven million records, each record is associated with an order id, a timestamp, and the GPS location information. The taxi track stream is a combination of the track records of all the taxis. Each record is collected by the taxi independently and periodically. The taxi track stream contains three billion records, each record is associated with a taxi id, the GPS location information, and a timestamp.

To find an optimal taxi order dispatching is complex [33]. Hence, we only consider a simplified model: each time when a passenger generates an order request, the system will check all the taxis which are nearby or once passed by the position of the passenger. Once a taxi comes to a new position, the system will inform it about all the order requests at this position. Additionally, we divide the datasets into different scales accordingly to the timestamps. The divided data volume is in a range from 10GB to 70GB.

We also generate synthetic datasets with different skewness. In each dataset, one stream has 300 million tuples, and 10 million unique keys. The keys in each stream is either uniformly distributed or following the zipf distribution. The zipf coefficient which determines the degree of skewness is set to either 1.0 or 2.0. Thus, we have nine groups of synthetic datasets in total.

We compare the performance of FastJoin with BiStream and BiStream-ContRand in terms of overall system throughput, average processing latency, and degree of load imbalance among join instances. The system throughput is defined as the rate of final resulting tuples being obtained by join instances, while the processing latency is considered as the average processing time in join instances when tuples first arrive at the instances until the completion. The degree of load imbalance is defined in the previous section. Because distributed stream join is widely used in various kind of real-time analysis applications, high throughput and low latency is essential. Our experimental results will demonstrate that keeping the load balanced among join instances help to improve the overall system performance. In the experiment we also measure scalability which is defined as the degree of improving processing ability when the system scales out.

We use a counter bolt to obtain the final result tuples and calculate the numbers of these tuples every second to measure system throughput. We record the average time spent

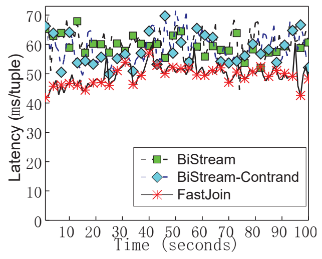


Fig. 3: The real-time system throughput during the processing procedure

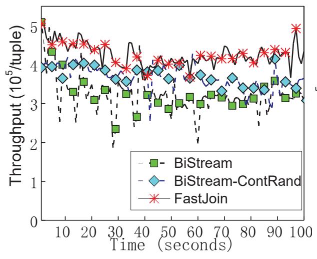


Fig. 4: The real-time processing latency during the processing procedure

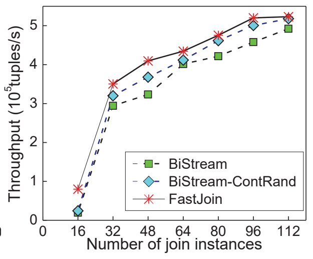


Fig. 5: The average system throughput with different number of instances

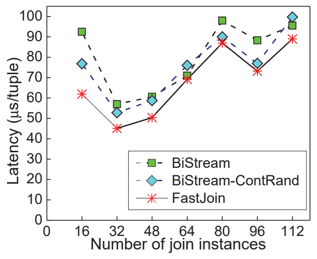


Fig. 6: The processing latency with different number of instances

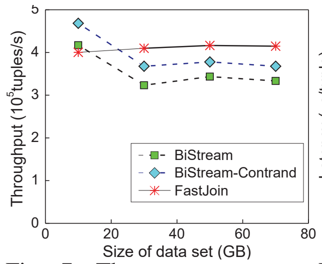


Fig. 7: The average system throughput with different sizes of the dataset

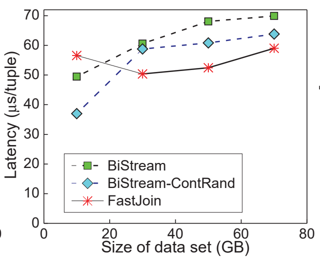


Fig. 8: The processing latency with different sizes of the dataset

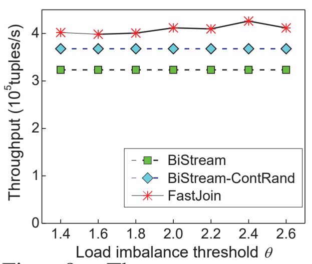


Fig. 9: The average system throughput with different threshold Θ

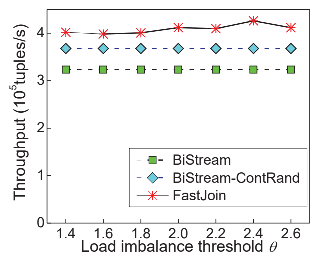


Fig. 10: The processing latency with different threshold Θ

for executing every 10,000 join operations in join instances as processing latency. Finally, we add a monitor bolt to BiStream to monitor the degree of load imbalance. We report the degree of load imbalance every second. Since it is hard to make all the join instances start execution at the same time, the system performance, e.g., the throughput of the first several seconds changes heavily. Thus, we only record the stable statistics after the application runs for around three minutes. To measure scalability we tailor original data into several smaller ones with different volumes and then conduct experiments using different number of join instances for each dataset.

We examine the system throughput and latency by setting different number of join instances, data volumes, load imbalance thresholds, and different skewness of datasets. For real-world DiDi's datasets we set the number of join instances to 48, the load imbalance threshold Θ to 2.2, and the data size to 30GB as default.

B. Results

Firstly, we examine the overall performance of our FastJoin system and that of both BiStream and BiStream-ContRand. We run the taxi order dispatching application using DiDi's dataset for a long time. We record the system throughput and the average processing latency. We use DiDi's dataset for this experiment, with the number of join instances = 48, the size of the dataset = 30GB, the load imbalance threshold Θ = 2.2. The results in Figs. 3 and 4 show that our FastJoin can always achieve a higher system throughput and lower processing latency than BiStream-ContRand which is better than BiStream in most time. The FastJoin system increases the average system throughput of BiStream-ContRand by 16 percent, and that of BiStream by 31.7 percent, while decreasing the average processing latency of BiStream-ContRand by 15.3 percent, and that of BiStream by 17.5 percent.

We further examine the performance of our FastJoin under different conditions. We run each experiment for ten minutes and report the average statistics. Figures 5 and 6 show the average system throughput and the average latency of FastJoin and BiStream when we use different numbers of join instances. We use DiDi's dataset for this experiment, with the size of the dataset = 30GB, the load imbalance threshold Θ = 2.2. The results show that when the number of join instances is as small as 16, the system has rather low throughput and high processing latency. In this case, FastJoin significantly outperforms BiStream-ContRand and BiStream. Specifically, Fastjoin increases the system throughput of BiStream-ContRand by 186 percent and that of BiStream by 258 percent, while decreasing the processing latency by 19.4 percent and 33.1 percent. When the number of instances increases, the performance of the two systems becomes closer to each other, this is because for the same dataset, each instance needs to deal with less keys, then the degree of load imbalance decreases. However, FastJoin still has better performance. We can see from the figure that the latency is increased when the number of instances increases. This may be caused by the increased communication overhead for tuple dispatching and results gathering when the number of instances increases.

Figures 7 and 8 show the average system throughput and the average latency of FastJoin, BiStream-ContRand, and BiStream when we use different scales of datasets. We use DiDi's dataset with the number of join instances = 48, the load imbalance threshold Θ = 2.2. The results show that the scale of dataset does not affect the system performance significantly.

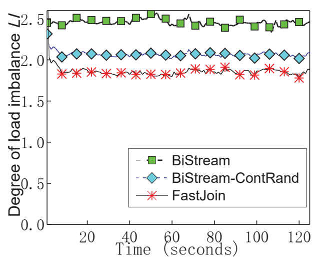


Fig. 11: The real-time degree of load imbalance LI during processing

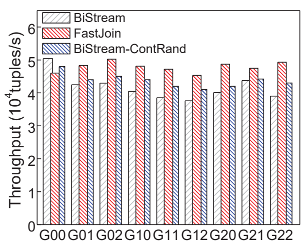


Fig. 12: The system throughput with different skewness of datasets

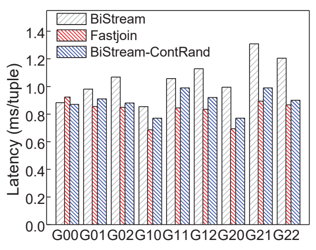


Fig. 13: The processing latency with different skewness of datasets

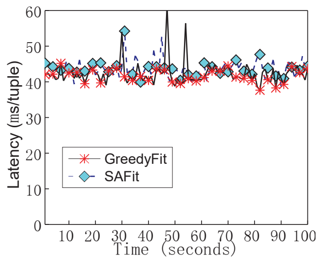


Fig. 14: The processing latency of FastJoin using GreedyFit and SAFit

Our FastJoin does not perform well with a small dataset. This is because the average number of keys stored in an instance is very small, and our key selection algorithm is limited by the solution space and cannot lead to an efficient load balancing. For datasets of large size, as the results shown in Figures 7 and 8, our FastJoin significantly improves the performance of both BiStream-ContRand and BiStream.

Figures 9 and 10 examine the influence of the load imbalance threshold Θ. We use DiDi's dataset for these experiments, with the number of join instances = 48, the size of the dataset = 30GB. The results show that a lower threshold or a higher threshold will decrease the system performance slightly. This is natural since if we set the threshold too large, it helps little for load balancing. If we set the threshold too close to one, it is impossible to migrate keys to make effective load balancing. We can see from Figs. 9 and 10 that FastJoin achieves better performance than both BiStream-ContRand and BiStream.

To further analyze the impact of the load balance threshold and the efficiency of our load balancing strategy, we monitor and record the real-time degree of load imbalance LI for each second. Figure 11 shows the results. We use DiDi's dataset for this experiment, with the number of join instances = 48, the size of the dataset = 30GB, the load imbalance threshold Θ = 2.2. At the very beginning, BiStream-ContRand, BiStream, and FastJoin all suffer from the load imbalance problem. The degree of load imbalance is as high as about 2.5, which means the load of the heaviest instance is twice as heavy as the load of the lightest one. When we set the threshold Θ to 2.2, the migration operation is triggered, and the degree of load imbalance rapidly decreases to 1.9. In contrast, the degree of load imbalance of BiStream-ContRand and BiStream nearly does not change. The results in Figure 11 show that the degree of load imbalance remains less than 2.2 from then on. However, we also find from the experiment that in fact there are several migration operations triggered. This means that our scheme can quickly reduce the degree of load imbalance once it reaches the threshold, and the procedure is less than one second. As a result, our scheme can realize efficient real-time dynamic load balancing.

We use synthetic datasets with different skewness distribution for evaluating the efficiency of our FastJoin system as well. We use nine different group of datasets for the evaluation. Figures 12 and 13 show the average system throughput and the average processing latency. The notation "Gxy" in the x-axis represents one group of the synthetic data, in which the data distribution of the two streams follows zipf distribution, and the zipf coefficient is 'x' and 'y' separately. For example, "G02" means stream R is uniformly distributed, and stream S is skewed with zipf coefficient equal to 2.0. The results show that even when two streams are both uniformly distributed, our FastJoin still outperforms BiStream and BiStream-ContRand. When there is at least one stream with a skewed distribution, our FastJoin performs much better than BiStream and BiStream-ContRand.

Finally, we examine the effectiveness of our GreedyFit algorithm by comparing it with the SAFit algorithm. We measure the average processing latency of FastJoin with these two key selection algorithms. We use DiDi's dataset with the number of join instances = 48, the size of the dataset = 30GB, and the load imbalance threshold Θ = 2.2. The results in Fig. 14 show that the average performance of these two algorithms are nearly the same. This means that GreedyFit algorithm is good enough for dynamic load balancing.

VII. Conclusion

In this paper we propose a new distributed stream join system, FastJoin, to deal with the load imbalance problem in hash stream join. We design and implement an efficient key selection algorithm, GreedyFit, and the corresponding load migration scheme for solving the problem. The proposed dynamic load balancing procedure is lightweight, very fast and incurs low overhead. The experimental results show that our FastJoin outperforms the state-of-the-art distributed system for hash stream join in terms of both system throughput and processing latency.

VIII. Acknowledgements

This research is supported in part by the National Key Research and Development Program of China under grant No.2016QY02D0302, NSFC under grant No. 6143301, and Research Fund of Guangdong Province under grant No.2015B010131001.

References

[1] Didi Chuxing GAIA Initiative, https://gaia.didichuxing.com, 2018.

[2] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh,

and S. Venkataraman, “Photon: fault-tolerant and scalable joining of continuous data streams,” in *Proceedings of SIGMOD*, 2013.

[3] S. Blanas, Y. Li, and J. M. Patel, “Design and evaluation of main memory hash join algorithms for multi-core CPUs,” in *Proceedings of SIGMOD*, 2011.

[4] D. Buono, T. D. Matteis, and G. Mencagli, “A high-throughput and low-latency parallelization of window-based stream joins on multicores,” in *Proceedings of ISPA*, 2014.

[5] H. Chen, F. Zhang, and H. Jin, “Popularity-aware differentiated distributed stream processing on skewed streams,” in *Proceedings of ICNP*, 2017.

[6] A. Drexl, “A simulated annealing approach to the multiconstraint zero-one knapsack problem,” *Computing*, vol. 40, no. 1, pp. 1–8, 1988.

[7] M. E. Dyer, N. Kayal, and J. Walker, “A branch and bound algorithm for solving the multiple-choice knapsack problem,” *Journal of Computational and Applied Mathematics*, vol. 11, no. 2, pp. 231–249, 1984.

[8] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch, “Scalable and adaptive online joins,” *Proceedings of the VLDB Endowment*, vol. 7, no. 6, 2014.

[9] J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou, and J. Zhu, “Parallel stream processing against workload skewness and variance,” in *Proceedings of HPDC*, 2017.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[11] B. Gedik, “Partitioning functions for stateful data parallelism in stream processing,” *The VLDB Journal*, vol. 23, no. 4, pp. 517–539, 2014.

[12] B. Heintz, A. Chandra, and R. K. Sitaraman, “Trading timeliness and accuracy in geo-distributed streaming analytics,” in *Proceedings of SOCC*, 2016.

[13] L. Hu, K. Schwan, H. Amur, and X. Chen, “ELF: efficient lightweight fast stream processing at scale,” in *Proceedings of USENIX ATC*, 2014.

[14] G. Jacques-Silva, R. Lei, L. Cheng, G. J. Chen, K. Ching, T. Hu, Y. Mei, K. Wilfong, R. Shetty, S. Yilmaz, A. Banerjee, B. Heintz, S. Iyer, and A. Jaiswal, “Providing streaming joins as a service at facebook,” *Proceedings of the VLDB Endowment*, vol. 11, no. 12, 2018.

[15] C. Lin, J. Zhan, H. Chen, J. Tan, and H. Jin, “Ares: A high performance and fault-tolerant distributed stream processing system,” in *Proceedings of ICNP*, 2018.

[16] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu, “Scalable distributed stream join processing,” in *Proceedings of SIGMOD*, 2015.

[17] O. Marcu, R. Tudoran, B. Nicolae, A. Costan, G. Antoniu, and M. S. Pérez-Hernández, “Exploring shared state in key-value store for window-based multi-pattern streaming analytics,” in *Proceedings of CCGrid*, 2017.

[18] T. D. Matteis and G. Mencagli, “Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing,” in *Proceedings of PPoPP*, 2016.

[19] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, and A. H. Teller, “Equation of state calculations by fast computing machines,” *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1–7, 1953.

[20] U. I. Minhas, M. Russell, S. Kaloutsakis, P. Barber, R. F. Woods, G. Georgakoudis, C. Gillan, D. S. Nikolopoulos, and A. Bilas, “Nanostreams: A microserver architecture for real-time analytics on fast data streams,” *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 396–409, 2018.

[21] J. Myung, J. Shim, J. Yeon, and S. Lee, “Handling data skew in join algorithms using mapreduce,” *Expert Systems with Applications*, vol. 51, pp. 286–299, 2016.

[22] M. Najafi, M. Sadoghi, and H. Jacobsen, “Splitjoin: A scalable, low-latency stream join architecture with adjustable ordering precision,” in *Proceedings of USENIX ATC*, 2016.

[23] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, “The power of both choices: Practical load balancing for distributed stream processing engines,” in *Proceedings of ICDE*, 2015.

[24] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini, “When two choices are not enough: Balancing at scale in distributed stream processing,” in *Proceedings of ICDE*, 2016.

[25] A. Okcan and M. Riedewald, “Processing theta-joins using mapreduce,” in *Proceedings of SIGMOD*, 2011.

[26] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann, “Flow-join: Adaptive skew handling for distributed joins over high-speed networks,” in *Proceedings of ICDE*, 2016.

[27] P. Roy, J. Teubner, and R. Gemulla, “Low-latency handshake join,” *Proceedings of the VLDB Endowment*, vol. 7, no. 9, 2014.

[28] A. Shukla and Y. Simmhan, “Model-driven scheduling for distributed stream processing systems,” *Journal of Parallel and Distributed Computing*, vol. 117, pp. 98–114, 2018.

[29] H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto, “Streaming, low-latency communication in on-line trading systems,” in *Proceedings of IPDPSW*, 2010.

[30] J. Teubner and R. Müller, “How soccer players would do stream joins,” in *Proceedings of SIGMOD*, 2011.

[31] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, “Storm@twitter,” in *Proceedings of SIGMOD*, 2014.

[32] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, “Handling data skew in parallel joins in shared-nothing systems,” in *Proceedings of SIGMOD*, 2008.

[33] L. Zhang, T. Hu, Y. Min, G. Wu, J. Zhang, P. Feng, P. Gong, and J. Ye, “A taxi order dispatch model based on combinatorial optimization,” in *Proceedings of KDD*, 2017.