# TriJoin: A Time-Efficient and Scalable Three-Way Distributed Stream Join System

*Shuiying Yu, Yinting Zheng, Fan Zhang, Hanhua Chen*, Hai Jin*

*School of Computer Science and Technology, Huazhong University of Science and Technology, China*
*{shuiying, zhengyinting, zhangf, chen, hjin}@hust.edu.cn*

## Abstract

Stream join is one of the most fundamental operations in data stream processing applications. Existing distributed stream join systems can support efficient two-way join, which is a join operation between two streams. Based the two-way join, implementing a three-way join require to be split into double two-way joins, where the second two-way join needs to wait for the join result transmitted from the first two-way join. We show through experiments that such a design raises prohibitively high processing latency. To solve this problem, we propose TriJoin, a time-efficient three-way distributed stream join system. We design a symmetric wait-free structure by symmetrically partitioning tuples and reused join. TriJoin utilizes reused join to join each new tuple with the intermediate result of the other two streams and stored tuples locally. For a new tuple, TriJoin only joins it with the intermediate result to generate the final result without waiting, greatly reducing the processing latency. In TriJoin, we design two partitioning and storage schemes according to two different forms of three-way stream join. We implement TriJoin and conduct comprehensive experiments to evaluate the performance using real-world traces. Results show that TriJoin significantly reduces the processing latency by up to 68%, compared to existing designs.

**Keywords:** Distributed stream processing, Stream join, Three-way stream join

## 1 Introduction

Recently, an increasing number of applications process streaming data in real-time, such as on-demand ride-hailing, live stream e-commerce, and live video recommendation. Stream join is a fundamental operation in these applications. For example, an on-demand ride-hailing platform [1] dispatches taxi orders in real-time, which entails continuously joining the order stream of passengers and the driving steam of taxis. A time-efficient and high throughput stream join technique is crucial for meeting the real-time requirements of these applications.

Stream join applications commonly involve three-way join (i.e., joining three streams), which can be used to join multiple streams. For example, a live stream e-commerce platform [2] (such as Alibaba's Taobao Live platform) has a tripartite relationship among streamers (i.e., the anchors employed by platform), users, and products. To enhance product recommendations, the platform obtains the influence of a streamer through the quantity of products purchased by users in real-time. This task can be performed by joining the user stream, the product (promoted by streamer) stream, and the product purchase stream. As another example, in a live streaming video recommendation system [3], users who click on the same live streaming video probably like similar live streaming videos, and rewarding a video means that the user likes the live streaming video. Since a live streaming video is time-limited, to accurately recommend videos to users in real-time, the system can discover which live streaming video has been rewarded by users reviewing the current live streaming video. This task can be performed by joining the click stream, review stream, and reward stream.

Since a stream join application usually processes continuous data in real-time, it faces more challenges than traditional database join operations. The basic requirements for performing effective stream join include three aspects [4-5]: 1) low latency and high throughput, which are critical for time-efficient big data analysis systems; 2) memory efficiency and scalability. Since real applications require low processing latency, the system should store and join the data in memory. To deal with large-scale streaming data, a system requires memory efficiency and scalability. When the system repeatedly stores data, this will cause redundant memory, and poor scalability; 3) completeness, that is the final result cannot be repeated or omitted. More concretely, a new tuple should be joined with all the tuples or all the join results of other streams exactly once.

To meet these three requirements, many systems have been proposed based on parallel and distributed designs [6-7]. Among existing designs, BiStream [8] is one of the most popular stream join system. To avoid replicating data, BiStream adopts the join-biclique model, which stores tuples only once. The join-biclique organizes all the processing units as a complete bipartite graph. When a tuple arrives, BiStream stores the tuple in one of the processing units to which it belongs, meanwhile broadcasts it to all the processing units on the other side to join. BiStream can join each new tuple with all the tuples of the other stream once to ensure completeness. Each new tuple that arrives at a processing unit performs a join operation with the stored tuples to generate the final result, ensuring the low processing latency. However, BiStream is a two-way join system (i.e., joining two streams), and cannot

*Corresponding Author: Hanhua Chen; E-mail: chen@hust.edu.cn
DOI: 10.53106/160792642023032402024

support three-way join.

A naive design of three-way stream join can employ a tree architecture [9-10] to split the process into two two-way joins. The tree architecture stores the intermediate result to reduce the join operations, and can be easily combined with the BiStream design. There is only one way to deploy BiStream+Tree (i.e., three-way stream join by BiStream and the tree architecture). BiStream+Tree first joins two of the streams to form a stream of intermediate result, and then joins the intermediate result stream and the third stream. Suppose the three streams are *R*, *S*, and *T*. Figure 1 illustrates three-way join by tree architecture. BiStream+Tree stores the tuples of three streams and intermediate result into four groups of processing units *R*, *S*, *T*, and *I_RS*. BiStream+Tree first joins *R* and *S* to generate the intermediate result stream *I_RS*, and then joins *I_RS* with *T*, which is the second stream join.

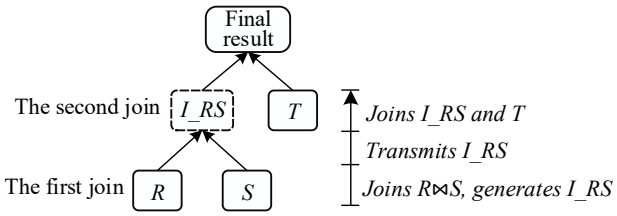


**Figure 1.** Three-way join by tree architecture

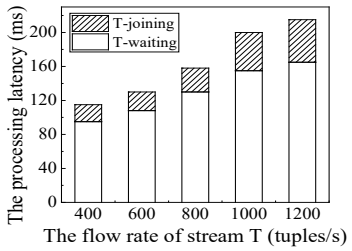


**Figure 2.** The latency break down of a tuple of *T*

BiStream+Tree cannot satisfy the low latency requirement of stream join. The processing latency of a tuple is the time from the arrival of the tuple to the generation of the final result. For BiStream+Tree, the processing latency of a tuple of stream *R* (or *S*) is the sum of the time to join the tuple with the tuples of *S* (or *R*), the transmission time of the intermediate result, and the time to join the intermediate result with the tuples of *T*. When a tuple of *T* arrives at processing unit *I_RS*, the tuples of *R* and *S*, which arrive at the same time as the tuple of *T*, are joining with the tuples of *S* and *R*. To ensure completeness, the tuple of *T* waits for the processing unit *I_RS* to receive all the intermediate results joining the tuples of *R* and *S* before the tuple of *T* arrives. We examine the proportion of waiting time in the processing latency of BiStream+Tree using GPS traces of the DiDi Chuxing GAIA Initiative [11]. In the experiment, we deploy BiStream+Tree on a cluster of 16 nodes, where each node is equipped with two 8-core CPUs. We break down the latency of the tuple from *T* into waiting time and joining time, and show the result in Figure 2. The result shows that the waiting time contributes to most of the latency. If the system can alleviate the waiting time, it can effectively reduce the latency.

The essential reason for waiting is that the system transmits the intermediate result between different processing units. To ensure the completeness of the join result, the second join must wait for a period of time until the intermediate result arrive after transmission because the tuples in the streams are out-of-order. However, the existing structures cannot avoid this transmission of the intermediate result. Therefore, it is challenging to alleviate the waiting time to generate the final result after the tuple arrives.

To solve the problem, in this work, we propose a novel transmit-free and wait-free structure called TriJoin, which avoids transmiting a large number of intermediate results to alleviate the waiting time. Instead, TriJoin utilizes symmetrically partitioning tuples and reused join. Specifically, TriJoin divides all the processing units into three groups according to the three streams. It combines the three groups into three different pairs, utilizes symmetrically partitioning to partition tuples into this three groups, and stores the tuples only in a processing unit. When a new tuple arrives in a processing unit to which it does not belong, the system utilizes reuse join to join the intermediate result and the stored tuple. TriJoin joins the new tuple with the stored tuple to generate the intermediate result and store it locally without transmission. TriJoin joins the new tuple with the intermediate result without waiting to generate the final result.

In addition, since TriJoin joins each new tuple with the corresponding intermediate result. In practice, a tuple will find multiple tuples of another stream that meet the join condition. The existing storage structure for intermediate result is tuple pair [12], which raises the repeated storage of tuples and join operations on the intermediate result. To reduce the cost of repeated storage and join, we design a TuplePacking structure for the intermediate result. TuplePacking consists of a tuple from one stream and many tuples from another stream that meet the join condition.

We implement TriJoin and conduct comprehensive experiments with real-world traces to evaluate the design. The results show that compared to BiStream+Tree, TriJoin significantly reduces the processing latency by 68%.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 describes the design of TriJoin. Section 4 analyzes the processing latency, completeness, and scalability of TriJoin. Section 5 presents the implementation of TriJoin. Section 6 evaluates this design. Section 7 concludes the paper.

## 2 Related Work

In this section, we review the related work in both two-way and multi-way stream joins.

### 2.1 Two-Way Stream Join

Based on the parallel stream join system, many stream join algorithms have been proposed. Existing designs [13-14] implement multi-core parallel processing stream join using the handshake join model. However, the handshake join model has the problems such as high communication cost

between processing cores, and complex synchronization and coordination.

To avoid frequent communication among cores in the low-latency handshake join model, Najafi et al. introduce SplitJoin [15], which abandons the linear sequential flow of tuples between different processing cores. SplitJoin adopts the method of simultaneously broadcasting tuples from two streams to all the processing cores in the same sequence, and then performs join operation. Some designs [16-17] utilize the *field programmable gate array* (FPGA) or multicore CPU to speed up the join operation. However, the parallel systems are difficult for a parallel stream join system to scale out for rapidly increasing workloads in practice.

Distributed stream join systems benefit from the high scalability of distributed clusters, and can potentially support the stream join operations of all historical data. The Join-Matrix [18-19] inputs the two streams as the rows and columns of the matrix. Each matrix cell represents a potential join result. When the system scales out/in, it adds/deletes a fixed number of processing units in a row or column. This model limits the scalability of the system.

To avoid storing data repeatedly, Lin et al. propose a two-way stream join system BiStream [8]. BiStream employs the join-biclique model, which organizes all the processing units into a bipartite graph, including two groups of storage corresponding to the two streams. Some studies [4-5] improve BiStream to dynamically deal with the load imbalance caused by a skewed distribution of real-world data. Yuan et al propose a novel ordered propagation model to eliminate abnormal results [20]. However, the two-way stream join cannot directly process three-way join.

### 2.2 Multi-Way Stream Join

It is a challenge to deal with multiple streams with inputs arriving at highly variable and unpredictable rates. Gomes et al. present a dynamic programming algorithm, OptDP [10], which produces the optimal join tree architecture. The optimal join tree architecture maximizes the throughput for sliding window-based multi-join queries over continuous data streams.

The expensive join conditions of multi-way join lead to a great challenge for real-time stream processing. Wang et al. introduced a multi-way join distribution scheme named *pipelined state partitioning* (PSP) [21], which transforms a macro join operator into a series of smaller sub-operators through time-slicing of the join states. However, PSP adopts a virtual computation ring, and a new tuple needs to be transmitted to several nodes. This results in high latency. Moreover, a large number of join operations are repeated because there is no intermediate result stored.

To trade off transmission consumption and the limited memory requirements for storing the intermediate results, CLASH [22-23] utilizes the features of the flat (i.e., ring) and tree architectures. However, CLASH still suffers from high latency due to the ring and tree architectures.

All in all, although distributed stream join systems have better scalability than parallel systems, the existing systems cannot meet the low latency requirement of three-way stream join. The two-way stream join method cannot directly realize three-way stream join while multi-way stream join. Utilizing a tree or ring architecture to realize three-way join will raises high latency due to the intermediate result transmission, the final result waiting for intermediate result, or the repeated joining. Therefore, our goal is to design a low latency stream join system.

## 3 TriJoin

In this section, we first briefly describe the system overview of TriJoin. Then, we present the two partitioning and storage schemes according to two forms of three-way join. Finally, we present the TuplePacking structure for storing the intermediate result.

### 3.1 Overview

TriJoin divides the processing units into three symmetric groups according to three streams, and utilizes symmetrically partitioning to partition tuples. The dispatcher partitions tuple in round-robin and broadcast manners. TriJoin stores the corresponding tuple and the intermediate result only once in the processing unit to ensure the scalability.

Figure 3 shows the overall architecture of TriJoin, which includes three components: routing, join, and postprocessing components.

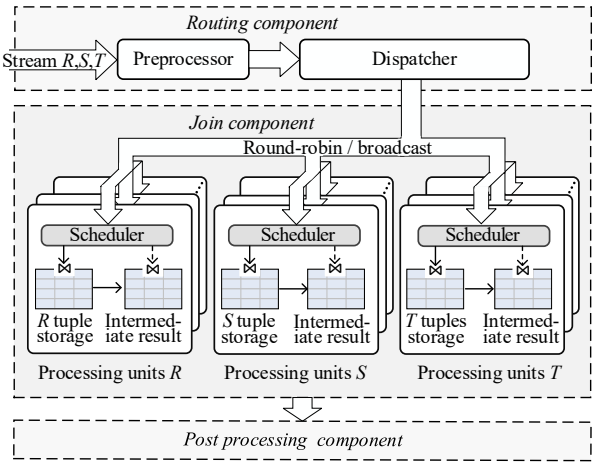


**Figure 3.** The system architecture of TriJoin

In routing component, the preprocessor is responsible for preprocessing the tuples. After preprocessing, according to the partitioning scheme, the dispatcher partitions the received tuples to the corresponding processing units in the join component. The join component consists of many processing units, which store and join the received tuples. When the processing unit receives a tuple, the scheduler judges whether to store the tuple or join it with the stored tuple and intermediate result. The processing unit stores the intermediate results generated locally to reduce the processing latency. The processing unit sends the final results to the postprocessing component. The postprocessing component outputs and analyzes the final result.

The partitioning and storage of tuple and intermediate result are the core of TriJoin. The partitioning and storage schemes are related to the form of the three-way join. We clarify the possible forms of three-way join as follows. The

three streams can only make up two join graphs, which are cyclic join and chain join [24]. We assume that the three streams are $R$, $S$, and $T$, and use the notation $\bowtie$ to represent theta-join. Therefore, we set the two forms of three-way join: $R \bowtie S$ AND $S \bowtie T$ AND $T \bowtie R$, and $R \bowtie S$ AND $S \bowtie T$, which are the cyclic and chain joins, respectively. Table 1 lists the notations in our design.

**Table 1.** Notations

| Notation | Description |
|---|---|
| $R$, $S$, $T$ | Three data streams for joining |
| $r$, $s$, $t$ | Tuples belonging to $R$, $S$, and $T$, respectively |
| R $\bowtie$ S | The theta-join of $R$ and $S$ |
| $R_i$, $S_i$, $T_i$ | The $i$-th processing unit of $R$, $S$, and $T$ |
| $K_{Ri}$, $K_{Si}$, $K_{Ti}$ | The sets of all the tuples stored in processing unit $R_i$, $S_i$, and $T_i$ |
| $\|K_{Ri}\|$,$\|K_{Si}\|$,$\|K_{Ti}\|$ | The numbers of tuples in set $K_{Ri}$, $K_{Si}$, and $K_{Ti}$ |
| $I\_sK_{Ri}$ | The intermediate result of $s \bowtie K_{Ri}$ |
| $\|I\_sK_{Ri}\|$ | The number of intermediate result $I\_sK_{Ri}$ |
| $K_R$ | The set of all the tuples stored in the system for stream $R$ |
| $I\_sK_R$ | The intermediate result of $s \bowtie K_R$ |
| $\|I\_sK_R\|$ | The number of intermediate result $I\_sK_R$ |

### 3.2 Partitioning and Storage Schemes

It is nontrivial to avoid the tuple waiting for intermediate result. Our schemes enable as many intermediate and final results as possible to be generated in one processing unit. For cyclic join, the scheme ensures that all the tuples perform only one process to generate the final results, and no tuples have a waiting process.

In the two schemes, the three groups of processing units are $R$, $S$, and $T$. We use the notations $n_r$, $n_s$, and $n_t$ to represent the number of processing units $R$, $S$, and $T$. We use $R_i$ to denote the $i$-th processing unit $R$ for $i \in \{1...n_r\}$, $S_i$ to denote the $i$-th processing unit $S$ for $i \in \{1...n_s\}$, and $T_i$ to represent the $i$-th processing unit $T$ for $i \in \{1...n_t\}$. We denote the set of all the tuples stored in processing units $R_i$, $S_i$ and $T_i$ using $K_{Ri}$, $K_{Si}$, and $K_{Ti}$. We denote the tuple of $R$, $S$, and $T$ using $r$, $s$, and $t$. We use $I\_sK_{Ti}$ to denote the intermediate result of $s \bowtie K_{Ti}$. The two different partitioning schemes are described as follows.

#### 3.2.1 The symmetric partitioning scheme of the cyclic join

For the cyclic join, in order to join a new tuple with the intermediate result without transmission, we design a symmetric partitioning scheme, as shown in Figure 4. The symmetric partitioning scheme combines the three groups of processing units into three different pairs of processing units. While storing the tuple only once, these three pairs of processing units form three join-biclique models by sharing the stored tuples. The three join-biclique models can generate all the intermediate results and store them locally without transmission. Therefore, when a tuple arrives, TriJoin joins the tuple with the intermediate result without waiting and generates the final result. When a new tuple arrives at the dispatcher, the dispatcher partitions the tuple into one of the processing units of the group to which it belongs in a round-robin manner. Meanwhile, it broadcasts the new tuple to all the processing units of the other two groups. Each processing unit joins the broadcasted tuple with the intermediate result to generate the final result. Then each processing unit joins the broadcasted tuple with the locally stored tuples to generate the intermediate result, and stores the intermediate result locally. Subsequently, the processing unit discards the broadcasted tuple.

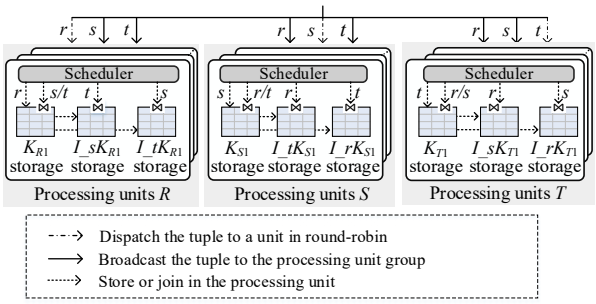


**Figure 4.** The symmetric partitioning scheme of the cyclic join

We describe the operations of the symmetric partitioning scheme with examples. Whenever a new tuple $r$ (or $s$, $t$) arrives, the dispatcher partitions the tuple in a round-robin manner to one of the processing units $R$ (or $S$, $T$) for storage, and broadcasts the tuple $r$ (or $s$, $t$) to the other two processing units $S$ and $T$ (or $R$ and $T$, $R$ and $S$). Each processing unit $S_i$ and $T_i$ joins $r$ with the intermediate result $I\_sK_{Si}$ and $I\_tK_{Ti}$ stored locally to generate the final result and outputs. Then each processing unit $S_i$ and $T_i$ joins $r$ with the tuples $K_{Si}$ and $K_{Ti}$ to generate intermediate result $I\_rK_{Si}$ and $I\_rK_{Ti}$, and stores the intermediate result locally. Similarly, each processing unit $R_i$ and $T_i$ (or $R_i$ and $S_i$) joins the broadcasted $s$ (or $t$) with intermediate result $I\_tK_{Ri}$ and $I\_rK_{Ti}$ (or $I\_sK_{Ri}$ and $I\_rK_{Si}$) to generate the final result. Each processing unit $R_i$ and $T_i$ (or $R_i$ and $S_i$) joins the broadcasted $s$ (or $t$) with the tuples stored locally to generate intermediate result $I\_sK_{Ri}$ and $I\_sK_{Ti}$ (or $I\_tK_{Ri}$ and $I\_tK_{Si}$). Then the processing unit discards the broadcasted tuple.

The symmetric partitioning scheme can guarantee the three requirements. Since any pair of processing unit groups conforms to the join-biclique model, this scheme generates the complete intermediate result. When a tuple arrives, it can join with all the complete intermediate result generated before it arrived. The symmetric partitioning scheme ensures completeness. When a tuple reaches the processing unit, the system can immediately join the tuple with the corresponding intermediate result, and perform only one process to generate the final result. The symmetric partitioning scheme achieves low latency in generating the final result of each tuple. TriJoin stores each tuple and intermediate result only once, ensureing the scalability of the system.

#### 3.2.2 The semi-symmetric partitioning scheme of the chain join

For the chain join, to obtain low latency, the partition-ing and storage scheme faces a greater challenge than the cyclic join, since the chain join lacks the join condition $R \bowtie T$. If TriJoin partitions tuples according to the symmetric partition-

ing scheme, the system will omit many final results, which cannot guarantee the completeness of the join result. For example, when $t$ arrives, the processing unit $R_i$ does not generate an intermediate result. Instead, it joins $t$ and $I\_sK_{Ri}$ and then discards $t$. Therefore, when $s$ arrives, the processing unit $R_i$ does not generate the final result. The processing unit $R_i$ omits the final result of joining the new tuple $s$ with $K_{Ri}$ and the tuples of $T$. The processing units $T$ perform similarly.

It is a challenge to solve the problem that processing unit $R_i$ omits the final result of the new tuple $s$. We assume that processing unit $R_i$ stores all the tuples $K_T$ (denoted as the set of all the tuples stored in processing unit $T$) to ensure the completeness of join result. The processing unit $R_i$ allows $s$ to join with $K_{Ri}$ and $K_T$ when $s$ arrives. However, storing $K_T$ in one processing unit will result in an unacceptable memory cost.

To eliminate the cost of storing all the tuples $K_T$ in the processing units $R$ and ensure the completeness of join results, we design a semi-symmetric partitioning scheme (shown in Figure 5). When $s$ arrives, each processing unit $R_i$ generates the intermediate result $I\_sK_{Ri}$, stores it locally. Meanwhile, it broadcasts $I\_sK_{Ri}$ to all the processing units $T$. Each processing unit $T_i$ joins $I\_sK_{Ri}$ with the tuples $K_{Ti}$ to generate the final result, and then dicards $I\_sK_{Ri}$. This avoids storing $K_T$ in one processing unit. In the same way, if the processing unit $T_i$ generates $I\_sK_{Ti}$, it broadcasts $I\_sK_{Ti}$ to all the processing units $R$. Then, each processing unit $R_i$ joins $I\_sK_{Ti}$ with $K_{Ri}$ to generate the final result.

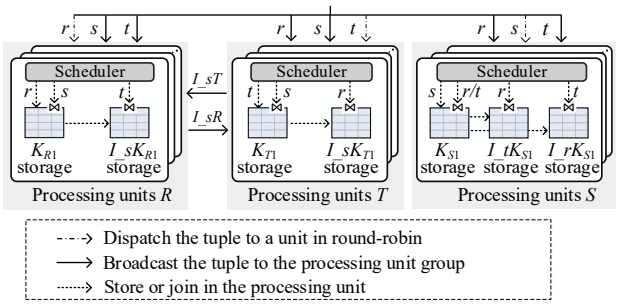


**Figure 5.** The semi-symmetric partitioning scheme of the chain join

The semi-symmetric partitioning scheme joins each tuple without waiting. The processing latencies of the tuples $r$ and $t$ are the time to generate the final result after the tuple arrives. When $s$ arrives, each processing unit $R_i$ joins $s$ and $K_{Ri}$ to generate $I\_sK_{Ri}$, and then sends $I\_sK_{Ri}$ to the processing unit $T$. When the processing unit $T_i$ receives $I\_sK_{Ri}$, it has received the tuple $t$ that arrived at the join component before $s$. Each processing unit $T_i$ can immediately join $I\_sK_{Ri}$ with $K_{Ti}$. The semi-symmetric partitioning scheme processes $s$ without waiting. The semi-symmetric partitioning scheme achieves a low processing latency for each tuple.

### 3.3 Storage of Intermediate Result

Existing stream join systems use the form of a tuple pair (i.e., the structure of binding two tuples) to store the join results of two streams [11-12]. This pair is composed of two tuples that satisfy the join condition. According to the actual requirement, the fields of the tuple include "relation", "timestamp", "seq", "$key_a$", "$key_b$", and "value". The field "relation" indicates the stream to which the tuple belongs. The field "timestamp" denotes the timestamp when the tuple reaches the dispatcher. The fields "$key_a$" and "$key_b$" are join attributes. The field "value" is the specific content of the tuple. As shown in Figure 6, the tuple pair structure of the intermediate result consisting of $tuple_1$ and $tuple_2$ that meet the join condition. The fields of tuple pair include "key", "timestamp", $tuple_1$, and $tuple_2$. The field "key" is the same as the field "$key_a$" or "$key_b$" of $tuple_1$ (denoted as "$key_{1a}$" or "$key_{1b}$"), which has a join condition with the new tuple from the third stream. The field "timestamp" is the maximum value among the timestamps of $tuple_1$ and $tuple_2$. The "relation" fields of the two tuples are different.

| header | $key_{1a}$/$key_{1b}$ | | timestamp | | | |
|---|---|---|---|---|---|---|
| $tuple_1$ | $relation_1$ | $timestamp_1$ | $seq_1$ | $key_{1a}$ | $key_{1b}$ | $value_1$ |
| $tuple_2$ | $relation_2$ | $timestamp_2$ | $seq_1$ | $key_{2a}$ | $key_{2b}$ | $value_2$ |

**Figure 6.** The tuple pair structure for storing intermediate Result

Since a processing unit joins a new broadcasted tuple with all the tuples stored locally, it may generate several intermediate results related to this new tuple. For storing the intermediate result, the processing unit will store the new tuple continuously and repeatedly with several tuple pair structures. Moreover, when another new tuple of the third stream arrives, the processing unit needs to perform several join operations on the new tuple and the repeated stored tuple. Regardless of whether the symmetric or semi-symmetric partitioning scheme, TriJoin needs to store the intermediate results and perform multiple join operations with the intermediate results. Therefore, the tuple pair structure leads to a large amount of unnecessary computation and memory costs for intermediate results.

To address this issue, we design a TuplePacking structure, which is a readable and writable packing structure, to store the intermediate results. TuplePacking is in a one-to-many form, which consists of a tuple of one stream (named a hub tuple) and several tuples of another stream. In the structure, the hub tuple and any tuple of another stream meet the join condition. The stream to which the hub tuple belongs has a join condition with the third stream. As shown in Figure 7, the TuplePacking structure of the intermediate result consisting of one join tuple and $k$ stored tuples that meet the join condition. The fields of TuplePacking include "key", "timestamp", and $k$ tuples. The hub tuple is $tuple_1$. The "key" field is the fields "$key_{1a}$" or "$key_{1b}$". The "timestamp" field is the maximum value among the timestamps of the $k$ tuples, which is the same as the timestamp of $tuple_1$. The "relation" field of $tuple_1$ is different from the other tuples, and the "relation" fields of all the tuples except $tuple_1$ are the same. In the following, we describe the storage and join operations for intermediate result in the two different partitioning schemes.

| | | | | | | |
|---|---|---|---|---|---|---|
| header | $key_{1a}/key_{1b}$ | | timestamp | | | |
| hub tuple ($tuple_1$) | $relation_1$ | $timestamp_1$ | $seq_1$ | $key_{1a}$ | $key_{1b}$ | $value_1$ |
| $tuple_2$ | $relation_2$ | $timestamp_2$ | $seq_2$ | $key_{2a}$ | $key_{2b}$ | $value_2$ |
| $tuple_3$ | $relation_2$ | $timestamp_3$ | $seq_3$ | $key_{3a}$ | $key_{3b}$ | $value_3$ |
| ⋮ | ⋮ | | | | | |
| $tuple_k$ | $relation_2$ | $timestamp_k$ | $seq_k$ | $key_{ka}$ | $key_{kb}$ | $value_k$ |

**Figure 7.** The TuplePacking structure for storing intermediate Result

For the symmetric partitioning scheme, the TuplePacking structure reduces the memory cost and the number of joins between the new tuple and the intermediate result. We take the processing unit $R$ as an example. If TriJoin adopts a tuple pair to store the intermediate result, the processing unit $R_i$ joins a new tuple $s$ and the tuples $T_{Ri}$. Then the processing unit $R_i$ generates the intermediate results $\{(s, r_1), (s, r_2), (s, r_3), (s, r_4)\}$, which are four tuple pair structures. The processing unit $R_i$ needs to store $s$ four times. When a tuple $t$ arrives, processing unit $R_i$ joins the "key" field of $t$ and these intermediate results to generate the final result. The join operation cost is four. If TriJoin stores the intermediate result $I\_sK_{Ri}$ with the TuplePacking structure, the intermediate result is composed of tuples $\{(s, r_1, r_2, r_3, r_4)\}$, which has one "key" field to be joined. When the new tuple $t$ arrives, the processing unit needs only one join comparison of the "key" fields of $t$ and TuplePacking $\{(s, r_1, r_2, r_3, r_4)\}$. For TuplePacking as the storage structure, the join operation cost is one.

For the semi-symmetric partitioning scheme, the TuplePacking structure not only reduces the memory cost but also reduces the communication cost. If TriJoin adopts tuple pair to store the intermediate result, the processing unit $R_i$ joins a new tuple $s$ and the tuples $K_{Ri}$, and generates the intermediate results $\{(s, r_1), (s, r_2), (s, r_3)\}$. The processing unit $R_i$ needs to store three tuple pair structures, and $s$ is stored three times. At the same time, the processing unit $R_i$ broadcasts three intermediate results to all the processing units $T$. The communication cost is $3 \times n_t$. In each processing unit $T_i$, the join operation cost for joining the "key" fields of the intermediate results and $K_{Ti}$ is $3 \times |K_{Ti}|$. If TriJoin adopts the TuplePacking structure, the TuplePacking $I\_sK_{Ri}$ is composed of tuples $\{(s, r_1, r_2, r_3)\}$, which has a "key" field. The system needs to store $s$ once; the communication cost is $n_t$; and the join operation cost is $|K_{Ti}|$. If TriJoin adopts the TuplePacking structure, the memory cost, the computation cost for the join operation, and the communication cost are all lower than those of the tuple pair structure.

# 4 Analysis of Processing Latency

The processing latency of a tuple is from the time when the tuple reaches the dispatcher to the time when the final result is generated. Tuples of BiStream+Tree and TriJoin require a very short queue waiting time in the dispatcher and the processing unit. Therefore, we use the time of the join operation and the network transmission as the processing latency. Since the time to send each tuple from the dispatcher to the processing unit is the same in the three schemes, we ignore this time in this section.

In the symmetric partitioning scheme, TriJoin adopts the TuplePacking structure to store the intermediate result. We denote the intermediate result $I\_K_SK_{Ti}$ stored in TuplePacking as $Io\_K_SK_{Ti}$. The processing latency of the tuple $r$ is that needed to join $r$ with $Io\_K_SK_{Ti}$ in the processing unit $T_i$, or to join $r$ with $Io\_K_TK_{Si}$ in the processing unit $S_i$. The processing latency of tuple $r$ is computed by Equation (1), where $\tau$ represents the processing time of a join comparison.

$$\begin{aligned} L_{sym-r} &= |Io\_K_SK_{Ti}| \times \tau \ (or\ |Io\_K_TK_{Si}| \times \tau) \\ L_{sym-s} &= |Io\_K_TK_{Ri}| \times \tau \ (or\ |Io\_K_RK_{Ti}| \times \tau) \\ L_{sym-t} &= |Io\_K_RK_{Si}| \times \tau \ (or\ |Io\_K_SK_{Ri}| \times \tau). \end{aligned} \quad \textbf{(1)}$$

The number of new tuple join with the intermediate result is $|Io\_K_SK_{Ti}|$ or $|Io\_K_TK_{Si}|$, which is the number of tuples less than $K_S$ or $K_T$. The process of generating the final result of each tuple is only join operation, and has no transmission.

In the semi-symmetric partitioning scheme, the processing latencies of tuples $r$ and $t$ are the same as in the symmetric partitioning scheme. The latency of tuple $s$ is computed by Eq. (2), where $T_{net}$ denotes the transmission time of the intermediate result.

$$L_{semi-s} = (|K_{Ri}| + |K_{Ti}|) \times \tau + T_{net}. \quad \textbf{(2)}$$

From the Equation (2), there is no waiting in the processing latency of $s$.

To ascertain the difference between the processing latencies of TriJoin and BiStream+Tree, we analyze the processing latency of BiStream+Tree. In BiStream+Tree, since the processing latencies of the cyclic join and chain join are very similar, we only discuss the latency of the chain join. We use $T_{wait}$ to represent the time that $t$ waits for all the intermediate results of $r$ and $s$ that arrive before $t$. We maintain that BiStream+Tree adopts the tuple pair structure to store the intermediate result, and represent the intermediate result as $Ip\_K_RK_{Si}$. The latencies of tuples $r$, $s$, and $t$ are computed by Equation (3).

$$\begin{aligned} L_{bis-r} &= (|K_{Si}| + |Ip\_rK_{Si}| \times |K_{Ti}|) \times \tau + T_{net} \\ L_{bis-s} &= (|K_{Ri}| + |Ip\_sK_{Ri}| \times |K_{Ti}|) \times \tau + T_{net} \\ L_{bis-t} &= |Ip\_K_RK_{Si}| \times \tau + T_{wait},\ or\ |Ip\_K_SK_{Ri}| \times \tau + T_{wait}. \end{aligned} \quad \textbf{(3)}$$

Each processing latency of the symmetric partitioning scheme is lower than that of BiStream+Tree. The value of $\tau$ is the time of the join operation, so $\tau$ is small. If the number of intermediate result is very large, TriJoin and BiStream+Tree can add processing units to reduce the number of intermediate result. Moreover, $|Io\_K_RK_{Si}| \le |Ip\_K_RK_{Si}|$, $|Io\_K_SK_{Ri}| \le |Ip\_K_SK_{Ri}|$, $|K_{Si}| \times \tau + T_{net} \le T_{wait}$, and $|K_{Ri}| \times \tau + T_{net} \le T_{wait}$. Obviously, each processing latency of the symmetric partitioning scheme is lower than that of BiStream+Tree.

In the semi-symmetric partitioning scheme, since the latencies of tuples $r$ and $t$ are the same as those in the symmetric partitioning scheme, the latencies of tuples $r$ and $t$ in the semi-symmetric partitioning scheme are lower than those in BiStream+Tree. Since BiStream+Tree adopts the tuple pair to store the intermediate result, $|Ip\_sK_{Ri}| \ge 1$. The latency of tuple $s$ is smaller than that in BiStream+Tree.

In summary, the latencies of the symmetric and semi-symmetric partitioning schemes are lower than that of BiStream+Tree.

# 5. Implementation

We implement TriJoin on top of Apache Storm, and make the source code publicly available[1]. In this section, we describe the implementations of TriJoin based on the Storm topology and the order-consistent protocol.

### 5.1 System Structure

TriJoin implements four main functional components: DataSpout, DispatchBolt, JoinBolt, and PostProcessBolt. The functions of DataSpout and DispatchBolt correspond to those of preprocessor and dispatcher in the routing component. The functions of JoinBolt and PostProcessBolt correspond to that of processing unit in the join and postprocessing component, respectively.

We condigurate the topology according to the different partitioning schemes. Figure 8 shows the topology for the symmetric partitioning scheme of the cyclic join. Figure 9 shows the topology for the semi-symmetric partitioning scheme of the chain join. DispatchBolt emits tuple by shuffle grouping and all grouping. We implement the shuffle grouping and all grouping methods in round-robin and broadcast manners [25-27], respectively.

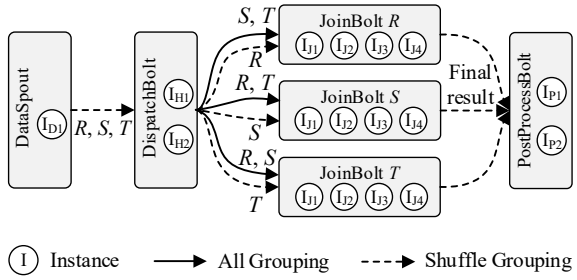


**Figure 8.** Topology of the symmetric partitioning scheme

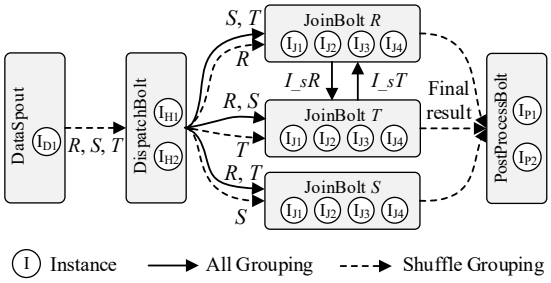


**Figure 9.** Topology for the semi-symmetric partitioning schem

### 5.2 Order Consistency

To guarantee the completeness of the join result, the join processing must have a chronological order. TriJoin does not store the tuples broadcasted to the units for join processing. It discards these tuples immediately after the join operation will cause an incomplete join result, due to out-of-order tuples. Therefore, it is necessary to ensure the order consistency of tuples [8, 28]. We implement order consistency in this paper.

To achieve order consistency of tuples, TriJoin first orders all the tuples. Specifically, the dispatcher maintains a logical timestamp, which is a monotonically increasing counter. The logical timestamp starts from zero and increases sequentially. Whenever the dispatcher receives a tuple from DataSpout, it The dispatcher will append the logical timestamp to the "seq" field of the tuple.

Each dispatcher periodically broadcasts a logical timestamp tuple to all the processing units as a signal. Since the message transmission and processing in each pair of a dispatcher instance and processing unit are first-in-first-out signals, they indicate that the processing unit has received all the tuples before the logical timestamp tuple. If the timestamp tuples of a certain value of all the dispatchers reach a processing unit, the processing unit must have received all the tuples before this timestamp tuple.

Each processing unit uses a priority queue to buffer each received tuple, stores or joins tuples whose timestamps are less than the smallest current latest signal timestamp. Each JoinBolt utilizes the priority queue to sort the received tuples according to the timestamps. Each JoinBolt maintains a table of the latest signal timestamps, which records the latest signal timestamp tuple sent by each DispatchBolt. When JoinBolt receives a stream tuple, it adds the tuple to the priority queue temporarily. When JoinBolt receives a timestamp tuple, it updates the latest signal timestamp table, and determines whether it has received all the timestamp tuples with the same sequence number sent by the dispatcher. If JoinBolt has received all the timestamp tuples, it updates the current smallest latest signal timestamp. Then, it sequentially processes the tuples in the priority queue whose logical timestamps are less than the current smallest latest signal timestamp.

# 6. Evaluation

### 6.1 Setup

We conduct experiments to compare the performance of TriJoin and BiStream+Tree. The experiments run on a cluster of 16 nodes, one works as the Nimbus node, and the remaining 15 work as Supervisor nodes. Each node in the cluster is equipped with 8-core Intel Xeon E5-2670@2.60GHz CPUs, 64 GB memory, 1 TB hard disk, and 1,000 Mbps network. We use real-world datasets collected from the DiDi Chuxing GAIA Initiative [11] and the network traffic traces [29] to evaluate the performance. Each dataset has three subsets. Each subset has 15,000,000 tuples. We use the DiDi Chuxing GAIA Initiative for the comprehensive evaluation, and use the network packet data to evaluate the processing latency [30].

There are six fields "$r_S$, $r_D$, $r_L$, $r_I$, $r_{Int}$, and $r_{Str}$" in each tuple $r$, and the corresponding field types are String, Double, Long, Integer, Integer, and String. Tuples $s$ and $t$ have six fields "$s_S$, $s_D$, $s_L$, $s_I$, $s_{Int}$, and $s_{Str}$", and "$t_S$, $t_D$, $t_L$, $t_I$, $t_{Int}$, and $t_{Str}$", respectively. Tuples $s$ and $t$ have the same field types as the tuple $r$. The size of each tuple stored in memory is approximately 80 bytes. We set the cyclic join as $|r_I - s_I| < \epsilon$ AND $|s_{Int} - t_{Int}| < \epsilon$ AND $|r_{Int} - t_I| < \epsilon$, where $\epsilon$ is a small value according to the

1 https://github.com/CGCL-codes/TriJoin

actual situation. We set the chain join as $|r_1 - s_1| < \epsilon$ AND $|s_{Int} - t_{Int}| < \epsilon$.

### 6.2 Results

We examine the performance of TriJoin and BiStream+Tree for the cyclic join and the chain join. To avoid system initialization and resource configuration from impacting the performance, we record the test results when the system is relatively stable [31]. The following experiments use the dataset of DiDi Chuxing by default.

Figure 10 plots the real-time processing latency over time when the two systems perform a cyclic join. The results show that TriJoin reduces the average latency by 63% compared to BiStream+Tree. Figure 11 presents the real-time throughput over time when the two systems perform a cyclic join. The results show that TriJoin improves the average throughput by 7% compared to BiStream+Tree.

Figure 12 plots the real-time processing latency over time when the systems perform a chain join. The results show that TriJoin reduces the average processing latency by 45% compared to BiStream+Tree. Figure 13 presents the real-time throughput over time when the systems perform a chain join. The results show that TriJoin increases the average throughput by 15% compared to BiStream+Tree.

To examine the impact of system parallelism on latency and throughput, we configureate different numbers of JoinBolt instances in the experiments. Figure 14 presents the latency of two systems with different parallelisms. For the cyclic join, TriJoin reduces the average latency by 51% compared to BiStream+Tree. For the chain join, TriJoin reduces the average latency by 26% compared to BiStream+Tree. When the number of instances is 180, for the cyclic join, TriJoin reduces the latency by 65% compared to BiStream+Tree, while for the chain join, TriJoin reduces the latency by 47%.

Figure 15 plots the throughput of the systems with different parallelisms. For the cyclic join, TriJoin increases the average throughput by 8% compared to BiStream+Tree. For the chain join, TriJoin increases the average throughput by 14%.

To ensure the order consistency of the tuples, each dispatcher needs to broadcast a signal to all the processing units periodically. We set different signal periods. If the dispatcher sends a timestamp signal with a longer signal period, the order consistency is better. However, a longer signal period will result in a higher processing latency. Figure 16 presents the processing latency of the systems under different signal periods. The results show that, for the cyclic join, TriJoin reduces the latency by 68% compared to BiStream+Tree, while for the chain join, TriJoin reduces the latency by 31%. Figure 17 presents the throughput of the two systems under different signal periods. For the cyclic join, TriJoin increases the throughput by 14% compared to BiStream+Tree, while for the chain join, TriJoin increases the throughput by 18%.

To examine the reduced memory cost and computation cost of the join operations of the TuplePacking structure compared to the tuple pair structure, we use DiDi Chuxing to count the number of intermediate results every five minutes. We deploy both the tuple pair and the TuplePacking structure in TriJoin, and denote the tuple pair in TriJoin as TriJoin-pair. TriJoin adopts the TuplePacking structure by default.

Figure 18 plots the number of intermediate results of TriJoin and TriJoin-pair generated by the symmetric partitioning scheme. The results show that TriJoin reduces the number of intermediate results by 71% compared to TriJoin-pair. Figure 19 presents the number of intermediate results of TriJoin and TriJoin-pair generated by the semi-symmetric partitioning scheme. The results show that TriJoin reduces the number of intermediate results by 66% compared to TriJoin-pair.

Figure 20 shows the processing latency of the two systems running under different window lengths. The results show that, for the cyclic join, the average latency of TriJoin is 65% lower than that of BiStream+Tree, while for the chain join, the latency of TriJoin is 44% lower than that of BiStream+Tree.

Figure 21 examines the latency with different parallelisms using the network traffic traces. For the cyclic join, TriJoin reduces the average latency by 51% compared to BiStream+Tree, while for the chain join, TriJoin reduces the latency by 29%. When the number of instances is 180, for the cyclic join, TriJoin reduces the latency by 60% compared to BiStream+Tree, while for the chain join, TriJoin reduces the latency by 44%.

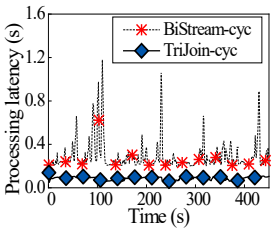


**Figure 10.** The real-time processing latency of the cyclic join

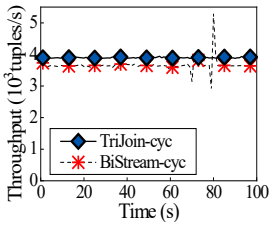


**Figure 11.** The real-time system throughput of the cyclic join

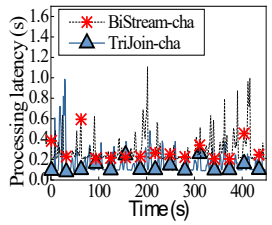


**Figure 12.** The real-time processing latency of the chain join

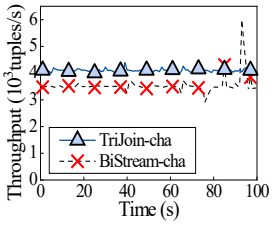


**Figure 13.** The real-time system throughput of the chain join

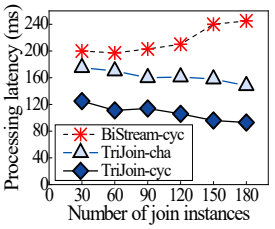


**Figure 14.** Processing latency with different parallelisms

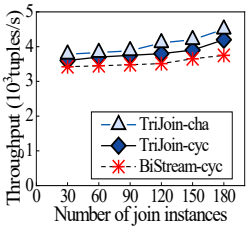


**Figure 15.** Throughput with different parallelisms

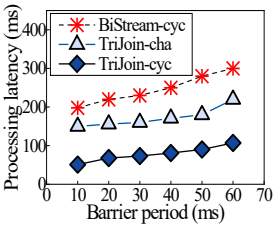


**Figure 16.** Processing latency with different signal periods

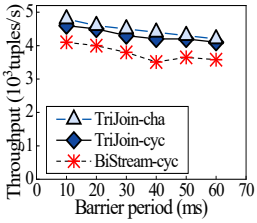


**Figure 17.** Throughput with different signal periods

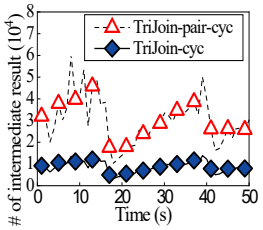


**Figure 18.** The number of intermediate results generated by the symmetric partitioning scheme

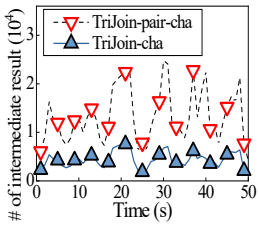


**Figure 19.** The number of intermediate results generated by the semi-symmetric partitioning scheme

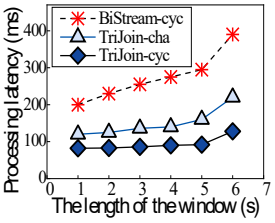


**Figure 20.** Processing latency with the length of the window

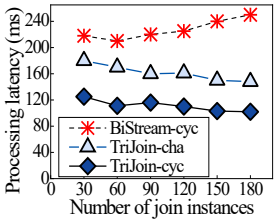


**Figure 21.** Processing latency with different parallelisms using network traffic trace

## 7 Conclusion

In this paper, we propose TriJoin, a time efficient and scalable three-way stream join system. We show through experiments that splitting three-way join results in double two-way joins yields a costly waiting relationship. To solve the problem, we design the symmetric and semi-symmetric partitioning schemes according to two forms of three-way join. The two schemes utilize symmetric partitioning and reused join to decouple the waiting relationship. We use the symmetric partitioning to generate all the intermediate result, and utilize reused join to join the new tuple with the intermediate result and stored tuple in a processing unit. The schemes avoid the waiting in generating the final reslut. The two schemes store the tuples and intermediate results only once, ensuring the scalability of the system. We implement TriJoin on Apache Storm. The experiment results show that TriJoin greatly outperforms the existing designs in terms of processing latency.

## Acknowledgements

This research is supported in part by the National Key Research and Development Program of China under grant No. 2018YFB1004602 and NSFC under grant No. 61972446.

## References

[1] L. Zhang, T. Hu, Y. Min, G. Wu, J. Zhang, P. Feng, P. Gong, J. Ye, A taxi order dispatch model based on combinatorial optimization, *Proceedings of ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, Halifax, NS, Canada, 2017, pp. 2151-2159.

[2] S. Yu, Z. Jiang, D. Chen, S. Feng, D. Li, Q. Liu, J. Yi, Leveraging tripartite interaction information from live stream e-commerce for improving product recommendation, *Proceedings of ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, Virtual Event, Singapore, 2021, pp. 3886-3894.

[3] Y. Huang, B. Cui, J. Jiang, K. Hong, W. Zhang, Y. Xie, Real-time video recommendation exploration, *Proceedings of International Conference on Management of Data (SIGMOD)*, San Francisco, CA, USA, 2016, pp. 35-46.

[4] S. Zhou, F. Zhang, H. Chen, H. Jin, B. B. Zhou, Fastjoin: A skewness-aware distributed stream join system, *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, Brazil, 2019, pp. 1042-1052.

[5] F. Zhang, H. Chen, H. Jin, Simois: A scalable distributed stream join system with skewed workloads, *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, Dallas, TX, USA, 2019, pp. 176-185.

[6] M. Elseidy, A. Elguindy, A. Vitorovic, C. Koch, Scalable and adaptive online joins, *Proceedings of the VLDB Endowment*, Vol. 7, No. 6. pp. 441-452, February, 2014.

[7] A. Shahvarani, H. Jacobsen, Parallel index-based stream join on a multicore CPU, *Proceedings of International Conference on Management of Data (SIGMOD)*, online conference, Portland, OR, USA, 2020, pp. 2523-2537.

[8] Q. Lin, B. C. Ooi, Z. Wang, C. Yu, Scalable distributed stream join processing, *Proceedings of International Conference on Management of Data (SIGMOD)*, Melbourne, Victoria, Australia, 2015, pp. 811-825.

[9] M. Najafi, M. Sadoghi, H. Jacobsen, Scalable multiway stream joins in hardware, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 32, No. 12, pp. 2438-2452, December, 2020.

[10] J. S. Gomes, H. Choi, Adaptive optimization of join trees for multi-join queries over sensor streams, *Information Fusion*, Vol. 9, No. 3, pp. 412-424, 2008.

[11] Didi chuxing gaia initiative, 2021, https://outreach.didichuxing.com/research/opendata/

[12] X. Zhu, H. Gupta, B. Tang, Join of multiple data streams in sensor networks, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 21, No. 12, pp. 1722-1736, December, 2009.

[13] P. Roy, J. Teubner, R. Gemulla, Low-latency handshake join, *Proceedings of the VLDB Endowment*, Vol. 7, No. 9, pp. 709-720, May, 2014.
[14] J. Teubner, R. Müller, How soccer players would do stream joins, *Proceedings of International Conference on Management of Data (SIGMOD)*, Athens, Greece, 2011, pp. 625-636.
[15] M. Najafi, M. Sadoghi, H. Jacobsen, Splitjoin: A scalable, low-latency stream join architecture with adjustable ordering precision, *Proceedings of USENIX Annual Technical Conference (ATC)*, Denver, CO, USA, 2016, pp. 493-505.
[16] L. Lin, H. Chen, H. Jin, Fjoin: An fpga-based parallel accelerator for stream join, *Scietia Sinica Informationis*, Vol. 52, No. 2, pp. 314-333, 2022.
[17] Y. Qiu, S. Papadias, and K. Yi, Streaming hypercube: A massively parallel stream join algorithm, *Proceedings of International Conference on Extending Database Technology (EDBT)*, Lisbon, Portugal, 2019, pp. 642-645.
[18] B. Gedik, Partitioning functions for stateful data parallelism in stream processing, *The VLDB Journal*, Vol. 23, No. 4, pp. 517-539, August, 2014.
[19] C. Balkesen, N. Tatbul, M. T. Ozsu, Adaptive input admission and management for parallel stream processing, *Processings of ACM International Conference on Distributed Event-Based Systems (DEBS)*, Arlington, TX, USA, 2013, pp. 15-26.
[20] J. Yuan, Y. Wang, H. Chen, H. Jin, H. Liu, Eunomia: Efficiently eliminating abnormal results in distributed stream join systems, *Proceedings of International Workshop on Quality of Service (IWQoS)*, Tokyo, Japan, 2021, pp. 1-11.
[21] S. Wang, E. A. Rundensteiner, Scalable stream join processing with expensive predicates: Workload distribution and adaptation by time-slicing, *Proceedings of International Conference on Extending Database Technology (EDBT)*, Saint Petersburg, Russia, 2009, pp. 299-310.
[22] M. Dossinger, S. Michel, Scaling out multi-way stream joins using optimized, iterative probing, *Proceedings of IEEE International Conference on Big Data (IEEE BigData)*, Los Angeles, CA, USA, 2019, pp. 449-456.
[23] M. Dossinger, S. Michel, Optimizing multiple multi-way stream joins, *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, Chania, Greece, 2021, pp. 1985-1990.
[24] F. Li, B. Wu, K. Yi, and Z. Zhao, Wander join: Online aggregation via random walks, *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, San Francisco, CA, USA, 2016, pp. 615-629.
[25] H. Dai, X. Peng, X. Shi, L. He, Q. Xiong, H. Jin, Reveal training performance mystery between tensorflow and pytorch in the single GPU environment, *Science China Information Sciences*, Vol. 65, No. 1, pp. 1-17, January, 2022.
[26] C. Dai, H. Cheng, X. Liu. A Tucker, Decomposition Based on Adaptive Genetic Algorithm for Efficient Deep Model Compression. *Processings of IEEE International Conference on High Performance Computing and Communications (HPCC)*, Yanuca Island, Cuvu, Fiji, 2020, pp. 507-512.
[27] C. Dai, X. Liu, H. Cheng, L. T. Yang, M. J. Deen, Compressing Deep Model with Pruning and Tucker Decomposition for Smart Embedded Systems, *IEEE Internet of Things Journal*, Vol. 9, No. 16, pp. 14490-14500, July, 2022.
[28] B. Ji, Y. Wang, K. Song, C. Li, H. Wen, V. G. Menon, S. Mumtaz, A survey of computational intelligence for 6g: Key technologies, applications and trends, *IEEE Transactions on Industrial Informatics*, Vol. 17, No. 10, pp. 7145-7154, October, 2021.
[29] WIDE project, 2020, http://mawi.wide.ad.jp/mawi/.
[30] H. Chen, H. Jin, S. Wu, Minimizing inter-server communications by exploiting self-similarity in online social networks, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, No. 4, pp. 1116-1130, April, 2016.
[31] Y. Bao, H. Zheng, Q. Zhao, Development and practice of mobile internet experimental platform system, *Journal of Internet Technology*, Vol. 23, No. 2, pp. 407-414, March, 2022.

## Biographies

**Shuiying Yu** is currently working toward the PhD degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, China. Her research interests include stream data processing and distributed algorithms.

**Yinting Zheng** received the master degree with computer science and engineering from Huazhong University of Science and Technology, China. Her research interests include data query and analysis technology.

**Fan Zhang** is currently working toward the PhD degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, China. His research interests include big data processing systems.

**Hanhua Chen** is a professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, China. His research interests include big data processing systems and distributed computing systems.

**Hai Jin** is a Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is a Fellow of IEEE, Fellow of CCF, and a life member of ACM.