The Journal of Supercomputing (2023) 79:8236–8264
https://doi.org/10.1007/s11227-022-04991-6

An adaptive non-migrating load-balanced distributed stream window join system

Qihang Wang[1] · Decheng Zuo[1] · Zhan Zhang[1] · Siyuan Chen[1] · Tianming Liu[1]

Accepted: 29 November 2022 / Published online: 15 December 2022

Abstract
Stream processing systems are widely used to process large amounts of data generated by applications in real time due to their advantages in latency and throughput. In most streaming applications, the system requires a comprehensive analysis of data from multiple data sources, so stream joins are the basis of stream processing systems. Similar to other big data problems, stream joins suffer from load imbalance, where a few nodes responsible for handling most of the load can become bottlenecks, thereby increasing latency and reducing throughput. Therefore, how to obtain a good load-balancing effect with low overhead is a critical issue in designing stream join systems. To solve this problem, we propose an adaptive non-migrating load-balancing method, which is mainly oriented to the stream window join problem. Considering that the completeness of the stream join results during the splitting of state to multiple downstream instances can be guaranteed by replicating the input tuples into multiple replicas and sending them to those downstream instances, our method can control the replication and forwarding of input tuples by setting up routing tables, and then when the system becomes unbalanced, our method can change the load distribution of the system by directly changing the partitioning of the tuples arriving later instead of state migration, and thus achieving load balancing with very low overhead. Based on our method, we develop a distributed stream window join system, NM-Join, which is built on Flink. We theoretically analyze the completeness and effectiveness of our method and provide extensive experimental evaluations of NM-Join in terms of load-balancing effect, latency, and throughput. Experimental results show that our method is able to perform load balancing with very low additional overhead, and thus outperforms existing load-balancing methods in terms of latency and throughput.

Keywords Big data · Distributed stream join system · Data skew · Dynamic load balancing · Non-migrating

✉ Zhan Zhang
zz@ftcl.hit.edu.cn

Extended author information available on the last page of the article

1 Introduction

In recent years, the distributed stream processing system has gained rapid development and industrial acceptance due to its advantages in processing latency and throughput [1, 2]. It has been used in areas such as search engines [3], social networks [4], financial transactions, and the Internet of Things [5–7]. In these applications, it is necessary to join the stream from multiple data sources together to process a complex query. Since the join operation will consume a lot of system resources, efficiently performing the stream join becomes the key to improving system performance.

Considering that memory is limited and stream data is infinite, window join is commonly used today to solve the join problem between streams. Similar to the traditional big data processing problem, the performance of distributed stream join system is greatly affected by data skew. The existence of data skew will cause a few processing units to take up most of the load, and these processing units will become the bottleneck and degrade system performance. This requires a load-balanced stream window join system.

In order to design an efficient load-balanced stream window join system, the following two requirements must be considered: (i) ability to perform load balancing when the input is skewed to obtain optimal system performance; and (ii) performing load balancing with low overhead so that system performance is not degraded too much. Traditional stream window join systems mainly adopt the following two categories of load-balancing methods to achieve load balancing, one is the static load-balancing method [8–11], and the other is the migration-based dynamic load-balancing method [12]. The static load-balancing methods perform load balancing by designing a static partitioning scheme for input tuples, mainly including the random partitioning method [8, 9] and ContRand [10]. The random partitioning method broadcasts all input tuples to all processing units for processing, but it will result in each input tuple being processed multiple times in all processing units, consuming a lot of additional network and computational resources. The ContRand divides all processing units into groups, using hash partitioning among all groups and random partitioning within each group. ContRand reduces the overhead of the random partitioning method due to tuple replication, but has two drawbacks: (1) it is not adaptive to the input, so load imbalance may occur among groups; (2) each input tuple is still processed multiple times in all processing units within a group. The migration-based dynamic load-balancing methods will monitor the load distribution of the system, and perform load balancing by data migration when the load distribution is unbalanced. The migration-based dynamic load-balancing method can be adaptive to the input and does not need to replicate the input tuples. However, this method needs to perform data migration during load balancing, and the tuples cannot be processed during the data migration, thus greatly increasing the transient latency of the system. In summary, the load-balancing methods proposed so far for stream join systems are either not adaptive to the input, or have too much load-balancing overhead. Therefore, the purpose of our work is to design a novel load-balancing method for distributed stream join systems, which can adapt well to the input and does not introduce excessive load-balancing overhead, thus achieving lower latency and higher throughput.

In order to overcome the weaknesses of traditional methods, in this paper, we propose a non-migrating adaptive load-balancing method for distributed stream window join system, which enables dynamic load balancing with very low overhead and, therefore, has advantages in terms of throughput and latency. The core idea of our method is to monitor system information during system runtime and perform load balancing by controlling the replication and forwarding of tuples instead of data migration when the system is unbalanced. We divide all tuples into two types, i.e., store tuples and join tuples, depending on how the tuple is processed inside the processing unit. The former will be stored in the corresponding processing unit, while the latter will be matched with the stored tuples in the corresponding processing unit without updating the state of the unit. When performing load balancing, the load distribution is controlled by changing the partitions of the store tuples, and the completeness is ensured by replicating the join tuples into multiple copies and sending them to all the partitions that may generate join results.

Our method has the following three important features: (i) the system adopts the routing table for partitioning, and when the system is load-balanced, each input tuple is sent to only one processing unit for processing (take the equal-join as an example), thus reducing the consumption of computational resources due to redundant processing; (ii) the system collects relevant statistics during system runtime and performs load balancing when there is a load imbalance in the system, so it can be adaptive to the input and maintain the system in a load-balanced state at all times; (iii) the system performs load balancing in a non-migrating way, so there is no need to pause the processing of input tuples during load balancing, thus reducing the processing latency of the system during load balancing.

We next propose a load-balanced stream window join system, NM-Join, based on our method. NM-Join uses the coordinator to periodically collect relevant statistics and calculate load-balancing schemes, and uses routing tables to control the forwarding of store tuples and join tuples, thus performing load balancing in a non-migrating way when the load is unbalanced. In addition, NM-Join divides the window into several sub-windows and calculates the load-balancing scheme based on the sub-windows, which obtains a better load-balancing effect.

In summary, we make the following contributions:

1. We propose a non-migrating load-balancing method for distributed stream window join systems, and design and implement an adaptive load-balanced distributed stream window join system, NM-Join, based on the proposed method, which can perform load balancing with low overhead.
2. We theoretically analyze the completeness and effectiveness of the proposed techniques and also conduct extensive experimental evaluations. The evaluation results confirm that NM-Join is effective in terms of load-balancing effect, latency, and throughput.

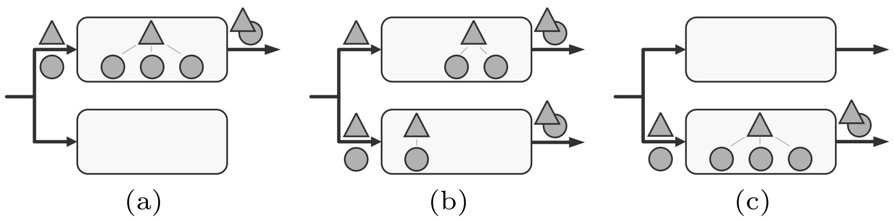


Fig. 1 Non-migrating load-balancing procedure

The paper is organized as follows. Section 2 introduces our load-balancing method and the detailed design of NM-Join. Section 3 gives the theoretical analysis of NM-Join. Section 4 evaluates the performance of NM-Join. Section 5 discusses the related work about stream join systems. Section 6 concludes the paper.

2 System design

In this section, we first introduce our non-migrating load-balancing method, next introduce the design of NM-Join and focus on the design of the store routing table and the join routing table.

2.1 Non-migrating load-balancing procedure

The non-migrating load-balancing procedure is shown in Fig. 1. In a distributed stream window join system, all tuples arriving at a processing node are logically divided into two categories, i.e., *store tuples* and *join tuples*. When a store tuple (represented by the circle in Fig. 1) is received, the processing node stores it in the local index structure; when a join tuple (represented by the triangle in Fig. 1) is received, the joiner compares it with all locally stored tuples that satisfy the time window condition belong to the relative stream and outputs all matched results that satisfy the join predicate, after which the join tuple is discarded.

In our method, to reduce the redundant network resource overhead and computational resource overhead due to tuple replication, we forward all tuples based on their keys. In this paper, processing nodes are also called partitions. We refer to the partition to which the store tuples with key k will be sent as the *store partition* of k. Meanwhile, we refer to the partition to which the join tuples with key k will be sent as the *join partition* of k. In Fig. 1, all the store tuples have the same key, and all the join tuples matched with that key.

Based on the description of the store tuple and join tuple, it can be observed that when the partitioning scheme of all store tuples is determined, in order to produce all join results, all join tuples must be sent to their respective corresponding partitions, and thus the partitioning scheme of all join tuples is also determined. Therefore, the partitioning scheme of the store tuples determines the load distribution of the system, and we can adjust the load distribution of the system by adjusting the partitioning scheme of store tuples.

As shown in Fig. 1a, initially all store tuples with a certain key are sent to the upper partition, at which point the corresponding join tuples are also sent to that partition. After a period of time, as shown in Fig. 1b, when the system is load-unbalanced, we recalculate the partitioning scheme of store tuples, change the store partition of some keys from a high-load partition to a low-load partition, and send the store tuples with these keys to the new partition. In this case, the store tuples with these keys are stored in both the old and new partitions, and in the traditional methods, all the store tuples stored in the old partition need to be migrated to the new partition in order to ensure that all join results are correctly generated. In contrast, instead of migration, we send the join tuples corresponding to these keys to both the old and new partitions, which also can generate all the join results correctly.

Since our method targets the window join problem, after a period of time, as shown in Fig. 1c, the store tuples stored in the old partition are removed from the old partition because they do not satisfy the window condition. At this point, all store tuples with these keys are stored only in the new partition, so all join tuples need only be sent to the new partition, and the system accomplishes load balancing in a non-migrating way.

2.2 NM-Join system architecture

Next, we present a distributed stream window join system, NM-Join, based on our non-migrating method. Figure 2 illustrates the overall architecture of NM-Join, which consists of three main components: *router*, *joiner*, and *coordinator*. NM-Join is similar to SplitJoin in that each joiner instance is responsible for storing and joining partial tuples of the two input streams, and each tuple is stored in only one joiner instance. However, unlike SplitJoin, NM-Join uses routing tables to specify the partitions of different tuples, instead of using broadcast to distribute tuples.

Router The router is responsible for ingesting tuples arriving at the system and partitioning them. When a tuple arrives at the router, the router constructs a store tuple for it and sends it to the corresponding store partition for storing, and constructs several join tuples for it and sends them to the corresponding join partitions for joining. There are two routing tables in each router instance for each input stream, i.e., the *store routing table* and the *join routing table*. The two routing tables control the partitioning scheme of the store tuples and the join tuples, respectively. We will describe the two routing tables in detail in Sects. 2.3 and 2.4. In addition, when the system performs load balancing, the router receives the latest store routing table from the coordinator and updates its local join routing table based on the store routing table. When both the store routing table and the join routing table are updated, the router starts forwarding tuples according to the new routing tables to perform load balancing.

Joiner The joiner is responsible for performing the actual join operation. When a tuple arrives at a joiner instance, the joiner instance performs the store or join operation depending on the type of the tuple. Since NM-Join is oriented toward window

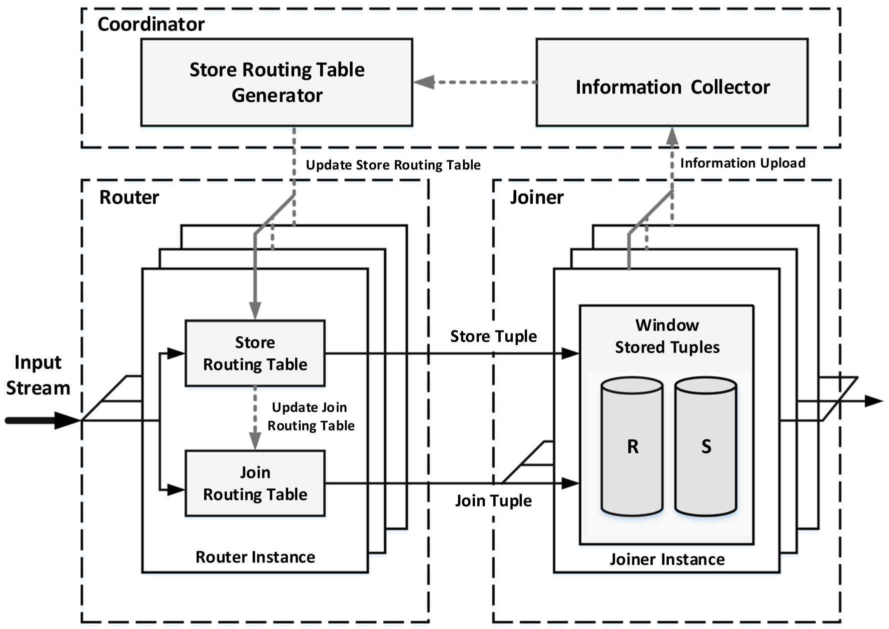


Fig. 2 The architecture of NM-Join

join and according to the description of window join in [13], each tuple in joiner needs to perform three steps to complete a window join, i.e., probe, insert, invalidate, we use the structure of the chained in-memory index [10] to reduce the overhead of window tuples invalidate. The chained in-memory index organizes all the stored tuples into several sub-indexes based on their timestamp, and the tuples are deleted with the granularity of the sub-indexes. In addition, to calculate the load-balancing scheme, each joiner instance periodically uploads its local load information and related statistics to the coordinator.

Coordinator The coordinator periodically collects load information and statistics uploaded by each joiner instance and then determines whether the system needs to perform load balancing. If the coordinator determines that the system needs to perform load balancing, the coordinator will recalculate a store routing table based on the statistics and then notify each router instance to get the latest store routing table.

2.3 Store routing table design

The store routing table is responsible for determining the store partition of each key and its structure is shown in Fig. 3. Each entry in the store routing table consists of <`key`, `partition`>, which represents the store partition of the key. Note that since each tuple is stored only once in all joiner instances, there is only one store partition for each key. In order to perform load balancing, the metric of load

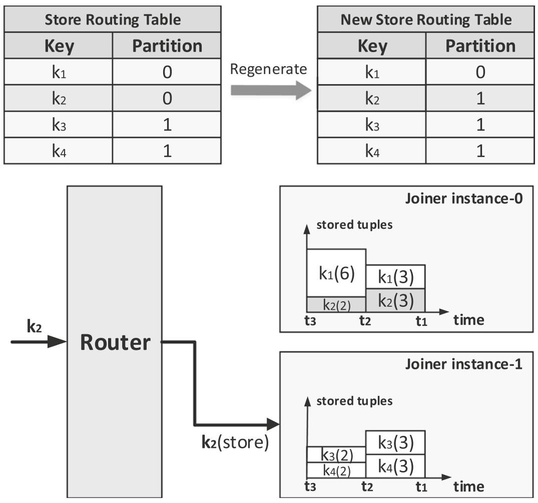


Fig. 3 Store routing table generation process

imbalance needs to be defined first. In this paper, we use the number of stored tuples as a measure of load, and the system load imbalance factor is defined as follows.

Definition 1 (*Load imbalance factor*) The system load imbalance factor is expressed as the ratio of the difference between maximum load and average load to the average load overall joiner instances. The load imbalance factor u can be represented as:

$$u = \max_{0 \le i < P} \frac{L_i - L_{avg}}{L_{avg}} \tag{1}$$

where P is the number of joiner instances, L_i denotes the load of the i-th joiner instance, and L_{avg} denotes the average load overall joiner instances. Meanwhile, the load imbalance threshold θ is defined, and the system is considered to suffer from load imbalance when $u \geq \theta$.

In NM-Join, in order to get a better load-balancing effect, we divide the window into several sub-windows and use the sub-windows as granularity to determine the degree of system load imbalance and generate a new store routing table. In Sect. 3.2, we will prove that we can obtain a good load-balancing result by using the sub-windows. For this purpose, we define the partial load imbalance factor as follows.

Definition 2 (*Partial load imbalance factor*) Assuming that the time window of the stream window join system is w, we divide w into n sub-windows, and the partial load imbalance factor of the j-th sub-window can be expressed as follows:

$$u'_j = \max_{0 \le i < P} \frac{L'_{i \cdot j} - L'_{avg \cdot j}}{L'_{avg \cdot j}}, \ 0 \le j < n \tag{2}$$

where $L'_{i \cdot j}$ denotes the increment load of the i-th joiner instance during the j-th sub-window, and $L'_{avg \cdot j}$ denotes the average increment load of all joiner instances during the j-th sub-window. For example, in Fig. 3, the whole time window (t_1, t_3) is divided into two sub-windows, i.e., (t_1, t_2) and (t_2, t_3), and we can calculate the partial load imbalance factor for each of the two sub-windows.

In NM-Join, the coordinator periodically calculates the partial load imbalance factor u'_0 of the latest sub-window and decides to perform load balancing when $u'_0 \ge \theta$. Unlike the global load imbalance factor, the partial load imbalance factor can react faster to changes in load distribution.

When the coordinator decides to perform load balancing, it recalculates a new store routing table based on the relevant statistics collected in the latest sub-window. We use a heuristic algorithm to generate the new store routing table. The core idea of the algorithm is to assign each key to a partition in sequence until the load assigned to that partition reaches the average load, after which the algorithm repeats above procedure to assign the remaining keys to the remaining partitions. It should be noted that the store routing table generation algorithm is not the focus of our study, and other algorithms can be used instead of this algorithm.

We use the example in Fig. 3 to demonstrate the generation process of a new store routing table. Assume that there are only four unique keys in the input stream, i.e., k_1, k_2, k_3, k_4, and that there are only two joiner instances in the system. Each rectangle in each joiner instance represents the tuples stored in that instance during the corresponding time range, and the contents of each rectangle represent its key and the number of the corresponding stored tuples.

At the beginning, the store partition of k_1, k_2 is 0, and the store partition of k_3, k_4 is 1. Assuming that the load imbalance threshold θ is 0.2, and at t_3, we can observe that the increased loads of joiner instance-0 and joiner instance-1 during the latest sub-window (t_2, t_3) are 8 and 4, respectively, and the partial load imbalance factor can be calculated as 0.33 according to (2), which exceeds the threshold, so the store routing table is recalculated. According to the heuristic algorithm, the system first assigns k_1 to instance-0. At this point, the assigned load of instance-0 is 6, which reaches the average load, so the system then assigns the remaining keys to the remaining partitions and generates a new store routing table which is shown as the New Store Routing Table in Fig. 3. After t_3, the store tuples of k_2 will be sent to joiner instance-1.

2.4 Join routing table design

The purpose of the join routing table is to ensure that all join results can be generated correctly when adjusting the partitions of store tuples. The structure of the join routing table is shown in Fig. 4. Each joinEntry in the join routing table consists of `<key, storePartitionEntryList>`

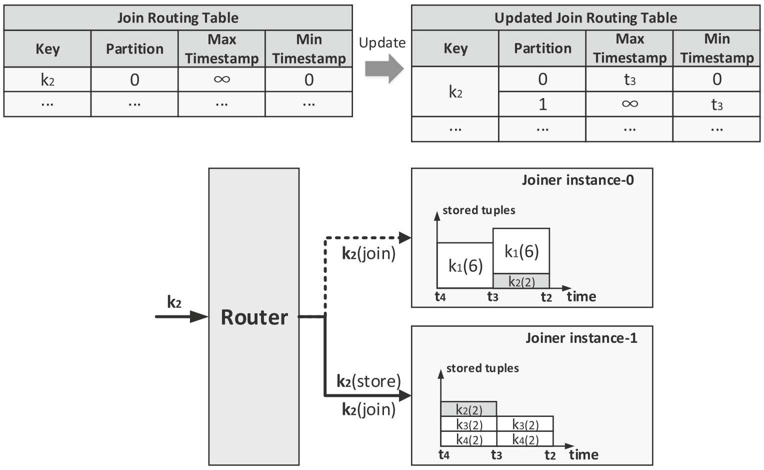


Fig. 4 Join routing table update process

and each storePartitionEntry in the storePartitionEntryList consists of <`storePartition`, `maxTimestamp`, `minTimestamp`>, which indicates all partitions in which the key may have been stored and the maximum and minimum timestamp among all stored tuples with that key in the corresponding partition. Note that since the store partition of each key in the store routing table may change, the store tuples with a certain key may be stored in multiple partitions, so there may be more than one storePartition for each key in the join routing table.

When calculating the join partitions of a key k, we first calculate all the keys matched with k based on the join predicate, and then look up all the storePartitions of these keys in the join routing table of the relative stream, which are the join partitions of k. After this, we send to each of these partitions a join tuple attached with the maximum and minimum timestamp of the corresponding storePartitionEntry. The purpose of this is that since tuples are stored as a chained in-memory index in each joiner instance, the maximum timestamp and the minimum timestamp can specify the time range of the stored tuples in the corresponding joiner instance so that the sub-indexes that do not satisfy this time range can be ignored to speed up the query.

In summary, it can be found that there are two types of operations related to changing the join routing table, i.e., *update* and *expire*, which we explain in detail next.

Update Based on the characteristics of the join routing table, it can be observed that the join routing table needs to record the new store partition and old store partition of each key before and after its store partition is changed, so it should be updated based on both the new and old store routing tables. The join routing table update algorithm is shown in Algorithm 1.

Algorithm 1 Join Routing Table Update Algorithm

Input: Store routing table before update, $RT_{store.old}$; Store routing table after update, $RT_{store.new}$; System current time, $t_{current}$; Join routing table before update, RT_{join};

Output: Join routing table after update, RT_{join};

```
1:  for each joinEntry $E$ in $RT_{join}$ do
2:      Find store partition $P_{old}$ from $RT_{store.old}$ based on $E.key$
3:      Find store partition $P_{new}$ from $RT_{store.new}$ based on $E.key$
4:      if $P_{old}$ != $P_{new}$ then
5:          Get the storedPartitionEntryList $List_e$ of $E$
6:          Get the storedPartitionEntry $e_{old}$ corresponding to $P_{old}$ from $List_e$
7:          $e_{old}.maxTimestamp \leftarrow t_{current}$
8:          Create a new storedPartitionEntry $e_{new}$
9:          $e_{new}.storedPartition \leftarrow P_{new}$
10:         $e_{new}.minTimestamp \leftarrow t_{current}$
11:         $e_{new}.maxTimestamp \leftarrow \infty$
12:         Insert $e_{new}$ to $List_e$
13:     end if
14: end for
15: return $RT_{join}$
```

We use the example in Fig. 4 to show the update process of join routing table. Take k_2 as an example. To simplify the analysis, we assume that the system is a self-join system and the join predicate is equal-join, so at the beginning, the join partition of k_2 is equal to its store partition, i.e., 0. As mentioned in Sect. 2.3, the store partition of k_2 is changed at t_3, so the join routing table is also updated at this point. Firstly, we find the old storePartitionEntry of k_2 and set its maxTimestamp to the current time t_3; secondly, insert a new storePartitionEntry corresponding to the new store partition 1, and set its minTimestamp to t_3 and maxTimestamp to infinity. The updated join routing table is shown in Fig. 4. After t_3, since the store tuples of k_2 are stored in both instances, the join tuples of k_2 are sent to both instances to ensure that all join results can be generated.

Expire The purpose of the expire is to stop sending join tuples to the partitions where there are no longer stored tuples that matched with the key. For that purpose, the system periodically traverses all the storePartitionEntrys in the join routing table and then removes the entries whose maxTimestamp does not satisfy the window condition. As shown in Fig. 5, at t_5, the tuples arriving before t_3 are expired, and the entries in the join routing table associated with partition 0 are deleted because its maxTimestamp is t_3. The join routing table after expiration is shown in Fig. 5. After t_5, the store tuples of k_2 are only stored in instance-1, so the join tuples of k_2 are also only sent to instance-1.

3 Analysis

In this section, we analyze the completeness and effectiveness of NM-Join.

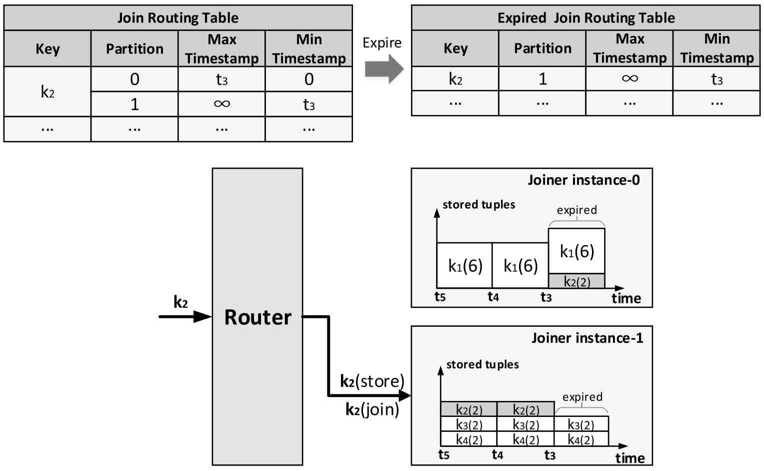


Fig. 5 Join routing table expiration process

3.1 Completeness

Our method is mainly oriented to interval time window join between two streams (R and S). Assuming that all tuples arrive in order, taking the tuple $s \in S$ as an example, the interval time window join means finding the set $M_R(s)$ that satisfies the following conditions:

$$M_R(s) = \{r \mid r \cdot key \overset{p}{=} s \cdot key, r \in R, s.t - w^- \leq r.t \leq s.t + w^+\} \tag{3}$$

where $r \cdot key$ and $s \cdot key$ represent the keys corresponding to the tuples, $\overset{p}{=}$ represents that the two tuples satisfy the join predicate p. w^- and w^+ represent the interval time window range, which are two fixed positive constants, and $r.t$ and $s.t$ represent the timestamps of r and s, respectively. Here we define the completeness of the stream window join system.

Definition 3 (*Stream window join completeness*) We determine that a stream window join system has completeness if and only if for $\forall s \in S$ (resp. $\forall r \in R$), the system can find $M_R(s)$ (resp. $M_S(r)$), and for $\forall r \in M_R(s)$ (resp. $\forall s \in M_S(r)$), the system outputs only one (r, s) tuple pair.

In the following analysis, we assume that all tuples can reach each joiner instance in the sequence of time. We analyze the case of equal-join, and this analysis can be simply extended to other join predicates. We use $T_r(t_o, t_l)$ and $T_s(t_o, t_l)$ to denote all the tuples of R-stream and S-stream with key k that arrive at the system during the time interval (t_o, t_l). To simplify the analysis, we assume that the partition scheme of both streams is the same, the store partition of k is changed from partition P_o to P_d at time t_c, and ignoring the expire operation. Therefore, in our method, both the store partition and join partition of $T_r(-\infty, t_c)$ and $T_s(-\infty, t_c)$ are P_o. The store partition of $T_r(t_c, +\infty)$ and $T_s(t_c, +\infty)$ is P_d, while the join partitions of $T_r(t_c, +\infty)$ and $T_s(t_c, +\infty)$ are P_o and P_d. Since the system can find $M_R(s)$ and generate only one join result for each tuple if s can be compared only once with $T_r(s.t - w^-, s.t + w^+)$, we first present the following lemma.

Lemma 1 *In our method, s (resp. r) can be compared only once with $T_r(-\infty, s.t)$ (resp. $T_s(-\infty, r.t)$).*

Proof If $s.t < t_c$ (resp. $r.t < t_c$), the store tuples of $T_r(-\infty, s.t)$ (resp. $T_s(-\infty, r.t)$) are stored in P_o, and the join tuple of s (resp. r) will be sent to P_o and be compared only once with $T_r(-\infty, s.t)$ (resp. $T_s(-\infty, r.t)$). And if $t_c \leq s.t$ (resp. $t_c \leq r.t$), the store tuples of $T_r(-\infty, t_c)$ (resp. $T_s(-\infty, t_c)$) are stored in P_o and the store tuples of $T_r(t_c, s.t)$ (resp. $T_s(t_c, s.t)$) are stored in P_d. At this point, the join tuples of s (resp. r) will be sent to P_o and P_d, and the join tuple in P_o will be compared only once with $T_r(-\infty, t_c)$ (resp. $T_s(-\infty, t_c)$), the join tuple in P_d will be compared only once with $T_r(t_c, s.t)$ (resp. $T_s(t_c, r.t)$), so the join tuples of s (resp. r) will be compared only once with $T_r(-\infty, s.t)$ (resp. $T_s(-\infty, r.t)$). $\square$

Next we prove the completeness of NM-Join.

Theorem 1 *The NM-Join system has completeness.*

Proof Take s as an example, based on Lemma 1, we can know that s can be compared only once with $T_r(s.t - w^-, s.t)$, and $\forall r \in T_r(s.t, s.t + w^+)$ can be compared only once with $T_s(-\infty, r.t)$, so $\forall r \in T_r(s.t, s.t + w^+)$ can be compared only once with s (because $s.t < r.t$). In summary, for $\forall s \in S$, s are able to be compared only once with $T_r(s.t - w^-, s.t + w^+)$, so the completeness of NM-Join is proved. $\square$

3.2 Effectiveness

Next, we show the load-balancing effect of NM-Join under ideal conditions. Here, we assume that the store routing table generation algorithm is efficient, i.e., each newly generated store routing table is able to reduce the partial load imbalance factor of the next sub-window when the system load is unbalanced and the input distribution does not change. We first present the following lemma.

Lemma 2 *In NM-Join, the partial load imbalance factor u' of each sub-window does not exceed θ.*

Proof Since the distribution of the input stream rarely changes radically in practice, we can divide more sub-windows to make the time range of each sub-window small enough (since the overhead of our method is small, this divide will not bring too much overhead to the system), so that the change of data distribution within each sub-window is not too large, and thus it can be considered that the system can detect and trigger the load balancing when the partial load imbalance factor u' of a sub-window has just reached θ.

When a sub-window w_s triggers load balancing due to the partial load imbalance factor reaching θ, the distribution of input stream within two adjacent sub-windows can be approximated as equal due to the small range of each sub-window and the slow change of data distribution, so the store routing table calculated based on the statistics of w_s can reduce the partial load imbalance factor of the next sub-window, i.e., the partial load imbalance factor of the next sub-window will be less than θ. After this, if the distribution of the input stream continues to change, the system can repeat the above process so that the partial load imbalance factor of each sub-window does not exceed θ. □

We next show the load-balancing effect of NM-Join.

Theorem 2 *The load imbalance factor u of NM-Join is upper bounded by θ.*

Proof Assume that a window contains d sub-windows, based on Lemma 2, for each sub-window the following inequality holds:

$$u'_j \leq \theta, j = 1, 2, \dots d \tag{4}$$

Therefore the global load imbalance factor of the system is:

$$\begin{aligned} u &= \max_{0 \leq i < P} \frac{L_i - L_{avg}}{L_{avg}} \\ &= \max_{0 \leq i \leq P} \frac{L_i - \sum_{j=1}^{d} L'_{avg \cdot j}}{\sum_{j=1}^{d} L'_{avg \cdot j}} \\ &\leq \frac{\sum_{j=1}^{d} (\max_{0 \leq i < P} L'_{i \cdot j}) - \sum_{j=1}^{d} L'_{avg \cdot j}}{\sum_{j=1}^{d} L'_{avg \cdot j}} \\ &= \frac{\sum_{j=1}^{d} (L'_{avg \cdot j} \cdot u'_j)}{\sum_{j=1}^{d} L'_{avg \cdot j}} \\ &\leq \frac{(\sum_{j=1}^{d} L'_{avg \cdot j}) \cdot \max_{1 \leq j \leq d} u'_j}{\sum_{j=1}^{d} L'_{avg \cdot j}} \\ &\leq \theta \end{aligned} \tag{5}$$

□

In summary, it can be seen that NM-Join has an upper bound of θ for the global load imbalance factor and is more responsive to load distribution changes due to the use of sub-windows.

4 Evaluation

NM-Join[1] is built on Flink and uses Kafka as the input stream adapter. Meanwhile, the components communicate with each other through Zookeeper. In this section, we compare NM-Join with the traditional static load-balancing methods and the migration-based dynamic load-balancing methods. The static methods include Hash, Random [8], and ContRand [10]. In ContRand, we divide all the joiner instances into 16 groups. The migration-based method (MK-Join) follows the general migration procedure [12], i.e., monitoring the system state, calculating the migration scheme, and later performing the migration scheme by data migration. To sharpen the focus of the comparison, the migration-based method takes a load-balancing algorithm similar to NM-Join, which differs only in that it replaces the sub-window with the whole window. We set the load imbalance threshold to 0.2 in our experiments, and we describe our reasons for this setting at the end of this section.

4.1 Experimental setup

Next, we introduce the experimental setup.

Environment We conduct all experiments on a Flink cluster consisting of 16 servers. Each server in the cluster runs CentOS 7.3.1611 and has 8 G RAM. Meanwhile, each server is set to have 4 slots, so there are at most 64 parallelisms available for our experiments.

Datasets We conduct experiments with synthetic datasets as well as TPC-H datasets. There are 100k unique keys in each synthetic dataset and the distribution of these keys follows the Zipf distribution. The Zipf coefficient which determines the degree of skewness is set to 0.0, 0.2, and 0.6, denoted by Zipf−0.0, Zipf−0.2, and Zipf−0.6, respectively, and each tuple in the synthetic datasets is appended with the current system time as the timestamp when it is generated. In each experiment, the distribution and input rates of the two input streams are the same. In addition, we generated 40GB TPC-H data for testing, which have been preprocessed to follow the Zipf distribution and have a Zipf coefficient of 0.2.

Queries We consider testing with two band-join queries because the band-join is more representative (it can be transformed into other join predicates such as equi-join by resizing the range). For the synthetic datasets, the band-join is as follows:

SELECT * **FROM** INPUT1 I1, INPUT2 I2
WHERE ABS (I1.key − I2.key) <= 1

For the TPC-H datasets, the band-join (BCI) which is presented in [14] is as follows:

SELECT * **FROM** LINEITEM L1, LINEITEM L2
WHERE ABS (L1.shipdate − L2.shipdate) <= 1
AND (L1.shipmode = 'TRUCK' AND L2.shipmode ! = 'TRUCK')
AND L1.Quantity > 45

[1] https://github.com/wufengliaoyu/NMJoin.

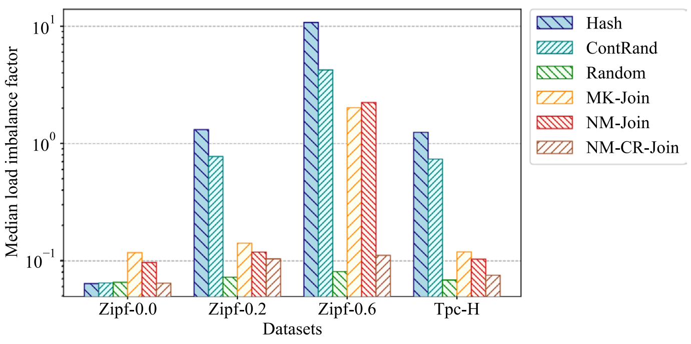


Fig. 6 Median load imbalance factor with various static datasets

We use event-time semantics for synthetic datasets and ingest-time semantics for TPC-H datasets when performing queries. Meanwhile, we set the time window size to 10 min in all experiments.

4.2 Static input experiment

We first test the performance of the different methods when the distribution of the two input streams does not change. We mainly compare the methods in terms of load-balancing effect, maximum throughput, and latency.

Load-balancing effect We first compare the load-balancing effects of different methods and use the load imbalance factor as a measure of the load-balancing effect. We conduct the experiments with the synthetic datasets and the TPC-H dataset, respectively. We periodically collect load distribution information and calculate the load imbalance factor. Figure 6 provides a comparison of the median load imbalance factor for the different methods at an input rate of 2k tuples per second.

As can be seen from the figure, for the datasets with skewed data (Zipf−0.2, Zipf−0.6, and Tpc-H), Hash has the highest load imbalance factor since it does not adopt any load-balancing strategy. For ContRand, since it randomly assigns tuples to all nodes within each group, all nodes within each group have the approximately equal load. Therefore, ContRand is able to reduce the overall load imbalance of the system compared to Hash, which can also be seen in the figure. However, ContRand cannot handle the load imbalance between each group, so its overall load imbalance factor is still high. For MK-Join and NM-Join, they have better load-balancing effects compared to Hash and ContRand because they can adapt to the input of the system and distribute the load equally among all nodes. This can also be seen in the figure, where MK-Join and NM-Join have similar load imbalance factors in the presence of skewed data, and both are lower than Hash and ContRand, which confirms that NM-Join can achieve a load-balancing effect similar to MK-Join and better than ContRand.

It can also be seen from the figure that the load imbalance factors of MK-Join and NM-Join are slightly higher than those of Hash and ContRand when the input data is not skewed. This is because the input data has a certain degree of randomness,

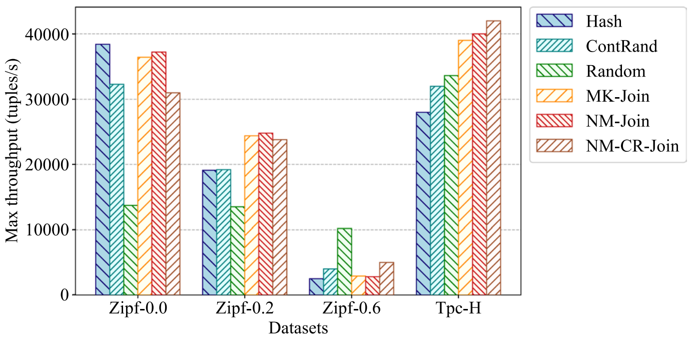


Fig. 7 Maximum throughput with various static datasets

which may trigger the load-balancing adjustment of the system during the runtime, and the heuristic load-balancing algorithm we adopted cannot achieve perfect load balancing, thus causing the load imbalance factor of the system to be slightly higher than the ideal load imbalance factor. However, the difference between the two is not significant and thus the impact on the system performance is minimal.

In order to further reduce the load imbalance factor of the system, we combine NM-Join and ContRand to propose a new load-balancing method named NM-CR-Join. NM-CR-Join divides all joiner instances into groups, adopts a random method within each group, and adopts our non-migrating load-balancing method between groups. In NM-CR-Join, as in ContRand, we divide all joiner instances into 16 groups. The corresponding experimental results are shown in Fig. 6. It can be seen that compared to NM-Join, NM-CR-Join further reduces the load imbalance factor due to its ability to both balance the load between groups and balance the load within groups.

The experimental results also show that Random has a very small load imbalance factor for any input, due to its ability to distribute the load evenly among all nodes. However, in the next experiments, we will show that the cost of Random to achieve this even load distribution is huge and can seriously degrade the performance of the system.

Maximum throughput Next, we study the maximum throughput of different methods with various datasets. In each experiment, we gradually increase the input rate of the system, and when the system is backpressured, we record the current input rate as the maximum throughput. Figure 7 provides the maximum throughput of the different methods with various datasets.

As can be seen from the figure, the maximum throughputs of NM-Join and MK-Join are approximately equal for all datasets, which confirms that for static datasets, NM-Join is able to achieve similar adaptiveness to the input as MK-Join. Compared to Hash, NM-Join is able to achieve higher maximum throughput in the case of skewed input. This is due to the fact that load imbalance occurs in Hash when there is skew in the input, so a few nodes take on more load at the same input rate and reach the upper limit of throughput faster. Meanwhile, the maximum throughput of Hash is slightly larger than that of NM-Join when the input is not skewed (Zipf−

0.0), this is due to the aforementioned fact that the load imbalance factor of NM-Join is slightly higher than that of Hash when the input is not skewed. However, as can be seen from the graph, the difference between the two is very small at this point.

Compared with ContRand and Random, NM-Join achieves higher maximum throughput when the input is not very skewed (Zipf−0.0, Zipf−0.2, and Tpc-H). This is partly because NM-Join achieves a better load-balancing effect compared to ContRand when the input is skewed, and partly because there is no redundant computation in NM-Join. In ContRand and Random, since all tuples with a certain key need to be sent randomly to a node within a group (in ContRand) or to one of all downstream nodes (in Random), all tuples arriving subsequently to be joined with that key need to be replicated as multiple replicas and to be sent to all nodes within the same group (in ContRand) or all downstream nodes (in Random) in order to guarantee the completeness of the join results, thus resulting in a large amount of redundant computation. In our experimental configuration, each tuple needs to be replicated as 4 replicas in ContRand, while each tuple needs to be replicated as 64 replicas in Random. As a result, ContRand and Random will reach the upper limit of processing power faster due to the presence of redundant computations, and thus have lower maximum throughput.

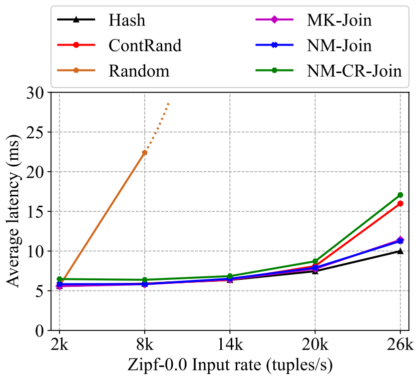
From the figure, we can also find that the maximum throughput of NM-Join is smaller than that of Random and ContRand when the skew of the input is very large. This is because the selectivity (i.e., the number of results produced) differs greatly among tuples when the skew is large, and thus the processing time differs greatly among tuples. As a result, the accuracy of using the number of stored tuples as a load measure decreases, and the actual load-balancing effect of NM-Join becomes worse. In this case, it can be seen from the figure that NM-CR-Join can achieve higher maximum throughput compared to NM-Join and ContRand. This is due to the fact that NM-CR-Join combines the advantages of NM-Join and ContRand to achieve better load balancing, which in turn effectively increases the maximum throughput of the system.

The experimental results confirm that for static datasets, NM-Join can achieve similar maximum throughput as MK-Join and outperform static load-balancing methods (ContRand and Random) when the skew of the input is not very large; while when the skew of the input is very large, our proposed NM-CR-Join can effectively improve the maximum throughput of the system and outperform ContRand.

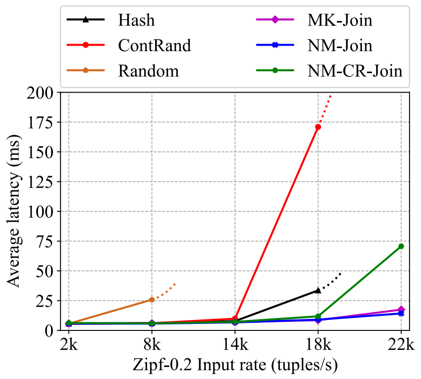
Latency We next study the latency of different methods with various datasets. Figure 8 shows the average latency of different methods under various input rates before reaching the maximum throughput in the case of various synthetic datasets, where the average latency is calculated in the same way as in [10]. The dotted line in the figure indicates that the corresponding method is close to reaching its maximum throughput, and the average latency of the method rises sharply toward infinity.

As shown in Fig. 8, NM-Join and MK-Join both have similar latencies for different synthetic datasets, which again confirms that NM-Join and MK-Join have the same adaptability to the input when the input distribution does not change.

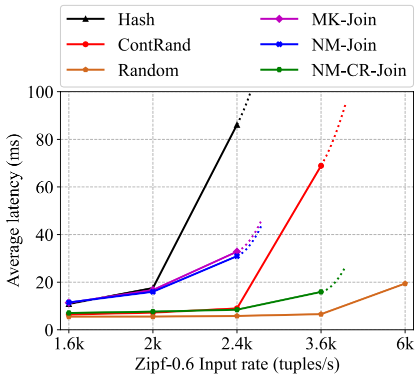
As can be seen in Fig. 8a, b, NM-Join has a lower latency compared to the static load-balancing methods (ContRand and Random) when the input skew is not very large. This is partly due to the better load-balancing effect of NM-Join compared to



(a) Average latency with Zipf-0.0.



(b) Average latency with Zipf-0.2.



(c) Average latency with Zipf-0.6.

Fig. 8 Average latency at different input rates with various datasets

ContRand, and partly due to the previously mentioned reason that there is redundant computation in ContRand and Random, which is equivalent to increasing the input rate of each Joiner node. In the synthetic datasets, the selectivity of all input tuples is low when the input skew is not vary large. When the selectivity of input tuples is low, the time to query the index occupies the majority of the total processing time of each tuple in the Joiner nodes [15], and thus the processing time of each tuple is approximately equal. According to queuing theory, when the processing time is the same, the total latency increases with the input rate, and thus NM-Join has lower latency than ContRand and Random when the input skew is not vary large.

From Fig. 8c, it can be found that NM-Join has higher latency compared to ContRand and Random when the skew of the input is very large. This is partly due to the previously mentioned reason that the accuracy of using the number of stored tuples as a load measure decreases when the skew of the input is very large, and partly due to the fact that when the skew of the input is very large, the selectivity of the partial high-frequency keys is high. When the selectivity of tuples is high, the processing time of each tuple is approximately proportional to the number of results generated [15], so sending tuples to multiple Joiner nodes for processing can reduce the processing time of tuples in each Joiner node, which in turn helps to reduce the average processing latency of all tuples. In this case, our proposed NM-CR-Join is able to obtain lower latency compared to NM-Join and ContRand due to its better load-balancing effect compared to NM-Join and ContRand, as well as the ability to distribute the tuples to multiple Joiner nodes for processing.

In summary, when the distribution of inputs does not change and the skew of inputs is not very large, NM-Join achieves similar performance to MK-Join and outperforms the static load-balancing methods ContRand and Random in terms of latency and throughput. In the latter section, we will focus on comparing the performance between NM-Join and MK-Join in the presence of changing input distributions.

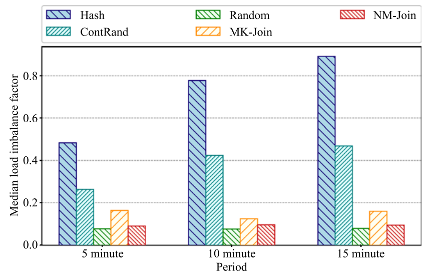
4.3 Dynamic input experiment

We next demonstrate the dynamic performance of the different methods when the distribution of the input streams changes. We conduct experiments with synthetic datasets as well as TPC-H datasets. For synthetic datasets, we keep the input rate and Zipf coefficient of the input streams constant, and we periodically change the distribution of keys of the input streams and test the performance of each method under different distribution change periods. We change the distribution of keys in the synthetic datasets by changing the way that the tuples are generated, specifically, we select the key k of a newly generated tuple in the following way:

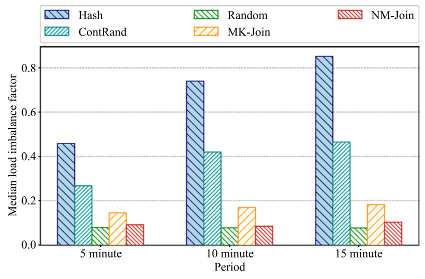
$$k = \left(zipf(K_{num}) + offset \times period\right) \bmod K_{num} \tag{6}$$

where K_{num} represents the number of unique keys, $zipf(K_{num})$ represents the random selection of a key from the K_{num} keys according to the Zipf distribution function, *offset* represents the offset of each change, and *period* represents the current number of distribution change periods. In our experiments, we set the Zipf coefficient of the synthetic datasets to 0.2. For the TPC-H datasets, we similarly change the distribution of the input streams periodically. In both experiments, we test the performance of the different methods for distribution change periods of 5, 10, and 15 min, respectively. Meanwhile, we set the input rate to 8k tuples per second and each experiment lasted 60 min.

Dynamic load-balancing effect Figure 9 shows the comparison of the median load imbalance factor of different methods for various periods. From the figure, we can see that NM-Join and MK-Join have lower load imbalance factors than Hash and ContRand for both synthetic and TPC-H datasets, which further confirms the effectiveness of NM-Join and MK-Join in adapting to the input. Meanwhile, it can also be observed from the figure that the load imbalance factor of NM-Join is lower than that of MK-Join for various distribution change periods. This is due to the fact that NM-Join adopts the sub-window technique, which enables NM-Join to react to the input distribution changes earlier and thus balance the load distribution of the system faster and obtain a lower load imbalance

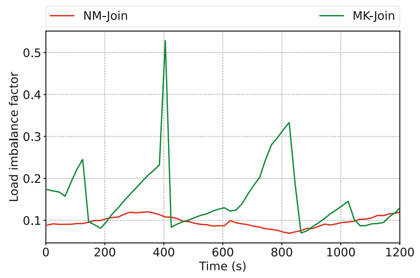


(a) Median load imbalance factor with synthetic datasets.

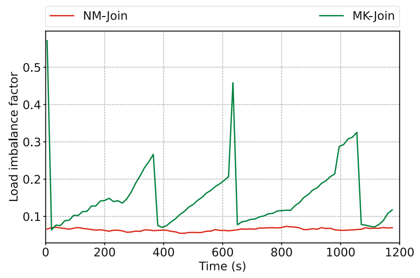


(b) Median load imbalance factor with TPC-H datasets.

Fig. 9 Median load imbalance factor under various distribution change periods



(a) Real-time load imbalance factor with synthetic datasets.

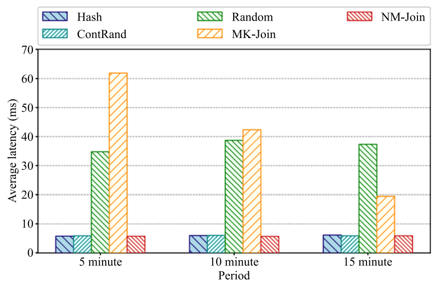


(b) Real-time load imbalance factor with TPC-H datasets.

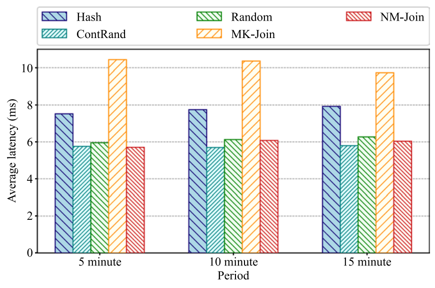
Fig. 10 Real-time load imbalance factor when the distribution change period is 10 min

factor. To verify this, we plot the real-time load imbalance factors of NM-Join and MK-Join at runtime with a 10-min distribution change period, respectively, as shown in Fig. 10. As can be seen from the figure, both for the synthetic dataset and the TPC-H dataset, there are multiple peaks in the curve of MK-Join, which means that MK-Join only performs load balancing to reduce the system load imbalance factor when the load-balancing factor within the whole window reaches the threshold, thus MK-Join usually has a higher load imbalance factor. In comparison, NM-Join is able to react more quickly to load changes and perform load balancing earlier due to the use of sub-windows, so it has a flatter curve and it usually has a lower load imbalance factor. The experimental results confirm that NM-Join has a better load-balancing effect compared to MK-Join when there are fluctuations in the distribution of input streams.

Load-balancing overhead Next, we compare the overhead required for load balancing between NM-Join and MK-Join. Figure 11 shows the average latency of the different methods under various distribution change periods. From the figure, it can be seen that both for the synthetic datasets and the TPC-H datasets, MK-Join has a high average delay compared to the other methods, which is due to the fact that MK-Join requires data migration to perform load balancing,

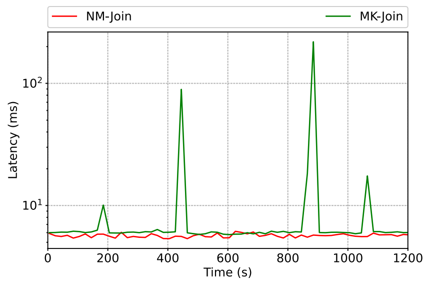


(a) Average latency with synthetic datasets.

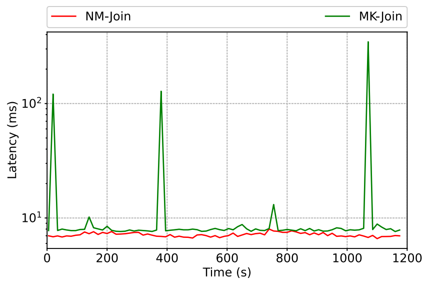


(b) Average latency with TPC-H datasets.

Fig. 11 Average latency under various distribution change periods



(a) Real-time latency with synthetic dataset.



(b) Real-time latency with TPC-H dataset.

Fig. 12 Real-time latency when the distribution change period is 10 min

and frequent migration introduces a lot of overhead to the system. In contrast, NM-Join has a lower average latency than MK-Join under various distribution change periods, and the latency of NM-Join is close to that of the static load-balancing method ContRand, which confirms that NM-Join has a low overhead in load balancing, which is close to that of the static load-balancing method. To further compare the overhead of NM-Join and MK-Join in load balancing, taking the 10-min distribution change period as an example, we plot the real-time processing latency of NM-Join and MK-Join with the synthetic and TPC-H datasets, as shown in Fig. 12. We can see that there are many peaks in the real-time latency curve of MK-Join, which corresponds to the peaks in Fig. 10, implying that the system is performing a migration. The migration causes a significant drop in transient performance and a spike in processing latency. In contrast, we can see that the real-time latency curve of NM-Join is relatively flat, which implies that our non-migrating load-balancing method has a very low overhead. The experimental results confirm that NM-Join has a lower load-balancing overhead compared to MK-Join.

In summary, the above experimental results confirm that NM-Join achieves lower latency and higher throughput than the static load-balancing methods ContRand and Random when the input skew is not very large, and does not introduce significant additional overhead when the input distribution changes. The experimental results also confirm that NM-Join achieves similar or even better adaptation to the input than MK-Join, while greatly reducing the overhead introduced for performing load balancing and thus achieving lower processing latency when adapting to input fluctuations.

The above experimental results show that our proposed non-migrating load-balanced distributed stream join system is well suited to perform join operations for streams that are not extremely skewed and whose distribution may change. Considering the adaptability and low load-balancing overhead of our proposed system, a possible application scenario well suited for this system is the real-time join processing of various data generated by IoT devices (e.g., sensors) in an edge computing environment, such as real-time join of data generated by two or more IoT devices monitoring the same object to obtain complete information about the monitored object [6]. In the above edge scenario, the computing resources of the edge computation nodes are limited, and the static load-balancing methods are not suitable for the edge computing scenario because of the redundant computational overheads mentioned above. The migration-based dynamic load-balancing methods are also not suitable for this scenario, which is partly because the limited computing and network resources of edge computation nodes can hardly afford the frequent state migration overhead, and partly because the system will suspend the processing of input data during state migration, and the unprocessed data will accumulate in the system. It can be handled in the cloud environment by using backpressure technology, but in the edge environment, this will lead to data loss with serious consequences because the data source of IoT devices cannot store a large amount of data. On the contrary, our proposed non-migrating load-balanced distributed stream join system does not incur redundant computational overhead when the input distribution does not change, and is able to perform load balancing with very low overhead when the input distribution changes. The system does not suspend the processing of input data for a long time, so that our proposed system is well suited to join operations on data streams in edge environments when the skew is not very large.

4.4 Load imbalance threshold determination

Finally, we study how to determine the load imbalance threshold of the system. Determining the threshold is a more subjective process, a too-large threshold can lead to poor load-balancing effects, while a too-small threshold may lead to too frequent load-balancing operations, which in turn affects the system performance. The users can determine the threshold by themselves according to their needs, and in our experiments, we determine the threshold by measuring the load-balancing effect of the system under different thresholds.

We conduct experiments with dynamic synthetic datasets to measure the load imbalance factor of NM-Join under different distribution change periods and each experiment lasted 30 min. The median load imbalance factors of NM-Join for different load imbalance thresholds and different distribution change periods are shown in Fig. 13. As can be seen from the figure, the median load imbalance factor increases roughly with the increase in the threshold, which is because the larger the threshold, the slower the system reacts to the change of the input distribution and thus the

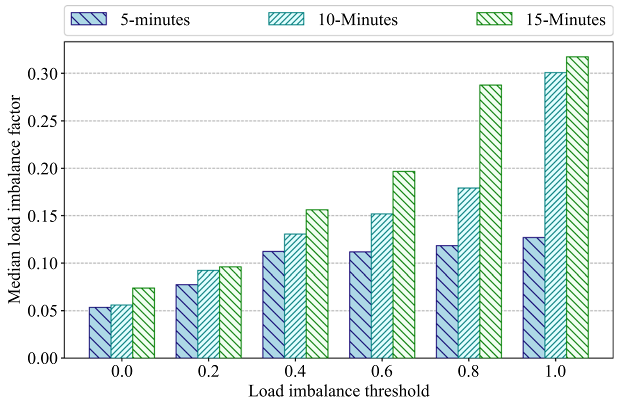


Fig. 13 Median load imbalance factor with different thresholds

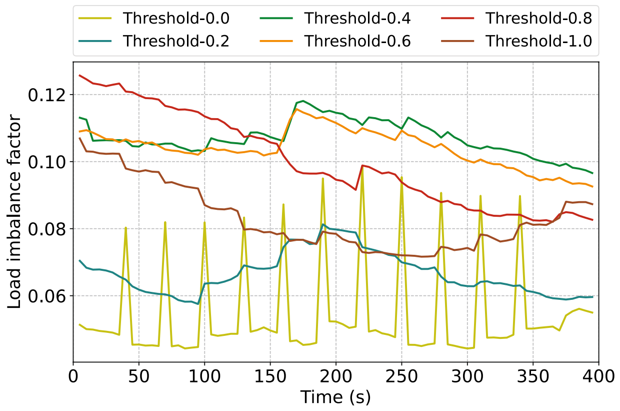


Fig. 14 Real-time load imbalance factor with different thresholds

less effective load balancing. Therefore, it seems that a very small load imbalance threshold (e.g., 0) should be chosen for a better load-balancing effect, however, we found in our experiments that a too-small threshold may lead to oscillations in the system load distribution, resulting in a large number of unnecessary load-balancing operations. Figure 14 shows the load distribution oscillation of NM-Join during a certain duration when the distribution change period is 5 min. From the figure, we can find that when the load imbalance threshold is 0.0, the load distribution of the system fluctuates drastically and frequently within a short period of time, which means that the system is performing load-balancing operations frequently and drastically. The reason for this phenomenon is that there is a certain randomness in the input distribution, which will lead to a slight load imbalance in the system. At this point, if the load imbalance threshold of the system is set too low, it may trigger unnecessary load-balancing operations frequently and cause oscillation of the system load distribution. During load distribution oscillation, the system will incur a large amount of unnecessary load-balancing overhead on the one hand (although the load-balancing overhead of our method is small, too frequent load-balancing operations will still result in a large number of tuples being replicated and forwarded, affecting the performance of the system), and on the other hand, the load-balancing effect of the system will be degraded during this period. Therefore, considering the effect of load balancing and the overhead of load balancing, we choose 0.2 as the load imbalance threshold in our experiments.

5 Related work

Stream join operators have recently attracted much research interest because of their great impact on distributed stream processing system performance. Earlier studies of stream join operators focus on stand-alone environments, and the main goal of these studies is to process stream join as fast as possible with limited memory [13, 16, 17]. But nowadays, with the development of business, the amount of data to be processed has increased drastically, so the traditional stand-alone join systems can no longer handle the huge amount of data.

To meet the performance requirements for processing large amounts of data, the stream join operators today are generally based on parallel/distributed stream processing systems [18]. Some of these parallel stream join algorithms are designed based on specific multicore environments, such as [19, 20], and these methods are not generalized and have limited processing power for large amounts of data.

For the more general environments, Teubner et al. [21] propose a non-centralized parallel join system called handshake join, which organizes all processing units into a linear structure, and the tuples in two streams flow through all processing units sequentially in opposite directions. Furthermore, Roy et al. [22] propose a method to reduce the delay of the handshake join system by fast-forwarding tuples. Handshake join can achieve load balancing of the system by flowing tuples between processing units, but since a tuple needs to flow through all processing units sequentially, handshake join has high processing latency as well as high network transmission volume.

Elseidy et al. [14] apply the join-matrix model used in batch processing systems [23] to the distributed stream processing system and design an adaptive online stream join operator. It organizes the processing units as a matrix, each side of which corresponds to an input stream. Each matrix cell represents a potential join output result. In addition, the method collects the statistics of the input streams during the system runtime and performs state migration to achieve optimal resource utilization. Similarly, Fang et al [24–26] design distributed stream join systems based on the join-matrix model and optimize the resource utilization. These methods can achieve load balancing by random partitioning, but the join-matrix model essentially replicates and forwards multiple copies of input tuples to multiple processing units for storing, thus consuming a large amount of additional network, computational and storage resources of the system, which significantly degrades the performance of the system and is memory inefficient.

Najafi et al. [8] propose a parallel stream join model called SplitJoin. SplitJoin introduces a top-down system structure where all tuples are broadcasted by the tuples distribution network to all processing units and each processing unit is responsible for storing only a portion of the tuples. The method is load-balanced by storing tuples in each processing unit in a polled manner, but it sends multiple copies of a tuple to all processing units to ensure completeness, resulting in significant additional system overhead.

Lin et al. [10] propose a join-biclique model, which organizes all processing units as a complete bipartite graph, where each side corresponds to a relation. This method proposes a strategy called ContRand to solve the load imbalance problem, which divides all processing units into groups, and each group consists of several processing units. Hash partitioning is used among groups, and random partitioning is used within each group. This method combines the features of random partitioning and content-sensitive partitioning to achieve system load balancing, but the method cannot handle the load imbalance between groups, and there are still a large number of tuple replications. Furthermore, Zhang et al. [27] solve the problem of load imbalance in the join-biclique model by using shuffle partitioning strategy for some keys, but the shuffle partitioning scheme also introduces additional network transmission overhead and computational resource overhead. In addition, Zhou et al. [12] use data migration to solve the load imbalance problem in the join-biclique model. They balance the load distribution of the system by migrating some of the tuples from the high-load processing units to the low-load processing units. This method is more flexible, but it is complex to implement and introduces migration overhead to the system, which has a significant impact on the transient performance of the system. In addition, Yuan et al. [28] solve the correctness problem in the join-biclique model by organizing all processing units into the tree structure. However, they do not focus on the problem of data skewing.

For the resource allocation in the general environment, Nikjoo et al. [29] propose a novel approach to joint optimal power allocation and user association techniques, which develops the solution to a mixed-integer programming framework and solves the goal function based on a Lagrangian convex optimization method. Mohajer et al. [30] propose a dynamic optimization model which optimizes carrier allocation and power utilization to meet the quality of service constraints and the highest energy efficiency levels. Mohajer et al. [31] also propose a dynamic max–min fairness-guaranteed optimization model which provides the essential coverage and capacity and minimizes the overall energy consumption of ultra-dense cellular heterogeneous networks. These methods can achieve good resource allocation, but they focus more on the energy consumption of the system than on the load balancing of the system. In addition, there has been a lot of research on the runtime adaptation of data stream processing systems [32]. For example, Lombardi et al. [33] use artificial neural networks to predict the state of the stream processing system and perform the scaling of the operator based on the predicted state. Cardellini et al. [34] perform the scaling of the operator in the stream processing system based on reinforcement learning techniques. However, these studies have not been designed for the stream join problem. Considering the special semantics of the stream join problem involving multiple data streams and the special definition of result completeness, these approaches cannot be directly applied to stream join systems.

6 Conclusion

In this paper, we proposed a novel load-balancing method for distributed stream window join. Our method is able to adapt to the inputs by monitoring the load distribution of the system during runtime and performing load-balancing operations when there is a load imbalance in the system. In addition, our method controls the replication and forwarding of tuples by setting the routing tables, and changes the load distribution of the system by directly changing the partition of the arrival store tuples when the system performs load balancing; at the same time, based on the characteristic of the completeness of the stream join problem, the completeness of the stream join results is guaranteed by replicating the arrival join tuples into multiple replicas and sending them to multiple partitions, thus realizing the load balancing of the system with very low overhead while guaranteeing the correct result. Theoretical analysis proves that our method can obtain a good load-balancing effect and guarantee the completeness of the results. Experimental results show that our method can achieve lower latency and higher throughput than the static load-balancing methods when the input skew is not very large, and does not introduce significant additional overhead when the input distribution changes. The experimental results also show that our method can achieve similar or even better adaptation to the input than the migration-based method, while greatly reducing the overhead introduced for performing load balancing and thus achieving lower processing latency when adapting to input fluctuations. In future work, we will further study the system load metrics so that we can better balance the system load distribution when the input skew is very large, and thus obtain lower latency and higher throughput.

Acknowledgements This work is supported by National Natural Science Foundation of China under Grant No. 62171155.

Author contributions All authors contributed to the study conception and design. Material preparation, data collection, and analysis were performed by QW, SC, and TL. The first draft of the manuscript was written by QW. The review and editing are mainly performed by DZ and ZZ. All authors read and approved the final manuscript.

Funding This study was funded by National Natural Science Foundation of China under Grant No. 62171155.

Data availability statement All of the material is owned by the authors and/or no permissions are required.

Declarations

Conflict of interest We declare that the authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Consent for publication All authors of this paper have read and approved the final version submitted and contents of this manuscript have not been copyrighted or published previously and are not under consideration for publication elsewhere.

References

1. Schranz C, Jeremias PM (2020) Deterministic time-series joins for asynchronous high-throughput data streams. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, vol 1, pp 1031–1034. https://doi.org/10.1109/ETFA46521.2020.9211958
2. Cheng Y, Hao Z, Cai R, Wen W (2018) Hpc2-ars: an architecture for real-time analytic of big data streams. In: 2018 IEEE International Conference on Web Services (ICWS), pp 319–322. https://doi.org/10.1109/ICWS.2018.00051
3. Ananthanarayanan R, Basker V, Das S, Gupta A, Jiang H, Qiu T, Reznichenko A, Ryabkov D, Singh M, Venkataraman S (2013) Photon: fault-tolerant and scalable joining of continuous data streams. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp 577–588. https://doi.org/10.1145/2463676.2465272
4. Gong Y, Zhang Q, Han X, Huang X (2017) Phrase-based hashtag recommendation for microblog posts. Sci China Inf Sci 60(1):1–13. https://doi.org/10.1007/s11432-015-0900-x
5. Shukla A, Chaturvedi S, Simmhan Y (2017) Riotbench: an iot benchmark for distributed stream processing systems. Concurr Comput Pract Exp 29(21):4257. https://doi.org/10.1002/cpe.4257
6. Mrozek D, Tokarz K, Pankowski D, Małysiak-Mrozek B (2019) A hopping umbrella for fuzzy joining data streams from IoT devices in the cloud and on the edge. IEEE Trans Fuzzy Syst 28(5):916–928. https://doi.org/10.1109/TFUZZ.2019.2955056
7. Zhang S, Liu C, Han Y, Li X (2018) Seamless integration of cloud and edge with a service-based approach. In: 2018 IEEE International Conference on Web Services (ICWS), pp 155–162. https://doi.org/10.1109/ICWS.2018.00027
8. Najafi M, Sadoghi M, Jacobsen H-A (2016) {SplitJoin}: a scalable, low-latency stream join architecture with adjustable ordering precision. In: 2016 USENIX Annual Technical Conference (USENIX ATC 16), pp 493–505. https://doi.org/10.5555/3026959.3027005
9. Gulisano V, Nikolakopoulos Y, Papatriantafilou M, Tsigas P (2016) Scalejoin: a deterministic, disjoint-parallel and skew-resilient stream join. IEEE Trans Big Data 7(2):299–312. https://doi.org/10.1109/BigData.2015.7363751
10. Lin Q, Ooi BC, Wang Z, Yu C (2015) Scalable distributed stream join processing. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp 811–825. https://doi.org/10.1145/2723372.2746485
11. Fang J-H, Zhao P-P, Liu A, Li Z-X, Zhao L (2019) Scalable and adaptive joins for trajectory data in distributed stream system. J Comput Sci Technol 34(4):747–761. https://doi.org/10.1007/s11390-019-1940-x
12. Zhou S, Zhang F, Chen H, Jin H, Zhou BB (2019) Fastjoin: a skewness-aware distributed stream join system. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE. pp 1042–1052. https://doi.org/10.1109/IPDPS.2019.00111
13. Kang J, Naughton JF, Viglas SD (2003) Evaluating window joins over unbounded streams. In: Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405). IEEE, pp 341–352. https://doi.org/10.1109/ICDE.2003.1260804
14. Elseidy M, Elguindy A, Vitorovic A, Koch C (2014) Scalable and adaptive online joins. VLDB. https://doi.org/10.14778/2732279.2732281
15. Shahvarani A, Jacobsen H-A (2020) Parallel index-based stream join on a multicore cpu. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp 2523–2537. https://doi.org/10.1145/3318464.3380576
16. Wilschut AN, Flokstra J, Apers PM (1995) Parallel evaluation of multi-join queries. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp 115–126. https://doi.org/10.1145/223784.223803
17. Viglas SD, Naughton JF, Burger J (2003) Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings 2003 VLDB Conference. Elsevier, pp 285–296. https://doi.org/10.1016/B978-012722442-8/50033-1
18. Zhang F, Chen H, Jin H (2019) Simois: a scalable distributed stream join system with skewed workloads. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp 176–185. https://doi.org/10.1109/ICDCS.2019.00026
19. Gedik B, Bordawekar RR, Yu PS (2009) Celljoin: a parallel stream join operator for the cell processor. VLDB J 18(2):501–519. https://doi.org/10.1007/s00778-008-0116-z

20. Buono D, De Matteis T, Mencagli G (2014) A high-throughput and low-latency parallelization of window-based stream joins on multicores. In: 2014 IEEE International Symposium on Parallel and Distributed Processing with Applications. IEEE, pp 117–126. https://doi.org/10.1109/ISPA.2014.24
21. Teubner J, Mueller R (2011) How soccer players would do stream joins. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp 625–636. https://doi.org/10.1145/1989323.1989389
22. Roy P, Teubner J, Gemulla R (2014) Low-latency handshake join. Proc VLDB Endowm 7(9):709–720. https://doi.org/10.14778/2732939.2732944
23. Okcan A, Riedewald M (2011) Processing theta-joins using mapreduce. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp 949–960. https://doi.org/10.1145/1989323.1989423
24. Fang J, Zhang R, Zhao Y, Zheng K, Zhou X, Zhou A (2019) A-dsp: an adaptive join algorithm for dynamic data stream on cloud system. IEEE Trans Knowl Data Eng 33(5):1861–1876. https://doi.org/10.1109/TKDE.2019.2947055
25. Fang J, Wang X, Zhang R, Zhou A (2016) Flexible and adaptive stream join algorithm. In: Asia-Pacific Web Conference. Springer, pp 3–16. https://doi.org/10.1007/978-3-319-45817-5_1
26. Fang J, Zhang R, Wang X, Zhou A (2017) Distributed stream join under workload variance. World Wide Web 20(5):1089–1110. https://doi.org/10.1007/s11280-017-0431-7
27. Zhang F, Chen H, Jin H (2019) Simois: a scalable distributed stream join system with skewed workloads. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp 176–185. https://doi.org/10.1109/ICDCS.2019.00026
28. Yuan J, Wang Y, Chen H, Jin H, Liu H (2021) Eunomia: efficiently eliminating abnormal results in distributed stream join systems. In: 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS). IEEE, pp 1–11. https://doi.org/10.1109/IWQOS52092.2021.9521286
29. Nikjoo F, Mirzaei A, Mohajer A (2018) A novel approach to efficient resource allocation in NOMA heterogeneous networks: multi-criteria green resource management. Appl Artif Intell 32(7–8):583–612. https://doi.org/10.1080/08839514.2018.1486132
30. Mohajer A, Sorouri F, Mirzaei A, Ziaeddini A, Rad KJ, Bavaghar M (2022) Energy-aware hierarchical resource management and backhaul traffic optimization in heterogeneous cellular networks. IEEE Syst J. https://doi.org/10.1109/JSYST.2022.3154162
31. Mohajer A, Daliri MS, Mirzaei A, Ziaeddini A, Nabipour M, Bavaghar M (2022) Heterogeneous computational resource allocation for NOMA: toward green mobile edge-computing systems. IEEE Trans Serv Comput. https://doi.org/10.1109/TSC.2022.3186099
32. Cardellini V, Lo Presti F, Nardelli M, Russo GR (2022) Runtime adaptation of data stream processing systems: the state of the art. ACM Comput Surv. https://doi.org/10.1145/3514496
33. Lombardi F, Aniello L, Bonomi S, Querzoni L (2017) Elastic symbiotic scaling of operators and resources in stream processing systems. IEEE Trans Parallel Distrib Syst 29(3):572–585. https://doi.org/10.1109/TPDS.2017.2762683
34. Cardellini V, Presti FL, Nardelli M, Russo GR (2018) Decentralized self-adaptation for elastic data stream processing. Fut Gen Comput Syst 87:171–185. https://doi.org/10.1016/j.future.2018.05.025

Authors and Affiliations

Qihang Wang[1] · Decheng Zuo[1] · Zhan Zhang[1] · Siyuan Chen[1] · Tianming Liu[1]

Qihang Wang
wangqihang@ftcl.hit.edu.cn

Decheng Zuo
zuodc@ftcl.hit.edu.cn

Siyuan Chen
chensiyuan@ftcl.hit.edu.cn

Tianming Liu
hit_ltm@yeah.net

1 Faculty of Computing, Harbin Institute of Technology, Harbin 150001, China