Distrib Parallel Databases (2010) 27: 211–254
DOI 10.1007/s10619-010-7062-7

Distributed stream join query processing with semijoins

Tri Minh Tran · Byung Suk Lee

Published online: 6 March 2010

Abstract This paper addresses the distributed stream processing of window-based multi-way join queries considering the semijoin as a key join operator. In distributed stream processing, data streams arriving at remote sites need to be shipped to the processing site for query execution. This typically introduces high communication overhead. Our observation is that semijoin, effective in reducing communication overhead in distributed database query processing, can be also effective in distributed stream query processing. The challenge, however, lies in the streaming nature of the tuples, as it requires continuous and incremental processing of an unbounded sequence of tuples instead of one-time processing of a set of stored tuples. This paper describes our comprehensive work done to address the challenge. Specifically, we first propose a distributed stream join processing model that handles the issue of network delays introduced from the shipment of data streams, and allows for efficient batch processing. Then, based on the model, we propose join algorithms in a multi-way join case: first, one-way join algorithms for different combinations of join placement and join method and, then, multi-way join algorithms assuming linear join ordering. Regarding the join method, two distributed join methods are introduced: (1) simple join, in which *full* tuples are forwarded to the query processing site and (2) semijoin-based join, in which *partial* tuples are forwarded. A semijoin-based join can be executed with different possible semijoin strategies which incur different communication overheads. We present a complete set of join algorithms considering all possible semijoin strategies, and propose an optimization algorithm. The join algorithms are executed continuously in an incremental manner as tuples arrive, and never

Communicated by Ling Liu.

T.M. Tran (✉) · B.S. Lee
Department of Computer Science, University of Vermont, Burlington, VT 05405, USA
e-mail: ttran@cems.uvm.edu

B.S. Lee
e-mail: bslee@cems.uvm.edu

ship tuples redundantly. The optimization algorithm constructs an efficient multi-way join plan by using a greedy heuristic which adds to the plan one stream with the minimum join execution cost in each step. Through extensive experiments, we conduct comparative studies of the performance among the proposed one-way join algorithms and the efficiency of the generated plan between the optimization algorithm based on the greedy heuristic and the exhaustive search, respectively.

Keywords Distributed data streams · Join queries · Semijoins

1 Introduction

The emergence of many applications on the data available as unbounded, continuous streams has drawn significant attention on data stream processing. Example applications include network packet traffic management [8, 18, 56], financial stock ticker analysis [4], sensor network data acquisition or processing [35], and telephone call monitoring in telecommunications [7]. In these applications, the data stream sources are typically *distributed* to different sites (or nodes) over the network (e.g., routers in the communication network, cluster heads in the sensor networks). Besides, the queries issued are typically *continuous queries* [9] against the data streams from the sources.

In this paper we focus on the problem of processing a join query over distributed data stream sources. We assume the join processing is window-based because processing a join over unbounded streams requires unbounded memory, which is impractical. Windows are commonly used in stream join processing [19, 21, 22, 24, 26, 28, 51]. The following examples are distributed stream join queries probable in real applications.

Example 1 (Network packet monitoring) Suppose we want to monitor the traffic of data packets passing through three routers in the last hour with the objective of finding the packets with the same destination address. The three routers are three stream sources, and the data packets going through the three routers make three data streams (S_1, S_2, and S_3). Each data packet contains a destination IP address *dest*. This monitoring task then can be specified as a distributed stream join query $S_1[1\text{ hour}] \bowtie_{S_1.dest=S_2.dest} S_2[1\text{ hour}] \bowtie_{S_2.dest=S_3.dest} S_3[1\text{ hour}]$.

Example 2 (News article filtering) Suppose we want to find recent articles on the same topic published by two news network services, say, Associated Press and Reuters, in a day. The two news network services are two stream sources, and the articles generated from the news network services are the news streams (S_A for Associated Press, and S_R for Reuters). Suppose each news article in a stream is tagged with a set of weighted keywords K. The monitoring task then can be specified as a distributed stream join query $S_A[1\text{ DAY}] \bowtie_{S_A.K=S_R.K} S_R[1\text{ DAY}]$ where $S_A.K = S_R.K$ is a set equality comparison. (This comparison may be generalized to an approximate matching.)

Example 3 (Building monitoring using sensor networks) Suppose we want to keep track of the temperature, humidity, and light intensity measured by sensors in the same room in the past 15 minutes. Assume that there are separate temperature sensors, humidity sensors, and light intensity sensors and that they periodically send readings to their respective sinks. That is, there are three sinks receiving a stream of temperature readings (S_T), a stream of humidity readings (S_H), and a stream of light intensity readings (S_L), respectively. Suppose a reading includes such fields as timestamp, sensor id (*sid*), room id (*rid*) in addition to the sensed value (i.e., temperature, humidity, or light intensity). This monitoring task then can be specified as a distributed stream join query S_T[15 minutes] $\bowtie_{S_T.rid=S_H.rid}$ S_H[15 minutes] $\bowtie_{S_H.rid=S_L.rid}$ S_L[15 minutes].

In the distributed stream environment, a data stream arriving at a remote site needs to be shipped to the processing site in order to generate an up-to-date result for the continuous query. The communication overhead of this shipment can be very high, and this brings a need for a technique to reduce the communication overhead while providing an up-to-date result. In this paper we present the *semijoin* as a viable technique to meet such a need.

The semijoin is well known in distributed databases as an effective operator for decreasing the communication overhead of a join query [12, 13, 16, 33, 50]. A semijoin from a relation R_1 to a relation R_2, denoted as $R_2 \ltimes R_1$, is equivalent to $\Pi_{Attr(R_2)}(R_2 \bowtie R_1)$, where $Attr(R_2)$ denotes all attributes of R_2. With a semijoin, the join between R_1 at site 1 and R_2 at site 2 can be computed using one of the following three equivalent *semijoin programs* [39]: $R_1 \bowtie (R_2 \ltimes R_1)$, $(R_1 \ltimes R_2) \bowtie R_2$, and $(R_1 \ltimes R_2) \bowtie (R_2 \ltimes R_1)$.

Computing a join using a semijoin program like these may incur lower communication overhead due to a relation size reduction resulting from the semijoin operation. Let us consider the semijoin program $R_1 \bowtie (R_2 \ltimes R_1)$ as an example. The semijoin $(R_2 \ltimes R_1)$ is processed by projecting R_1 on the join attributes, shipping the projection result to R_2's site and joining with R_2. The result of the semijoin is a reduced R_2 which contains only those tuples contributing to the final join with R_1. If the difference between the size of R_2 and the size of the reduced R_2 is larger than the size of the projection result of R_1, then using the semijoin incurs lower communication cost.

In our earlier work [47], we have proposed the *simple join* and the *semijoin-based* join as two distributed window-based join methods over data streams. The simple join is based on the idea of forwarding all tuples in a window from a remote site to the processing site for finding matching tuples. In contrast, the semijoin-based join is based on the idea of first shipping only the *partial tuples* (tuples consisting of only the join attributes) to the processing site and then shipping the full tuples only if matching tuples are found for the partial tuples at the processing site. In addition, the join methods support *incremental* processing of the streaming tuples, since tuples are processed continuously over data streams.

This earlier work has established the foundation for processing a distributed stream join utilizing the semijoin, but the work is very preliminary in its scope and the practical applicability. More specifically, it considers only two-way joins and only one of different possible semijoin programs, and is not concerned with the query op-

timization issue. Moreover, the proposed algorithms and their implementations leave much room for further improvement for correctness and efficiency, and the experiments are conducted using a simple linear model (i.e., transmission cost = transmission latency + transmission rate × transmitted data volume) to compute the network transmission cost.

The work presented in this paper is far more comprehensive and complete. Specifically, it considers multi-way joins and considers all possible semijoin programs for any given multi-way join query. This extended scope adds significant complexities to the query processing model and algorithms. The proposed algorithms work correctly despite network delays in shipping tuples and work more efficiently by processing tuples in a batch. Besides, the join algorithms incrementally ship only the tuples that have not been shipped to the processing site yet, while synchronously updating all windows stemming from the same window. Furthermore, it addresses the query optimization issue. Last but not least, the experiments are far more extensive and are conducted with both synthetic and real data sets, using virtual machines and a virtual network to measure the network transmission cost.

In this paper we first propose the processing model of a distributed stream join query. The proposed model is designed to reduce the adverse effects of network latencies on the correctness and efficiency of the join algorithm execution. That is, the join algorithms are resilient to the delays due to network latency (as long as the delays remain within a measurable bound) and also reduce the overheads due to network latency. We then propose distributed stream join algorithms under the query processing model. For this we first propose one-way join algorithms and, using them as the building blocks, propose a multi-way join algorithm.

A one-way join algorithm is characterized by the join method and the join placement. Given a one-way join "from a stream S_1 at N_1 to a stream S_2 at N_2", the join method[1] may be either the simple join or the semijoin-based join. Additionally, in the case of the semijoin-based join, the semijoin program may have either one semijoin operation or two semijoin operations (respectively called a *one-step semijoin program* and a *two-step semijoin program*). Thus, there are three possible one-way join methods in total. Regarding the join placement (i.e., determining where the output is produced), it may be at either the source node (N_1) or the destination node (N_2). Thus, combining the three join methods and the two join placements, six possible one-way join algorithms make a complete set.

A multi-way join algorithm is characterized by the join order as well as the join placement and the placed join method [14]. The join order specifies the order of the streams participating in the multi-way join. For stream joins, two commonly used ordering approaches are *linear ordering* [49] and *tree ordering* [24]. In this paper we propose an optimization algorithm to work with linear ordering, and then discuss using tree ordering instead in the multi-way join algorithm.

The query optimization is based on the cost formulas we have developed for the individual one-way join algorithms. The complexity of an exact optimization algorithm is exponential and, thus, we use a greedy approach to reduce the complexity to

[1] In our work, each join method is implemented as a nested loop join.

polynomial time. Specifically, the optimization algorithm constructs a join execution plan by adding one stream that has the minimum execution cost in each step. With the linear ordering employed, the optimization algorithm generates a join execution plan that includes a *set of join sequences*, with one sequence for each stream. In this paper, we also prove the equivalence of alternative plans executed in different semijoin programs, as this will show the correctness of the alternative plans.

The experiments are comprised of comparing the performances of the six one-way join algorithms and comparing the efficiencies of the join execution plans produced by the proposed greedy algorithm and an exhaustive search algorithm. Through extensive comparisons of the alternative one-way join algorithms, we make interesting observations and analyze the reasons for them given the parameter settings of the experiments. The experiments for comparing the two optimization algorithms show that the greedy algorithm is almost as effective as the exhaustive search algorithm in generating an efficient plan.

Main contributions made through this paper include proposing the model of processing a distributed window-based stream (multi-way) join query, introducing the notion of the semijoin as a key operator for the efficient execution of such a join query, developing join algorithms which may utilize the semijoin operator in query processing, presenting an optimization algorithm to find an efficient multi-way join execution plan and evaluating the proposed join algorithms and optimization algorithm through extensive experiments.

In the rest of the paper, we describe the query model of a distributed multi-way stream join in Sect. 2, the join algorithms in Sect. 3, the query optimization algorithm in Sect. 4, and the performance evaluation in Sect. 5. Then, we discuss the related work in Sect. 6 and conclude the paper in Sect. 7.

2 Query processing model

In this section we present the distributed window-based stream join processing model assumed in our work. In Sect. 2.1 we present some preliminary models in an evolving order from the stream model, the window model, and the window-based stream join model combining these two models. Then, in Sect. 2.2 we present the basic distributed window-based stream join model and enhance it to deal with the delays due to network latencies and to reduce the overheads of network latencies.

2.1 Preliminary models

Stream model

We define a stream S_i as a sequence of tuples, $s_{i_1}, s_{i_2}, \ldots$, arriving in order. We do not consider out-of-order arrival tuples—dealing with them is outside the scope of our work. Each tuple in the stream has a timestamp ts and a join attribute j as part of the schema. The stream rate of each stream is defined as the average number of tuples arriving in the stream per unit time (second).

Window model

As mentioned in Sect. 1, a window is used to restrict the number of tuples processed. A window may be either tuple-based or time-based as specified in the query. If tuple-based, the window size is the number of the most recent tuples in the window. If time-based, the window size is the time interval from the current time point to the past, and the number of tuples in the window depends on the stream rate. In our model we consider the time-based window only, as it is common in many applications of window stream join queries. (The discussion on how to adapt the proposed algorithms to tuple-based windows is presented in Sect. 3.3.) We denote a window W_i with size T_i as $W_i[T_i]$. At time t, a tuple is in the window $W_i[T_i]$ if and only if its timestamp is in the range $[t - T_i, t)$. We assume that all windows fit in main memory.

Window-based stream join model

A two-way window-based stream join [28] between a stream S_1 with a window W_1 and a stream S_2 with a window W_2 is computed in two symmetric *one-way joins* as follows. For the one-way join from S_1 to S_2, each time a new tuple arrives on S_1, we probe the window W_2 to find matching tuples and generate the corresponding join output tuples, and then update the window W_1 by inserting the new tuple and removing any expired tuples. For the one-way join from S_2 to S_1, the join computation is symmetric to the one-way join from S_1 to S_2. That is, each time a new tuple arrives on S_2, we probe W_1 to generate the join output tuples and then update W_2. In this paper, we denote a one-way join from S_i to S_j $(i \neq j)$ as $S_i \vec{\bowtie} S_j$. Thus, the two-way window-based stream join between S_1 and S_2 with the windows W_1 and W_2, respectively, is computed as two one-way joins $S_1 \vec{\bowtie} S_2$ and $S_2 \vec{\bowtie} S_1$.

Generalized from the two-way join, a multi-way join [24] among m $(m > 2)$ streams is computed as a sequence of $m - 1$ one-way joins from each stream S_k $(k = 1, 2, \ldots, m)$ to the other $m - 1$ streams. Without loss of generality, we consider only equijoins over a single join attribute. An extension to the case of a non-equijoin over multiple join attributes requires little modification of the proposed algorithms.

2.2 Distributed window-based stream join model

Consider a set of nodes (or sites) $N_1, N_2, \ldots, N_n$ connected through a communication network. We assume that there is only one stream at each node[2] and all local processing (e.g., selection, projection) has already been done at each node. We also assume that the time is synchronized among the nodes and, thus, assume the timestamps of tuples from different nodes are on the same clock. We denote a stream S_i at a node N_i as $S_i@N_i$ and, thus, denote a one-way join from S_i at N_i to S_j at N_j as $S_i@N_i \vec{\bowtie} S_j@N_j$. Here we call N_i the *source node* and N_j the *destination node*. Additionally, we refer to the node where the one-way join output is generated as the *processing* node.

[2]For the case that there are n (≥ 2) streams at a node N_i, N_i can be separated into k nodes $N_{i_1}, N_{i_2}, \ldots, N_{i_k}$ where the transmission cost among k nodes is zero and the transmission cost from other nodes to those k nodes are the same as that to N_i.

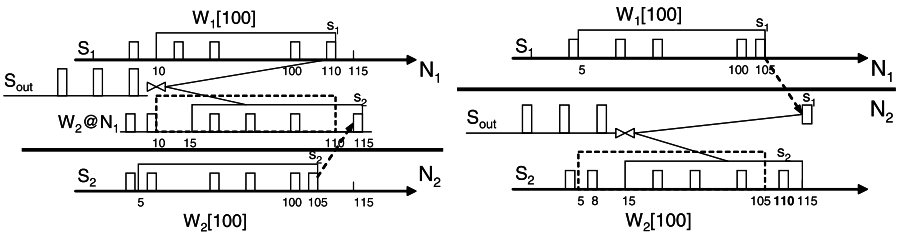
Resiliency to network latency

The key difference in processing a distributed stream join from processing a centralized stream join is that tuples needs to be shipped between the nodes before the processing can be done. The shipping introduces a delay due to network latency, and the delay may cause the join output to be incorrect. Let us clarify this point using two scenarios of processing a one-way stream join between two node, $S_1@N_1 \vec{\bowtie} S_2@N_2$. These two scenarios respectively correspond to the two simple join algorithms that will be described in Sect. 3.1.

In the first scenario, the join is processed at N_1 (see Fig. 1(a)). A tuple s_2 arrives at N_2 and is shipped to N_1 at time 105. Then, for the new tuple s_1 arriving at N_1 at time 110, we are supposed to probe the tuples in $W_2@N_1$ (a copy of W_2 maintained at N_1) in the time range $[10, 110)$. However, due to the latency of the shipment, s_2 will arrive at N_2 and be inserted into $W_2@N_1$ at time 115, delayed by 10. Thus, s_1 cannot be matched against s_2, although it should (since s_2 arrived at time 105 at N_2). As a result, the join output may be incorrect. To avoid this problem, the following constraint should be enforced on the probing of the window $W_2@N_1$.

> Constraint 1 [Delayed probe]: *If s_1 arrives at N_1 at time t and the delay of tuples s_2 from N_2 is δ, then $W_2@N_1$ should not be probed until the time $t + \delta$.*

In the second scenario, the join is processed at N_2 (see Fig. 1(b)). For the tuple s_1 arriving at N_1 and shipped to N_2 at time 105, we are supposed to probe W_2 for matching tuples in the time range $[5, 105)$. However, due to the latency of the shipment, s_1 arrives at N_2 at time 115, delayed by 10. By this time, a new tuple s_2 with the timestamp 110 has been inserted into W_2 and an expired tuple with the timestamp 8 has been removed from W_2. Thus, s_1 is matched against the new tuple s_2 with the timestamp 110, although it should not, and is not matched against the tuple with the timestamp 8, although it should. As a result, the join output may be incorrect. To avoid this problem, the expired tuple with the timestamp 8 should not be removed from W_2 when the new tuple with the timestamp 110 is inserted into W_2 and the tuple s_1 should be matched against those tuples s_2 in W_2 if and only if $5 \leq s_2.ts < 105$. As can be seen from this example, to ensure that a tuple s_1 is matched against tuples in W_2 correctly, the following two constraints should be enforced.



(a) Processed at N_1. (b) Processed at N_2.

(Assume the window size 100 on both streams. Assume the network latency is 10.)

Fig. 1 Examples of a distributed one-way stream join $S_1@N_1 \vec{\bowtie} S_2@N_2$

Constraint 2 [Delayed removal]: *If a new tuple is inserted into W_2 at time t and the delay is δ, then the expired tuples in W_2 at time t should not be removed until the time $t + \delta$.*

Constraint 3 [Limited-interval match]: *The tuple s_1 should be matched against only those tuples s_2 in W_2 whose timestamp ts is in the range $[s_1.ts - T_2, s_1.ts)$ (i.e., within the past T_2 from the timestamp of ts).*

The three constraints introduced here will be implemented in all join algorithms (both simple joins and semijoin-based joins) presented in Sect. 3.1.

One remaining question is how the value of δ is determined in the general case of a multi-way join. When extended to a multi-way join, given a join sequence a new arrival tuple at the node of the first stream in the join sequence may be shipped through all nodes of the streams in the sequence to probe for matching tuples and generate output tuples. In this case, the delay of shipping the new arrival tuple to the node of the last stream in the sequence will be the *accumulated* latency of shipping the tuple through all nodes in the sequence. Thus, δ needs to be the maximum accumulated delay among all possible join sequences. In our model, with this δ mechanism, a tuple s_1 can never be delayed more than δ throughout the sequence of joins to the last stream. So, by the time s_1 arrives at any node, all tuples in the window at the node are valid (because their invalidations are delayed by δ) and, therefore, $s1$ is correctly matched with the tuples in the window.

One concern regarding the $delta$ mechanism is that network delay can vary over time. However, it typically varies within a certain bound and, thus, the maximum value can be determined through measurements. Moreover, the value of δ can be updated to be current through either periodic or scheduled measurements. Sudden significant changes may still challenge the correctness of the model. This is a well-known hard problem for which there is no complete solution yet as far as we know, and is beyond the scope of this paper.[3]

Reduction of network latency overhead

The distributed stream join described so far is executed each time a new tuple arrives. Since each shipment incurs a delay due to network latency, the per-tuple processing may be infeasible in situations where the stream rate is high and/or the network latency is high. A practical solution is to execute the join for each *set of tuples* at a regular interval instead of each single tuple as they arrive. (This idea is similar to the lazy-evaluation multi-way join proposed in [24].) Let τ be the time interval of such join execution; that is, the join is executed every τ time units. During the time interval τ, each new arrival tuple s_i at the node N_i is kept in the input buffer until processed in a batch.

If there is no tuple arriving during the interval τ, then the buffer is empty and therefore no action is taken. This may happen if τ is smaller than the arrival interval

[3]Recently a new solution has been proposed in [34], which is geared to avoid such a risk. It is based on the notion of periodically sending a "heartbeat" or a punctuation. It, however, requires a different join processing model and needs an extension to handle *semijoin-based* and *multi-way* distributed joins. We leave this to the future work.

of two consecutive tuples. In general, when the value of τ is larger, the transmission overhead is smaller due to fewer shipments, but larger memory space is needed to buffer the new arrival tuples and a longer delay is incurred until all tuples in the larger buffer are shipped and the corresponding join result is generated. Thus, the value of τ should be specified by the system appropriately in consideration for the arrival interval of tuples, the limit on the buffer space, and the tolerable delay in generating a join output.

3 Distributed stream join algorithms

In this section, we present the one-way join algorithms over distributed data streams for all possible combinations of the join methods and the join placements. As mentioned in Sect. 1, one-way join algorithms are the basic building blocks of multi-way join processing. We describe the one-way join algorithms at a high level in Sect. 3.1 and efficient implementation details for the semijoin-based join algorithms in Sect. 3.2. Table 1 summarizes the notations used in this paper.

3.1 Distributed one-way stream join algorithms

Let us consider a one-way join $S_1@N_1 \vec{\bowtie} S_2@N_2$ executed every τ time units. (As mentioned in Sect. 2.2, we call N_1 the source node and N_2 the destination node.) The one-way join finds matching tuples from the window W_2 at N_2 for each batch of tuples (B_1) in the input buffer at N_1. It can be processed using either the simple join method or the semijoin-based join method. Moreover, for a given one-way join, there are two possible join placements depending on which node the one-way join output is generated, that is, either the source node N_1 or the destination node N_2.

Table 1 Notations used in this paper

Notation ($i = 1, 2$)	Meaning
τ	The execution interval of a join algorithm.
δ	The maximum accumulated delay among all possible join sequences.
S_i	The ith stream.
N_i	The node at which the stream S_i arrives.
T_i	The size of the window on the stream S_i.
B_i	The set of arrival tuples stored in the buffer at node N_i during the execution interval τ.
W_i	The set of tuples in the window on the stream S_i. (W_i also denotes the window itself.)
V_i	The set of partial tuples resulting from the projection of tuples in W_i on the join attribute.
K_i	The set of partial tuples resulting from the projection of tuples in B_i on the join attribute.
B_i'	The set of tuples reduced from B_i using semijoin.
W_i'	The set of tuples reduced from W_i using semijoin.
V_i'	The set of tuples reduced from V_i using semijoin.
K_i'	The set of tuples reduced from K_i using semijoin.

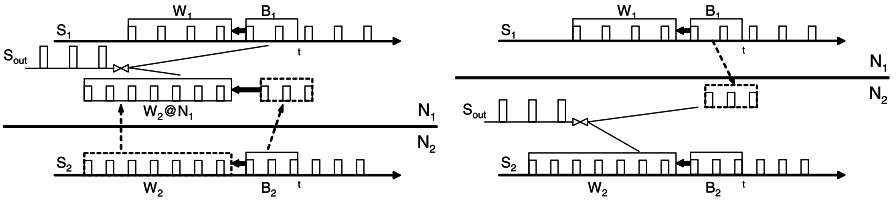
The combination of the join placements and the join methods makes different one-way join algorithms. We describe the simple join algorithms in Sect. 3.1.1 and the semijoin-based join algorithms in Sect. 3.1.2.

3.1.1 Simple join

The simple join is based on the idea of shipping *all full tuples* from a remote node to the processing node. There are two cases depending on whether the processing node is the source node or the destination node. Figure 2 illustrates the two cases: *SP-S* (simple join at the source node) and *SP-D* (simple join at the destination node). Note first that the set of tuples in the window that is "probed" during one-way join (W_2 in this case) must be at the processing node. For this purpose, in SP-S, the tuples that are supposed to be in the window W_2 at N_2 are shipped to the processing (source) node N_1 and maintained there (denoted as $W_2@N_1$) to be probed with the tuples in B_1. W_2 is not maintained at N_2, so in effect W_2 is "exported" to N_1. In contrast, in SP-D, only the tuples in the buffer B_1 (at N_1) need to be shipped to the processing (destination) node N_2 for join with the window W_2 there.

Algorithms 1 and 2 show the specific processing steps of the two cases. Each algorithm has two independent parts, and each part is triggered at a regular interval τ. Let us refer to these two parts as the N_1 (source) part and the N_2 (destination) part of the algorithm, respectively. The statements in each part are the actions (triggered at the regular interval of τ).

In these algorithms, the window update operation and the join operation need to be slightly modified as follows to satisfy the constraints mentioned in Sect. 2.2:



(a) Processed at the source node (SP-S). (b) Processed at the destination node (SP-D).

Fig. 2 Simple join processing of a one-way join $S_1@N_1 \vec{\bowtie} S_2@N_2$

Algorithm 1 One-way simple join processing at the source node N_1 (SP-S)

```
1 At each time point t at the interval of τ at N1 begin
2 |  N1 updates W1 with B1 semi-delayed by δ;
3 |  N1 performs a T2-limit join between B1 and W2@N1 delayed by δ;
4 end
5 At each time point t at the interval of τ at N2 begin
6 |  N2 ships B2 to N1;
7 |  N1 receives B2, and then updates W2@N1 with B2 semi-delayed by δ;
8 end
```

Algorithm 2 One-way simple join processing at the destination node N_2 (SP-D)

```
1 At each time point t at the interval of τ at N1 begin
2 |   N1 updates W1 with B1 semi-delayed by δ;
3 |   N1 ships B1 to N2;
4 |   N2 receives B1, and then performs a T2-limit join between B1 and W2;
5 end
6 At each time point t at the interval of τ at N2 begin
7 |   N2 updates W2 with B2 semi-delayed by δ;
8 end
```

- To satisfy Constraint 1, the join between B_1 and $W_2@N_1$ (in Line 3 of Algorithms 1) is performed *delayed by* δ (i.e., at time $t + \delta$), as N_1 needs to wait until the tuples in B_2, shipped from N_2, arrive at N_1 and are available to update $W_2@N_1$.
- To satisfy Constraint 2, updating a window W_i with the tuples in B_i is done by delaying the removal of any expired tuples in W_i by δ while inserting the tuples in B_i into W_i without delay. We say in this case the window update is *semi-delayed* by δ.
- To satisfy Constraint 3, joining B_i and W_j should check, for each tuple s_i in B_i, whether the timestamp of a matching tuple in W_j is in the range of $[s_i.ts - T_j, s_i.ts)$. In other words, the join considers only those tuples within the limit of T_j from the timestamp of s_i. We refer to this join as the T_j*-limit join*.

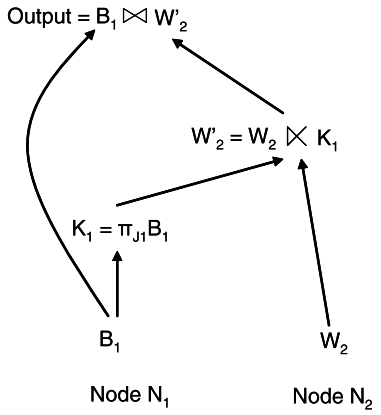
These modified window update and join operations are generic to all join algorithms and, thus, used in the semijoin-based join algorithms as well.

3.1.2 Semijoin-based join

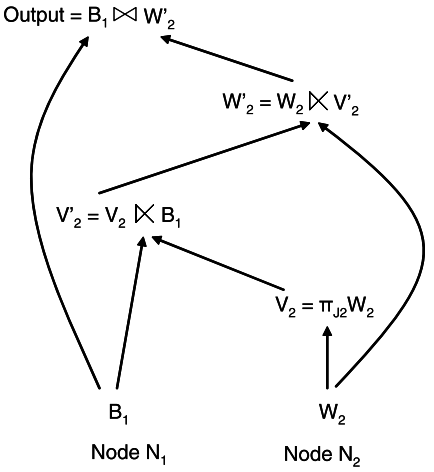
The semijoin-based join is based on the idea of forwarding *partial tuples* from a remote node to the processing node. A partial tuple contains only the join attribute extracted from a full tuple. It is shipped to a remote node for probing the window to find matching tuples, and the full tuple is shipped only if matching tuples are found. We describe the semijoin-based join steps at a high level here, while deferring some implementation details to Sect. 3.2.

Given a one-way join $S_1@N_1 \vec{\bowtie} S_2@N_2$, there are four different semijoin programs possible. Specifically, for each possible join placement (i.e., at N_1 or N_2), the join can be done using either a *one-step reduction* semijoin program or a *two-step reduction* semijoin program. These two programs differ in the number of semijoin operations performed in the semijoin-based join.

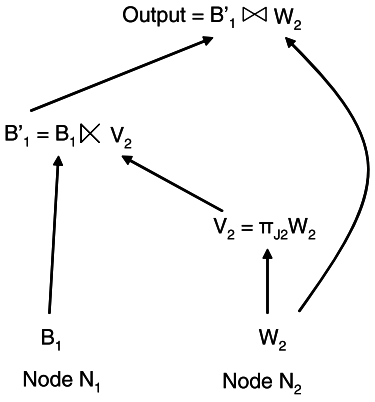
Figure 3 illustrates the four alternative semijoin programs. In Figs. 3(a) and (b), the source node (N_1) is the processing node and, thus, to generate the join output at N_1, we ship W_2' (the reduced W_2, i.e., W_2 tuples matching the tuples in B_1) to N_1. The reduction of W_2 is done using either a one-step semijoin program (Fig. 3(a)) or two-step semijoin program (Fig. 3(b)). In Figs. 3(c) and (d), the destination node (N_2) is the processing node and, thus, we ship B_1' (the reduced B_1, i.e., B_1 tuples matching the tuples in W_2) to N_2. The reduction of B_1 is done using either a one-step semijoin program (Fig. 3(c)) or a two-step semijoin program (Fig. 3(d)). Note that there is no



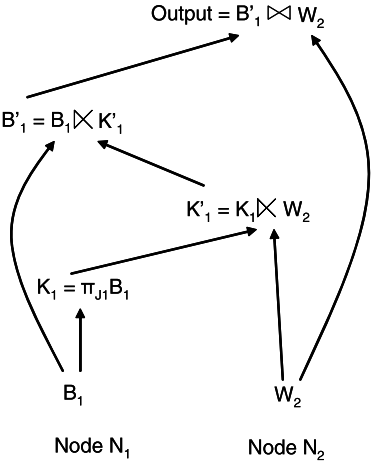
(a) Join processing at the source node (N_1) with one-step reduction.



(b) Join processing at the source node (N_1) with two-step reduction.



(c) Join processing at the destination node (N_2) with one-step reduction.



(d) Join processing at the destination node (N_2) with two-step reduction.

Fig. 3 The four alternative semijoin programs for one-way join $S_1@N_1 \vec{\bowtie} S_2@N_2$

need for more than two steps of semijoin because an additional semijoin step does not reduce the operands any further.

Based on the four semijoin programs, we develop algorithms that execute the semijoin programs continuously as tuples arrive. Two key ideas are employed in the design of the algorithms, both for reducing the communication overhead. One idea, mentioned above, is to ship partial tuples instead of full tuples, that is, ship only the join attributes to the processing node and ship the corresponding full tuples only if the

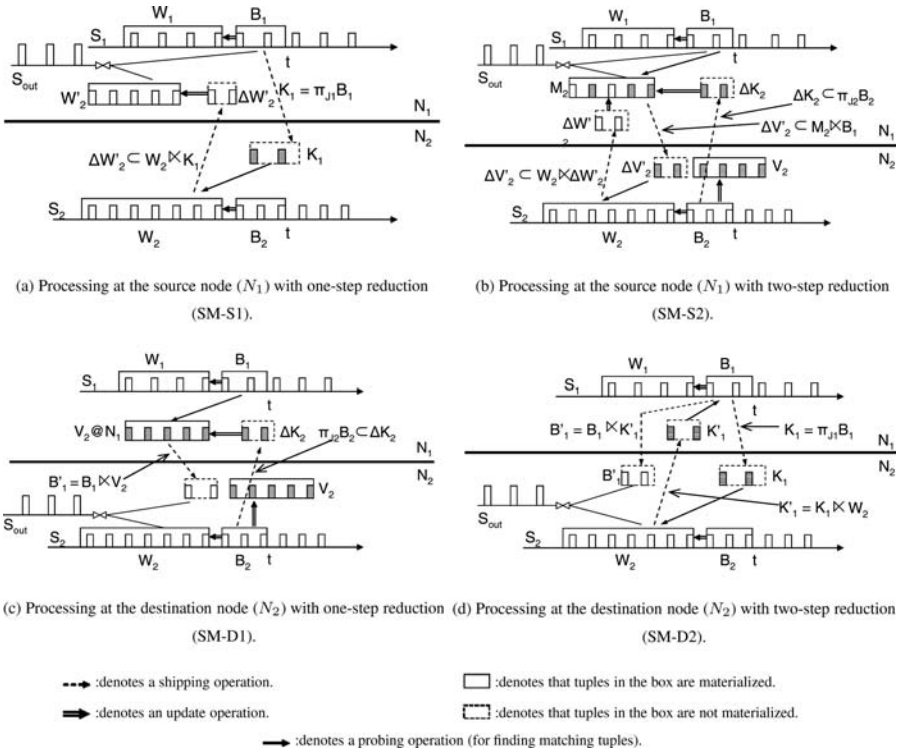


Fig. 4 Continuous processing of the semijoin-based one-way join $S_1 @ N_1 \vec{\bowtie} S_2 @ N_2$

partial tuples have matching tuples in the window at the processing node. The other idea is to ship tuples only once, that is, only if they have not been shipped already. Figure 4 illustrates how the four semijoin programs are executed continuously over the input streams S_1 and S_2.

Algorithms 3 through 6 respectively outline the steps of executing the four semijoin programs shown in Fig. 3 continuously as shown in Fig. 4. We now describe each of the four algorithms.

Algorithm 3: one-step semijoin at the source node (SM-S1)

In the N_1 (source) part of the algorithm, Line 4 corresponds to the semijoin $W_2 \ltimes K_1$ in Fig. 3(a). The semijoin generates W_2', which is then shipped to N_1 for the join with B_1. As mentioned earlier, we reduce the communication cost by avoiding shipping the same tuple more than once. (We call this semijoin the *non-redundant semijoin* (details in Sect. 3.2.1).) Thus, the non-redundant semijoin (Line 4) returns only the W_2 tuples that have not been shipped to N_1, which we denote as $\Delta W_2'$. (The same technique is used in the SM-S2 algorithm, to be described below, as well.) Then, once $\Delta W_2'$ is shipped to N_1 (Line 5), we augment W_2' with $\Delta W_2'$ and perform the join between B_1 and W_2' (Line 6).

Algorithm 3 One-way join processing at the source node N_1 using a one-step semijoin program (SM-S1)

```
1  At each time point t at the interval of τ at N1  begin
2      N1 updates W1 with B1 semi-delayed by δ;
3      N1 projects B1 on the join attribute and ships the resulting set of partial tuples (K1)
       to N2;
4      N2 receives K1 and then performs a non-redundant semijoin (see Algorithm 7 in
       Sect. 3.2.1) from K1 to W2 (i.e., W2 ⋉ K1) to obtain the result ΔW2';
5      N2 ships ΔW2' to N1;
6      N1 receives ΔW2', inserts tuples in ΔW2' into W2', and then performs a T2-limit join
       between B1 and W2' and outputs the result;
7  end
8  At each time point t at the interval of τ at N2  begin
9      N2 updates W2 with B2 semi-delayed by δ;
10     N2 sends an update message to N1 for those tuples removed from W2 in Line 9;
11     N1 receives the update message and removes corresponding tuples from W2'@N1
       (see Algorithm 8 in Sect. 3.2.2 for details);
12 end
```

In the N_2 (destination) part of the algorithm, since $W_2'@N_1$ is a subset of W_2, when any expired tuples are removed as a result of updating W_2 (Line 9), they should be removed from $W_2'@N_1$ as well if they are in it (Line 11). An efficient implementation of this synchronous update is discussed in Sect. 3.2.2.

Algorithm 4: two-step semijoin at the source node (SM-S2)

In this algorithm, we maintain a window of "mixed" tuples, denoted as M_2, at N_1. This window contains a mixture of partial tuples from V_2 ($= \Pi_{J_2} W_2$ in Fig. 3(b)) and full tuples from W_2' ($= W_2 \ltimes V_2'$ in Fig. 3(b)).

In the N_1 part, for each set of tuples in B_1, the first semijoin from B_1 to M_2 (i.e., $M_2 \ltimes B_1$, corresponding to $V_2 \ltimes B_1$ in Fig. 3(b)) and the final join between B_1 and M_2 (i.e., $B_1 \bowtie M_2$, corresponding to $B_1 \bowtie W_2'$ in Fig. 3(b)) are performed (together for efficiency's sake) (Line 3). Note that these two operations are executed delayed by δ in order to satisfy the Constraint 1 (see Sect. 2.2).

Tuples in M_2 initially are partial tuples coming from the initial V_2, but after the first semijoin from the first batch of B_1 (i.e., $M_2 \ltimes B_1$, producing the initial V_2'), some of the partial tuples in M_2 are replaced by their corresponding full tuples. Thus, from the second batch of B_1 onward, the first semijoin produces fewer partial tuples than those in the initial V_2', as they do not include those tuples that have already been replaced by full tuples. We thus denote the resulting set of partial tuples produced by $M_2 \ltimes B_1$ as $\Delta V_2'$ (Line 3). In addition, in Line 3, we keep in a temporary buffer (*Temp*) the tuples of B_1 for which the matching partial tuples in $\Delta V_2'$ have been found. This is to avoid a redundant scanning of the tuples in B_1 for the join execution later in Line 8. Line 5 corresponds to the second semijoin (i.e., $W_2 \ltimes V_2'$) shown in Fig. 3(b)), but we use $\Delta V_2'$ instead of V_2' to reduce the communication overhead. Besides, this semijoin is a non-redundant semijoin which finds those matching tuples in W_2 that have not been shipped to N_1 yet, that is, $\Delta W_2'$.

Algorithm 4 One-way join processing at the source node N_1 using a two-step semijoin program (SM-S2)

```
1  At each time point t at the interval of τ at N1  begin
2      N1 updates W1 with B1 semi-delayed by δ;
3      N1 performs a semijoin from B1 to partial tuples in M2 (i.e., M2 ⋉ B1) delayed by δ
       to obtain the result ΔV2' and saves the matching B1-tuples in a temporary buffer
       Temp, and at the same time performs a T2-limit join between B1 and full tuples in
       M2 and outputs the result;
4      N1 ships ΔV2' to N2;
5      N2 receives ΔV2' and performs a non-redundant semijoin from ΔV2' to W2 (i.e.,
       W2 ⋉ ΔV2') to obtain the result ΔW2';
6      N2 ships ΔW2' to N1;
7      N1 receives ΔW2' and updates M2 by replacing the matching partial tuples by the
       full tuples in ΔW2';
8      N1 performs a T2-limit join between Temp and ΔW2' and outputs the result;
9  end
10 At each time point t at the interval of τ at N2  begin
11     N2 updates W2 with B2 and then updates V2 with the partial tuples of B2, both
       semi-delayed by δ;
12     N2 sends an update message to N1;
13     N1 receives the update message and updates M2 according to the updates made to
       W2 and V2 (see Algorithm 9 in Sect. 3.2.2 for details);
14 end
```

In the N_2 part, when W_2 is updated by inserting tuples from B_2 and removing expired tuples, V_2 is updated accordingly as it is a projection of W_2 (Line 11). Furthermore, M_2 (at N_1) should be updated as well since it is a mixture of full tuples from W_2 and partial tuples from V_2 (Lines 13). An efficient implementation of these synchronous updates is presented in Sect. 3.2.2.

Algorithm 5: one-step semijoin at the destination node (SM-D1)

In the N_1 part, Line 3 corresponds to the semijoin $B_1 \ltimes V_2$ in Fig. 3(c). For this semijoin, N_1 keeps a local copy of V_2 resulting from the projection of W_2. This semijoin generates B_1', the reduced B_1, which is then shipped to N_2 for the join with W_2. As in SM-S2, the semijoin is executed delayed by δ in order to satisfy the Constraint 1 (see Sect. 2.2).

In the N_2 part, once W_2 is updated with the new arrival tuples and the expired tuples, then V_2 and $V_2@N_1$ need to be updated accordingly (Lines 8 and 10). An efficient implementation of these synchronous updates is presented in Sect. 3.2.2.

Algorithm 6: two-step semijoin at the destination node (SM-D2)

In the N_1 part, K_1 is the set of partial tuples of B_1 (i.e., corresponding to $\Pi_{J_1} B_1$ in Fig. 3(d)). Line 4 performs the first semijoin (i.e., $K_1 \ltimes W_2$) for reducing K_1. K_1', the reduced K_1, consists of the partial tuples that have matching tuples in W_2. Line 6 performs the second semijoin (i.e., $B_1 \ltimes K_1'$) for reducing B_1. Then, Line 8 performs

Algorithm 5 One-way join processing at the destination node N_2 using a one-step semijoin program (SM-D1)

1 At each time point t in the interval of τ at N_1 **begin**
2 N_1 updates W_1 with B_1 semi-delayed by δ;
3 N_1 performs a semijoin from $V_2@N_1$ to B_1 (i.e., $B_1 \ltimes V_2$) delayed by δ to obtain the result B_1';
4 N_1 ships B_1' to N_2;
5 N_2 receives B_1', and performs a T_2-limit join between B_1' and W_2 and outputs the result;
6 **end**
7 At each time point t in the interval of τ at N_2 **begin**
8 N_2 updates W_2 with B_2 and then updates V_2 with partial tuples of B_2, both semi-delayed by δ;
9 N_2 sends an update message to N_1;
10 N_1 receives the update message and updates $V_2@N_1$ (see Sect. 3.2.2 for details);
11 **end**

Algorithm 6 One-way join processing at the destination node N_2 using a two-step semijoin program (SM-D2)

1 At each time point t at the interval of τ at N_1 **begin**
2 N_1 updates W_1 with B_1 semi-delayed by δ;
3 N_1 projects B_1 on the join attribute and ships the resulting set of partial tuples (K_1) to N_2;
4 N_2 receives K_1 and performs a semijoin from W_2 to K_1 (i.e., $K_1 \ltimes W_2$) to obtain the result K_1' and saves the matching full tuples in W_2 in a temporary buffer *Temp*;
5 N_2 ships K_1' to N_1;
6 N_1 receives K_1' and performs a semijoin from K_1' to B_1 (i.e., $B_1 \ltimes K_1'$) to obtain the result B_1';
7 N_1 ships B_1' to N_2;
8 N_2 receives B_1' and performs a T_2-limit join between B_1' and *Temp* and outputs the result;
9 **end**
10 At each time point t at the interval of τ at N_2 **begin**
11 N_2 updates W_2 with B_2 semi-delayed by δ;
12 **end**

the final join (i.e., $B_1' \bowtie W_2$) to generate the output. As in SM-S2, we store the full matching tuples of W_2 in a temporary buffer (*Temp*) and use it in place of W_2 to avoid scanning the W_2 tuples more than once.

3.2 Implementation details of the semijoin-based one-way stream join algorithms

In this subsection we present the implementation details of the non-redundant semijoin (used in Algorithms 3 and 4) and the synchronous window updates (used in

Algorithms 3, 4, and 5). All implementation ideas employed are to reduce the communication overhead incurred to perform the operations continuously.

3.2.1 Non-redundant semijoin

The non-redundant semijoin (in Line 4 of Algorithm 3 and Line 5 of Algorithm 4) ships only the tuples that have not been shipped yet. This is done by marking a bit associated with each tuple to record the shipment status of the tuple. Algorithm 7 outlines the steps.

3.2.2 Synchronous window updates

There are three cases of updating windows between the source node and the destination node to synchronize the window contents. In this subsection, we explain the details of each synchronization.

Updating $W_2'@N_1$ in SM-S1

In Line 11 of Algorithm 3, whenever expired tuples are removed from W_2, the same tuples must be removed from $W_2'@N_1$ as well if they exist in $W_2'@N_1$. Algorithm 8 shows the steps of this update. In order to avoid shipping all expired tuples to N_1, we organize $W_2'@N_1$ as a *FIFO queue* of tuples (ordered in an increasing order of timestamp) and have N_2 ship only the number (k) of expired tuples instead of the tuples themselves; then, N_1 only needs to remove the first k tuples from the queue. The expired tuples need to be counted only if they have been shipped to N_1. For this, we use a *bit map*, where each bit b denotes whether the associated tuple has

Algorithm 7 Semijoin operation modified to reduce the network overhead

```
1  Operator: Non-redundant semijoin (A1, A2);
   Input: A1: a set of partial tuples; A2: a set of tuples;
   Output: Aout: a set of tuples reduced from A2 through a semijoin A2 ⋉ A1;
2  begin
3      for each tuple a1 in A1 do
4          for each tuple a2 in A2 do
               // If a2 matches a1 but has never been sent, then put in
                  Aout and mark it as "sent" (i.e., set the bit b to 1).
5              if (a1.J = a2.J and a2.b == 0) then
6                  Insert a2 into Aout;
7                  a2.b = 1;
8              end
9          end
10     end
11 end
```

Algorithm 8 Updating $W_2'@N_1$ in SM-S1

```
1  begin
2      k = 0;
3      for each expired tuple s2 do
4          if the bit b associated with s2 equals 1 then
5              k = k + 1;
6          end
7      end
8      N2 ships the value k to N1;
9      N1 receives the value k and removes the first k tuples from the FIFO queue
       representing W2'@N1;
10 end
```

been shipped ($b = 1$) to N_1 or not ($b = 0$), and count the expired tuples whose bits are 1.

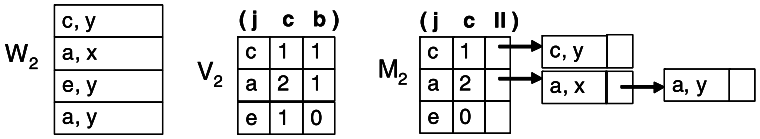
Updating V_2 and M_2 in SM-S2

As mentioned in Algorithm 4, M_2 is a mixture of full tuples from W_2' (the reduced W_2) and partial tuples from V_2 (the result of projecting W_2 on the join attribute). Thus, once W_2 is updated with the full tuples in B_2, V_2 needs to be updated accordingly with the partial tuples of B_2 (Line 11) and M_2 needs to be updated as well according to the updates of W_2 and V_2 (Line 13).

Since V_2 is a set of partial tuples resulting from the projection of W_2 on the join attribute, the join attribute values of the partial tuples are all distinct. Moreover, any partial tuple in M_2 that has the same join attribute value as any full tuples in W_1 is replaced by the corresponding full tuples from W_2. Based on these observations, we organize V_2 as an array where the join attribute is the unique key, and organize M_2 as a hash table where the hashing key is the join attribute and the hashed value is the set of full tuples with the join attribute value.

Figure 5 shows the array structure of V_2 and the hash table structure of M_2 using an example. Each entry in the array representing V_2 has three fields: a distinct join attribute value (j), the number (c) of full tuples in W_2 with the same join attribute value j, and a bit (b) indicating whether the full tuples from W_2 with the same join attribute value j have been shipped ($b = 1$) to N_1 or not ($b = 0$). Each entry in the hash table representing M_2 has three fields as well: a distinct join attribute value (j), the number (c) of full tuples with the same join attribute value j, and a pointer to the linked list (ll) representing the set of those full tuples. Note that the extra fields c, b, and ll are specific to the implementation and are not part of the tuple schema. In the remainder of this section, we use the notations V_2 and M_2 respectively to refer to the array and the hash table representing them.

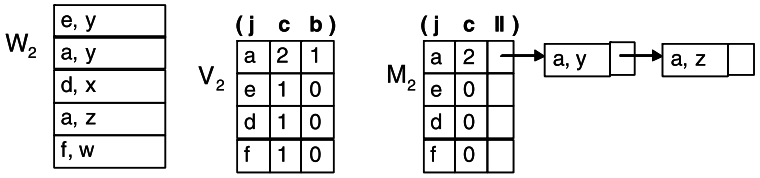
The update algorithm needs to synchronize the distinct join attribute values of M_2 with those of V_2 and W_2. Specifically, it is done as follows. First, for any tuple expiring from W_2, we reduce the c value of the matching (i.e., with the same j value) entry in V_2 by 1 and, if the resulting c equals zero, then remove the entry from V_2 and



(a) Before updating W_2.

Update on W_2: Insert three tuples $\langle d, x\rangle$, $\langle a, z\rangle$, $\langle f, w\rangle$, and remove two tuples $\langle c, y\rangle$ and $\langle a, x\rangle$.
Shipment: N_2 ships $\mathbf{v}_J = \langle d, f\rangle$, $\mathbf{v}_C = \langle 0, 2, 1, 1, 1\rangle$ (for $j = c, a, e, d, f$), and $\mathbf{s}_F = \{\langle a, z\rangle\}$ to N_1.

(b) Updating W_2 and shipping update data.



(c) After updating W_2.

Fig. 5 Illustration of updating V_2 and M_2 synchronously with updating W_2

remove the same entry from M_2 as well. Second, for any tuple inserted into W_2, we need to check if the b field of the matching entry in V_2 is equal to 1 and, if so, insert the same tuple in M_2. We handle this synchronization by maintaining the order of the join attribute values the same between M_2 and V_2 and shipping two vectors and one set to N_1. The two vectors are the *distinct join attribute (DJA) value vector* $\mathbf{v}_J$ and the *distinct join attribute (DJA) count vector* $\mathbf{v}_C$. The former ($\mathbf{v}_J$) is the vector of the distinct join attribute values in the new entries added to V_2. The latter ($\mathbf{v}_C$) is the vector of the number (or count) of tuples in W_2 for each distinct join attribute value in V_2, including both the old entries removed and the new entries added. The set is denoted as $\mathbf{s}_F$. It contains the set of full tuples to be inserted into M_2 as well among those inserted into W_2.

Example 4 Figure 5 illustrates synchronous updates on W_2, V_2, and M_2. Suppose three new tuples $\langle d, x\rangle$, $\langle a, z\rangle$, and $\langle f, w\rangle$ are inserted into W_2 and two expired tuples $\langle c, y\rangle$ and $\langle a, x\rangle$ are removed from W_2. Then, the synchronous updates on V_2 and M_2 are done as follows.

- N_2 updates V_2 by inserting the new entries $\langle d, 1, 0\rangle$ and $\langle f, 1, 0\rangle$ and removing the old entry $\langle c, 1, 1\rangle$ (because $\langle c, y\rangle$ has been removed from W_2). Note that the existing entry $\langle a, 2, 1\rangle$ is modified twice, first to $\langle a, 3, 1\rangle$ (i.e., increasing c by 1 to reflect the new tuple $\langle a, z\rangle$ inserted into W_2) and back to $\langle a, 2, 1\rangle$ to reflect the removal of the old tuple $\langle a, x\rangle$ from W_2.
- Next, N_2 ships the two vectors $\mathbf{v}_J = \langle d, f\rangle$ and $\mathbf{v}_J = \langle 0, 2, 1, 1, 1\rangle$ to N_1 and the set $\mathbf{s}_F = \{\langle a, z\rangle\}$. The first element of $\mathbf{v}_C$ is 0 for the old entry $\langle c, 1, 1\rangle$ removed

from V_2 and the remaining three elements reflect the entries in V_2 after the update. Regarding $\mathbf{s}_F$, the other two tuples inserted into W_2 (i.e., $\langle d, x\rangle$, $\langle f, w\rangle$) are not included because the values of b in the corresponding entries in V_2 are both zeros (that is, the full tuples have not been shipped to M_2 yet).

- Upon receiving $\mathbf{v}_J$, $\mathbf{v}_C$, and $\mathbf{s}_F$, N_1 updates M_2 by (1) (based on $\mathbf{v}_J$) inserting the new entries $\langle d, 0, null\rangle$ and $\langle f, 0, null\rangle$ (where $c = 0$ for both entries because no full tuple has been fetched from W_2 to M_2 yet), (2) (based on $\mathbf{s}_F$) inserting the new full tuple $\langle a, z\rangle$ to the entry with $j = a$ (because there are already existing full tuples with the same j) and increasing the c field value of the entry by 1 (i.e., to $\langle a, 3, \&\{\langle a, x\rangle, \langle a, y\rangle, \langle a, z\rangle\}\rangle$) to reflect the new tuple inserted, (3) (based on $\mathbf{v}_C$) removing the old entry $\langle c, 1, \&\{\langle c, y\rangle\}\rangle$ because the element in $\mathbf{v}_C$ corresponding to the entry with $j = c$ is zero, and (4) (based on $\mathbf{v}_C$) removing the first full tuple $\langle a, x\rangle$ from ll of the entry with $j = a$ (i.e., the entry $\langle a, 3, ll\rangle$) because the corresponding element in $\mathbf{v}_C$ is 2 (not 3) and changing the c value of the same entry to 2 to reflect the number of full tuples in ll (thus resulting in the entry $\langle a, 2, \&\{\langle a, y\rangle, \langle a, z\rangle\}\rangle$).

Algorithm 9 outlines the steps of updating V_2 and M_2. Lines 2–13 show the steps of updating V_2 with the new tuples from B_2 and constructing $\mathbf{v}_J$ and $\mathbf{s}_F$. Lines 14–20 show the steps of updating V_2 with the tuples expired in W_2 and constructing $\mathbf{v}_C$ (Line 20). Lines 24–40 show the steps of updating M_2 at N_1. The *for* loop in Line 24 adds new entries of distinct join attribute values from $\mathbf{v}_J$ to M_2. This step guarantees that the updated M_2 and V_2 have the same number of distinct join attribute values in the same order. The *for* loop in Line 27 updates the set of full tuples (in the linked list) and the count field of each entry in M_2 based on $\mathbf{v}_C$ and the number of tuples in $\mathbf{s}_F$. Each element (p) in $\mathbf{v}_C$ is the number of full tuples in W_2 with the join attribute value j and, thus, M_2 must have the same number of full tuples. Given this, the number (k) of expired tuples removed from each entry of M_2 is computed by adding the current count c and the number (q) of new tuples inserted into the linked list and subtracting the count p.

Updating V_2 and $V_2@N_1$ in SM-D1

In Algorithm 5, V_2 (at N_2) and $V_2@N_1$ need to be updated to synchronize with W_2 whenever it is updated with B_2. This update is done by using $\mathbf{v}_J$ and $\mathbf{v}_C$ in a manner similar to Algorithm 9. Specifically, each entry of V_2 has two fields (j, c) where j is the distinct join attribute value and c is the number of tuples in W_2 with the same join attribute value. When W_2 is updated, we update V_2 by inserting a new entry if there is no entry in V_2 with the same join attribute value (j) and removing any entry with the value of c equal to 0. The DJA value vector $\mathbf{v}_J$ and the DJA count vector $\mathbf{v}_C$ are constructed in the same manner as in Algorithm 9, and are shipped to N_1 to update $V_2@N_1$.

It is important to ensure that, for any given multi-way join query, all alternative join execution plans with different combinations of the join order, join placement, and join method are equivalent, as this is essential to the correctness of distributed join algorithms. We give a proof of the equivalence in Appendix A.

Algorithm 9 Updating V_2 and M_2 in SM-S2

```
1  begin
       // Update V2 (at N2).
2      for each tuple s in B2 do
3          N2 probes V2 for matching entries;
4          if a matching entry ⟨j, c, b⟩ is found then
5              c++; // Update the count field of the entry
6              if b == 1 then
7                  N2 inserts s into s_F;
8              end
9          else
10             N2 inserts a new entry ⟨s.J, 1, 0⟩ into V2;
11             N2 inserts an element s.J into the DJA value vector v_J;
12         end
13     end
14     for each expired tuple s do
15         N2 probes V2 for matching entries;
16         if a matching entry ⟨j, c, b⟩ is found then
17             c--; // Update the count field of the entry
18         end
19     end
20     N2 scans V2 and generates a DJA count vector v_C;
21     N2 removes from V2 any entries with c = 0;
22     N2 ships v_C, v_J and s_F to N1;
23     N1 receives v_C, v_J and s_F;
       // Update M2 (at N1).
24     for each element j in v_J do
25         N1 inserts a new entry ⟨j, 0, null⟩ into M2;
26     end
27     for each entry ⟨j, c, ll⟩ of M2 do
28         N1 finds the corresponding count p in v_C;
29         if p = 0 then // All tuples with join attribute value j have been
           removed from W2
30             N1 removes the entry ⟨j, c, ll⟩ from M2
31         else
32             N1 finds full tuples in s_F that has the join value j;
33             if there are q full tuples found then
34                 N1 appends these q full tuples at the end of the linked list ll;
35             end
36             k = c + q − p; // Calculate the number of the expired tuples with
               the join attribute value j
37             N1 removes the first k tuples from the linked list ll;
38             N1 updates the current count c to p;
39         end
40     end
41 end
```

3.3 Handling a tuple-based window

In this section, we discuss how to adapt the proposed algorithms into the case of tuple-based windows. The difference between tuple-based and time-based window is that the window size is determined by the number of tuples instead of the time interval. Therefore, window probing and window update mechanism needs to be modified.

First, for window probing, a tuple s_1 needs to probe tuples s_2 in window W_2 for matching tuples. With tuple-based window, instead of probing tuples s_2 whose timestamp ts is in the range $[s_1.ts - T_2, s_1.ts)$, we want to probe the last w_2 tuples s_2 in W_2 whose timestamp ts is less than or equal $s_1.ts$, where w_2 is the window size of W_2. For this, we need to modify the constraint 3 in our distributed window-based stream join model as follows:

> Constraint 3 [Limited-count match]: *The tuple s_1 should be matched against only the last w_2 tuples s_2 in W_2 whose timestamp ts is less than or equal to $s_1.ts$*

Second, for window update, the number of new tuples k added to the window is equal to the number of expired tuples removed from the window. Thus, as a result we need to change the implementation details for the synchronous window updates. Specifically, for updating $W_2'@N_1$, the algorithm is simpler, since we know the number of expired tuples is the same the number of new tuples k, we only need to ship this number to N_1 and remove the first k tuples from the FIFO queue of $W_2'@N_1$. For updating V_2 and M_2, the algorithms need no modification.

As we can see from the discussion, with tuple-based windows, the high-level algorithms proposed in Sect. 3.1 can be easily adapted with little modifications in the distributed stream join model and in the implementation details of the synchronous window updates.

4 Distributed stream join query optimization algorithm

In a query processing system, the query optimizer generates an efficient query execution plan and the query processor executes the plan. The plan of a multi-way join is characterized by the join order as well as the join placement and the placed join method [14]. The join order specifies the order of the streams participating in the multi-way join. For stream joins, two commonly used ordering approaches are *linear ordering* [49] and *tree ordering* [24]. In this section, we first define a join execution plan for the linear ordering and propose a greedy algorithm which finds an efficient join execution plan in Sect. 4.1. Then, in Sect. 4.2 we discuss the execution of a join plan for the tree ordering given the one-way join algorithms.

4.1 Greedy algorithm

For the join execution plan, we assume *linear* ordering [49] which has proven to be effective in streaming scenarios [10, 11, 21, 44, 56]. For a given m-way join query, linear ordering determines separate join sequences for each stream, thus m join sequences altogether. The join sequence associated with a head stream S_i

$(i = 1, 2, \ldots, m)$ is defined as an ordered set of the other $m - 1$ streams that are joined in sequence for each new tuple s_i arriving in S_i. Thus, there can be a maximum of $(m - 1)!$ possible join orders for each stream, while the actually possible number depends on the structure of the join graph.

With the linear ordering approach, the execution plan of a multi-way stream join $S_1 \bowtie \cdots \bowtie S_m$ is defined as a set of join sequences respectively associated with the streams $S_1, S_2, \ldots, S_m$. The following definition summarizes how a multi-way stream join execution plan is configured and how the plan is executed.

Definition 1 (Multi-way stream join execution) A multi-way join execution plan is defined as a set of join sequences $\{P_1, P_2, \ldots, P_m\}$, where each P_i $(i = 1, 2, \ldots, m)$ is associated with the stream S_i. A join sequence P_i $(i = 1, 2, \ldots, m)$ is in turn defined as $\{\langle S_{i_1}, A_{i_1}\rangle, \ldots, \langle S_{i_{m-1}}, A_{i_{m-1}}\rangle\}$ where A_{i_j} $(j = 1, 2, \ldots, m - 1)$ is the algorithm of one-way join from the output stream of executing the plan up to the stream $S_{i_{j-1}}$, i.e., one-way joins in the subsequence $\{\langle S_{i_1}, A_{i_1}\rangle, \ldots, \langle S_{i_{j-1}}, A_{i_{j-1}}\rangle\}$, to the next stream S_{i_j}. Given such a join execution plan, the sequence of one-way joins in each P_i $(i = 1, 2, \ldots, m)$ is evaluated for each tuple of the head stream S_i, which terminates when either there is no matching tuple in any intermediate join output or the join reaches the last stream $(S_{i_{m-1}})$ in the sequence.

Given the join execution plan and its associated cost formulas (presented in Appendix B), an exhaustive search algorithm may be used to find an optimal plan among all possible alternative plans. However, the computation complexity is exponential. Obviously, this exponential running time becomes prohibitive as the number of joins increases. We thus propose a greedy algorithm which generates an efficient plan in polynomial time.

The greedy algorithm produces a join execution plan that includes a join sequence separately for each input stream. The join sequence is formed by adding one stream at a time at the end of the sequence. The stream chosen to be added each time is the one to which the one-way join is estimated to take the minimum execution time. The estimation is done using the cost formulas of the six one-way join algorithms.

Algorithm 10 shows the steps of the greedy algorithm. The algorithm takes a set of input streams $S_1, S_2, \ldots, S_m$ and a join graph G as the input. Although not shown explicitly, cost parameters like the number of tuples w_i in a window W_i $(i = 1, 2, \ldots, m)$ on each stream, the stream statistics parameters (e.g., the stream rate r_i, the full tuple size F_{S_i}, the partial tuple size P_{S_i} and the selectivity factor f_i of each stream S_i $(i = 1, 2, \ldots, m)$), etc. (see Table 4 in Appendix B) are input as well, as they are needed to compute the estimate of execution time in Line 9. In the input join graph $G \equiv (V, E)$, a vertex $v \in V$ represents an input stream and an edge $e \in E$ represents a join predicate between two streams. The output of the algorithm is a join execution plan as defined in Definition 1 above, that is, one join sequence P_i for each input stream S_i $(i = 1, 2, \ldots, m)$, with each join sequence specifying the join order and the execution algorithm of each join in the sequence.

In the algorithm, each iteration of the outer *for* loop (Lines 2–20) constructs P_i for the input stream S_i. Inside the *for* loop, the *while* loop (Lines 6–19) finds the next stream S_{next} and the next join algorithm A_{next} to be added to P_i. For this, the

Algorithm 10 Greedy distributed stream join query optimization

Input: A set of streams $\{S_1, S_2, \ldots, S_m\}$; a join graph G;
Output: A set of sequences $\{P_1, P_2, \ldots, P_m\}$ for the input set of streams

```
1  begin
2      for each stream S_i (i = 1, 2, ..., m) do
           // Find the join sequence P_i for this stream S_i
3          S_current = S_i; P_i = ∅;
4          Candidates = {S_1, S_2, ..., S_m} − {S_i};
5          Done = {S_i};
6          while Candidates ≠ ∅ do
7              S_next = null; C_min = ∞;
8              for each stream S_k directly connected to any S_j ∈ Done in the join graph
               G do
9                  Estimate the execution time of the six possible join algorithms ({SP-S,
                   SP-D, SM-S1, SM-S2, SM-D1, SM-D2}) between S_current and S_k, and
                   find the algorithm A_next with the smallest estimated execution time
                   (denoted here as C_{A_next});
10                 if (S_next == null) OR (C_{A_next} < C_min) then
11                     S_next = S_k;
12                     C_min = C_{A_next};
13                 end
14             end
15             S_current = S_current ⋈→ S_next;
16             P_i = P_i ∪ {⟨S_next, A_next⟩}; // Append ⟨S_next, A_next⟩ to the join
               sequence P_i
17             Candidates = Candidates − {S_next};
18             Done = Done ∪ {S_next};
19         end
20     end
21 end
```

inner *for* loop (Lines 8–14) estimates the minimum of the costs of executing the six possible one-way join algorithms from $S_{current}$ to each of the streams that can be directly joined with $S_{current}$ in the join graph but has not been included in P_i yet. The estimation of the execution time is computed using the formulas presented in Appendix B. Once S_{next} and A_{next} are determined, $S_{current}$ is updated to be the output stream of the one-way join between $S_{current}$ and S_{next} (Line 15). (The output stream rate of the new $S_{current}$, $r_{current\_new}$, can be estimated as $r_{current\_new} = r_{current\_old} \times w_{next} \times \rho_{old,next}$ where w_{next} is the number of tuples in the window on S_{next} and $\rho_{old,next}$ is the join selectivity factor of the join between $S_{current\_old}$ and S_{next}.) As we can see, this algorithm takes $O(m^2)$ time.

4.2 Discussion on using the tree ordering

In the tree ordering, there is one join order fixed by the tree structure and applied to all streams. That is, a stream at a higher level in the join tree always joins with an

output stream generated from the join represented by the subtree at the immediately lower level. Each join in the join tree is a two-way join between an input stream and an output stream from the subtree, and the two-way join can be computed in two one-way joins using the distributed join algorithms described in Sect. 3.1.

With the tree ordering approach, the execution plan of a multi-way stream join $S_1 \bowtie \cdots \bowtie S_m$ is defined as a sequence of $(m-1)$ two-way joins; each two-way join between S_i and the output stream $S_{1,\ldots,i-1}$ of the subtree is computed in two one-way joins A_{i_1} and A_{i_2} where A_{i_1} is the algorithm of one-way join from S_i to $S_{1,\ldots,i-1}$ and A_{i_2} is the algorithm of one-way join from $S_{1,\ldots,i-1}$ to S_i. For example, for a join execution plan $(S_1 \bowtie S_2) \bowtie S_3$, the two-way join between S_3 and $S_{1,2}$ can be computed as a one-step semijoin-based join from S_3 to $S_{1,2}$ and a simple join from $S_{1,2}$ to S_3.

In order to find an efficient join execution plan with the *tree ordering* approach, we only need a different cost model pertinent to the ordering. The plan optimization algorithm is not affected by the join ordering scheme; only the alternative plans explored are specific to tree ordering. Since the focus of this paper is on developing semijoin-based join algorithms working in distributed stream environments and not on studying the particulars of join ordering, we save the comparison between the two join ordering approaches as the future work.

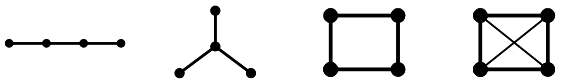
5 Performance evaluation

We have conducted a comprehensive evaluation of the proposed join algorithms. In this section we describe the design, setup, and results of the experiments.

5.1 Experiment design

Two sets of experiments have been conducted. The first set of experiments is to compare the efficiencies of the six different one-way join algorithms presented in Sect. 3 and analyze the observed performance trends. The objective is to see how the join performance is affected by such key factors as the join placement (i.e., source node vs. destination node) and the join method (i.e., simple join vs. semijoin, one-step semijoin vs. two-step semijoin). The second set of experiments is to compare the execution time of a suboptimal plan generated by the proposed greedy algorithm (see Algorithm 10) with the execution time of the optimal plan generated by an exhaustive algorithm. The exhaustive algorithm enumerates on all possible join sequences and join algorithms. Thus, for an m-way join query, the running time complexity of the exhaustive algorithm is $O(m!6^m)$, as there are $(m-1)!$ possible join sequences and there are six alternative join algorithms for each join in the sequence. The objective of this set of experiments is to see how much longer it takes to execute the suboptimal plan generated by the proposed greedy algorithm. We conduct this set of experiments on both synthetic and real data streams.

Depending on the objective of the experiments, we vary different parameters to examine their effects on the performance. The performance metric used is the total query execution time per unit-time (1 second) of tuple arrival. The total query execution time is the sum of the total transmission time (on transmitting data between



(a) Chain. (b) Star. (c) Cycle. (d) Fully connected.

Fig. 6 The topology of a join graph

the nodes participating in the query processing) and the total processing time (on processing data at the individual nodes).

There are three types of parameters used in the experiments: *stream* parameters, *query* parameters, and *system* parameters. The stream parameters are the *tuple size*, the *join attribute selectivity factor* (or *selectivity factor* in short), and the *stream rate* for each stream. The join attribute selectivity factor is the average ratio of tuples selected from a stream for any given join attribute value. We assume the uniform distribution of join attribute values. The stream rate is the number of tuples arriving per second. The query parameters are *window size* of each window and the *join graph topology* of a multi-way join as specified in the query. The three stream parameters and the window size are commonly used in the performance study of data stream processing. The join graph topology will be discussed below. The system parameters are the *network latency*, the *transmission rate* (i.e., *network bandwidth*), and the *execution interval*. The execution interval is the time interval between processing two consecutive batches of tuples (see Sect. 2.2). These system parameters are unique to the distributed stream processing of this paper.

For a multi-way join, the *join graph topology* is another query parameter. We consider the four types of topology shown in Fig. 6. The chain, star and cycle topologies are borrowed from other literature [10, 36, 45], and the fully connected topology is a new one we add. Given a fixed number of nodes participating in a join, the fully connected topology is the worst case to our query optimization algorithm. That is, with the fully connected topology, the produced join sequence is the most likely to be different from the optimal one produced by the exhaustive algorithm. The reason for this is that, at each step of appending another stream to the intermediate join sequence produced by the greedy algorithm thus far, the fully connected topology offers the largest number of possible streams to choose from.

5.2 Experiment setup

We have built a distributed data stream processing system prototype. The prototype runs on each node and has three main modules: the optimizer, the executor, and the communicator. In the optimizer we have implemented two optimization algorithms: the greedy algorithm presented in Sect. 4 and the exhaustive algorithm mentioned above. In the exhaustive algorithm, we use the breadth-first search with simple pruning to reduce the search space. The inputs to the optimizer are the multi-way join query specification (i.e., stream names, window sizes and the join attribute for each pair of streams joined), the stream statistics parameters (i.e, stream rate, tuple size, and selectivity factor), and the optimization algorithm used (i.e., either greedy or exhaustive). The output is a query execution plan, which is then passed to the executor. The executor takes the plan and the data streams as inputs and generates a query output stream. We have implemented the six one-way join algorithms (see Sect. 3.1)

in the executor module. The communicator transfers data between nodes using the TCP/IP protocol. The prototype software is written in Java 2 SDK 1.6.2.

We have conducted the experiments in a simulated wide area network (WAN) environment. In order to simulate a WAN, we use VMWare [1] to create multiple virtual machines connected through a virtual network in which the network bandwidth can be adjusted as needed. The network latency is not supported by VMWare, so we simulate it by injecting a delay in every packet sent out to the virtual network. Each virtual machine is configured with Pentium Core 2 Duo 2.0 GHz processor and 256 MB RAM and runs Linux OS. We run multiple instances of the prototype on separate virtual machines which are the nodes participating in a multi-way join execution. Input streams are read from input files and output streams are written to output files.

We have written a data generator to generate a stream data set as a sequence of tuples. Inputs to the data generator are the number of tuples to be in the data set, the stream rate, the number of attributes in the stream schema, and the name, size, and selectivity factor of each attribute in the schema. The values of the join attributes are assigned randomly with the uniform distribution. We use the string data type for all attributes. Each tuple has a timestamp, the value of which is determined based on the stream rate. We also use the daily access logs of requests made to the 1998 World Cup web site [2] as real data streams.

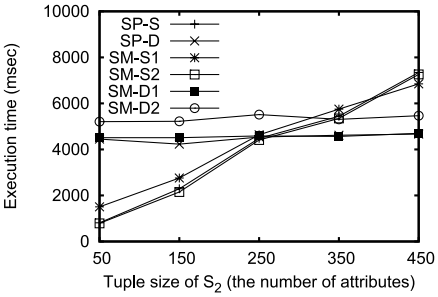
The performance metric is the total elapsed time for executing a join execution plan across all nodes, and is measured as follows. For a given query issued at a certain node (called the *query node*), the optimizer of the node generates a plan and disseminates it to all the nodes. Then, the executor of each node extracts its own portion of the plan and executes it. The execution time, which includes the processing time and the transmission time, is recorded at each node and sent to the query node, which then adds up all from the individual nodes to obtain the total execution time.

For the virtual network environment, we set the default values to 1 Mbps for network bandwidth and 25 ms for network latency. In the experiments that vary the values of these parameters, the bandwidth is varied in the range from 256 Kbps to 4 Mbps and the latency is varied from 5 ms to 45 ms. These ranges have been chosen based on the statistics in the Year 2008 Federal Communications Commission report [3]. The report shows that 40.2% of high-speed lines is in the range of 200 Kpbs to 2.5 Mbps and, among the high-speed lines ranging from 200 Kbps to 2.5 Mbps, 61.9% are DSL and cable modem which have low latency from 5 to 10 ms and from 2.5 ms to 40 ms, respectively.

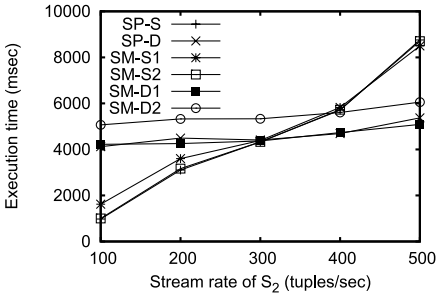
5.3 Experiment results

5.3.1 Comparison of the one-way join algorithms

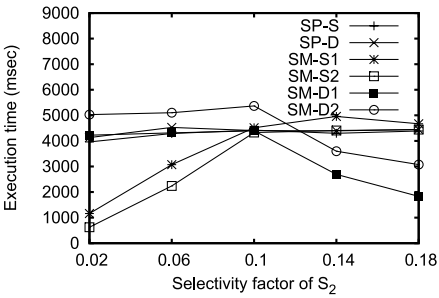
In this set of experiments, we measure the execution times of the six one-way join algorithms (for $S_1@N_1 \vec{\bowtie} S_2@N_2$) while varying the values of the parameters mentioned in Sect. 5.1 except for the join graph topology. Note that the join graph topology is not applicable to one-way joins. For the three stream parameters and the window size, we fix the values of stream S_1 and vary the values of stream S_2. This is to study how the relative performance between two algorithms changes with the relative



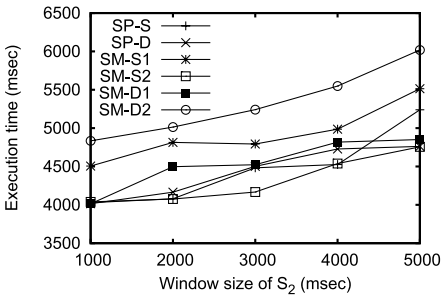
(a) Varying the tuple size of S_2.



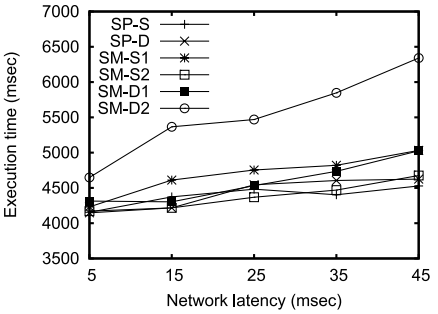
(b) Varying the stream rate of S_2.



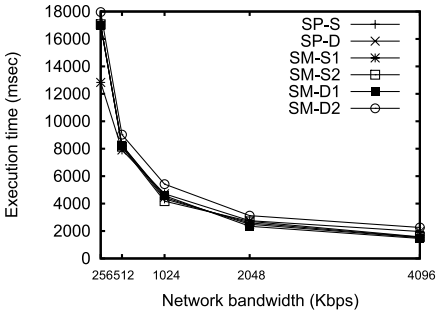
(c) Varying the selectivity factor of S_2.



(d) Varying the window size of S_2.



(e) Varying the network latency.



(f) Varying the network bandwidth.

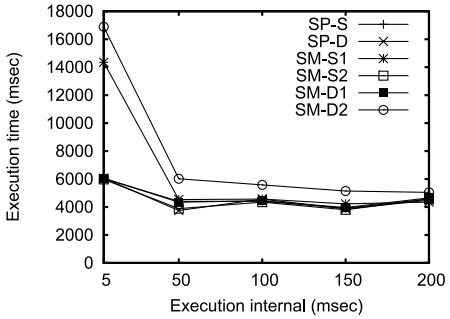
The default settings are as follows for both S_1 and S_2: tuple size = 1250 bytes (amounting to the number of attributes = 250), selectivity factor = 0.1, stream rate = 300 tuples/sec, window size = 3000 msec; network latency = 25 msec, network bandwidth = 1024 Kbps, execution interval = 100 msec. The maximum delay δ due to network latency is set to 80 msec.

Fig. 7 Comparison of the execution times of the six one-way join algorithms

parameter values between the two streams S_1 and S_2. In order to suppress the noise in the measurements, we run each experiment ten times for the duration of one time-unit at each run, and compute the average execution time.

Figure 7 shows the graphs of the experimental results. As we see in the figure, the execution time varies significantly depending on the join algorithm. In order

Fig. 7 (*Continued*)



(g) Varying the execution interval.

to see an overall comparison among the join algorithms, for each of the three factors that characterize the join algorithms (i.e., destination vs. source, simple join vs. semijoin-based join, one-step semijoin reduction vs. two-step semijoin reduction), we divide the join algorithms into two groups and distinguish any observable performance trends between them.

Destination vs. source There are several interesting phenomena observed between the group of algorithms that process join at the source node (SP-S, SM-S1 and SM-S2, called the "source group") and the group of algorithms that process join at the destination node (SP-D, SM-D1 and SM-D2, called the "destination group").

In Figs. 7(a) and (b) (for varying tuple size and stream rate), two performance trends are apparent. First, the performance trend is the same among the source group algorithms, and the same among the destination group algorithms. Second, the execution times of the source group algorithms increase with the parameter values, whereas those of the destination group stay more or less constant. This second trend can be explained based on the following two points: (1) the source group algorithms ship the full tuples of S_2 (from N_2 to N_1) whereas the destination group algorithms do not,[4] and (2) the total volume of full tuples (of S_2) shipped increases with the increase of the stream parameter values (of S_2) and, accordingly, the total data transmission time increases. These two points together explain the crossover of the two performance curves.

In Figs. 7(d), (e), (f) and (g) (for varying window size and system parameters, respectively), there is no particular distinction that can be made between the source group algorithms and the destination group algorithms. This is because varying these parameters affect both groups to the same degree, that is, without regard to where the join processing is done.

In all figures except for Fig. 7(c), the execution times of SP-S and SM-S2 in the source group are very close to each other, and so are the execution times of SP-D and SM-D1 in the destination group. Our analysis of this phenomenon is follows. The default values of the selectivity factors of S_1 and S_2 are set to the same values (i.e,

[4]The destination group algorithms ship the full tuples of S_1 from N_1 to N_2 instead, but the stream parameters of S_1 are fixed in this set of experiments.

$f_1 = f_2 = 0.1$, where f_1 and f_2 are the selectivity factors of S_1 and S_2, respectively) and, thus, in SM-S2 there are always matching tuples in M_2 for every arrival tuple of S_1 (as a result of $B_1 \bowtie M_2$ in Line 3 in Algorithm 4; the result of the first semijoin $M_2 \ltimes B_1$ is empty in this case). Consequently, full tuples of S_2 are always shipped to N_1, and this makes SM-S2 have the same communication overhead as SP-S. Likewise, in SM-D1 every arrival tuple of S_1 has matching tuples in $V_2@N_1$ (as a result of $B_1 \ltimes V_2@N_1$ in Line 3 of Algorithm 5) and, consequently, full tuples of S_1 are always shipped to N_2, and this makes SM-D1 have the same communication overhead as SP-D. Note that Fig. 7(c) is an exception because $f_1 = f_2$ does not hold, since f_2 is varied in this experiment.

Simple join vs. semijoin-based join There is no clear distinction between the group of simple join algorithms (SP-S, SP-D) and the group of semijoin-based join algorithms (SM-S1, SM-S2, SM-D1, SM-D2) for any of the varying parameters, with the exception of the selectivity factor (Fig. 7(c)). In Fig. 7(c), as the selectivity factor (on S_2) increases, the execution times of the simple join algorithms stay unchanged whereas the execution times of the semijoin-based join algorithms do change because the execution times of the simple join algorithms do not depend on the selectivity factors while they do for the semijoin-based join algorithms.

In the group of semijoin-based join algorithms, we have two observations. First, the execution times of the semijoin-based join algorithms in the source group (i.e., SM-S1 and SM-S2) increase and then become nearly constant when f_2 is greater than 0.1 (which is the value of f_1). The reason is as follows. When $f_2 < f_1$, the number of full matching tuples of S_2 increases as f_2 increases and, as a result, the communication overhead increases. In contrast, when $f_2 \geq f_1$, all tuples in W_2 match the arrival tuples of S_1, therefore all shipped to N_2 and, as a result, the communication overhead stops increasing. Second, the execution times of the semijoin-based join algorithms in the destination group (i.e., SM-D1 and SM-D2) decrease when f_2 increases. This is because there are fewer distinct join attribute values of S_2, and thus, fewer tuples of S_1 match the tuples of S_2. This causes fewer tuples of S_1 to be shipped to N_2 and, as a result, the communication overhead decreases.

One-step semijoin vs. two-step semijoin In all figures, we see a trend that the one-step algorithm SM-S1 is more efficient than the two-step algorithm SM-S2 in the source group while, conversely, the two-step algorithm SM-D2 is more efficient than the one-step algorithm SM-D1 in the destination group. Our analysis of the reason is as follows. SM-S1 always makes two shipments for the set of arrival tuples of S_1 in each execution interval (see Lines 3 and 5 in Algorithm 3), while SM-S2 may not (that is, if the result of the first semijoin in Line 3 in Algorithm 4 is empty). In the case of SM-D1 and SM-D2, SM-D2 makes three shipments (see Lines 3, 5 and 7 of Algorithm 6) during each execution interval while SM-D1 makes only one shipment (see Line 4 of Algorithm 5). Thus, SM-S1 and SM-D2 incur higher communication overheads than SM-S2 and SM-D1, respectively.

Other observations In Figs. 7(d) and (e), when the parameter values increase, the execution times increase for all algorithms. The reason is that in Figs. 7(d), when

there are more tuples in the window, it takes longer to probe the window and ship matching tuples, and in Figs. 7(c), when the latency increases, obviously it takes longer to ship data packets.

Figure 7(f) shows the result of the experiment for varying network bandwidth. First, as the network bandwidth increases, the execution time decreases for the obvious reason that the data transmission time decreases. Second, the semijoin-based join algorithms gradually lose their performance advantages when the network bandwidth increases. (Note that the horizontal axis is in the (base 2) exponential scale.) The reason for this is that, as the network bandwidth increases, the communication overhead of shipping full tuples compared with that of shipping partial tuples becomes relatively small.

In Fig. 7(g), when the execution interval increases, the execution time decreases in all join algorithms. The reason is that, as mentioned in Sect. 2.2, as the execution interval increases, there are fewer number of shipments made per unit time and, thus, the total network latency is reduced.

5.3.2 Plan execution time

In this set of experiments, we compare the execution times of the query execution plans produced using the greedy algorithm and the exhaustive algorithm, respectively. We use a four-way join query and vary the values of the stream parameters and the query parameters introduced in Sect. 5.1. The results are compared aggregated for different parameter values. Specifically, for each join graph topology, we generate 100 different queries and, for each query, measure the execution times of the plans produced by the greedy algorithm and the exhaustive algorithm. The parameter values are set randomly, following the approach proposed in [45]. That is, each parameter value is picked randomly from a set of values specified in Table 2. In this way, the algorithms are tested with different input streams and different queries covering a broad range of possible parameter values. Then, for each query, we compute the ratio between the execution times of the plans produced by the two algorithms.

Table 3 shows the statistical summary of the ratio. We see from the table that the ratio increases as the join graph topology changes from the star topology to the fully-connected topology. The reason for this is that, in the greedy algorithm, at each step the next stream in the join sequence is picked from all streams to which there are join edges in the join graph. The fully-connected topology has more possible number of streams that can be joined than the other three topologies and, therefore, the greedy algorithm is less likely to produce an optimal plan. Note that the mean of

Table 2 Parameter values used in the aggregate comparison

Parameter	Set of values
Tuple size (the number of attributes)	10, 50, 100, 150, 200
Selectivity factor	0.02, 0.04, 0.06, 0.08, 0.1
Stream rate (tuples/s)	10, 50, 100, 150, 200
Window size (ms)	400, 800, 1200, 1600, 2000

Table 3 Statistical summary of the ratio T_G/T_E

Topology	Mean $\{\frac{T_G}{T_E}\}$	Median $\{\frac{T_G}{T_E}\}$	Standard deviation $\{\frac{T_G}{T_E}\}$	Max $\{\frac{T_G}{T_E}\}$
Star	1.02	1.00	0.11	1.39
Chain	1.06	1.01	0.23	2.43
Cycle	1.08	1.01	0.21	2.19
Fully-connected	1.14	1.04	0.29	2.33

T_G and T_E denote the execution times of the plans generated by the greedy algorithm and the exhaustive algorithm, respectively. The measured time is an average obtained from ten repeated runs. We do not report the Min ratio as it is theoretically 1.0 (i.e., $T_G \geq T_E$), although some measured ratios are smaller than 1.0 due to the measurement noise

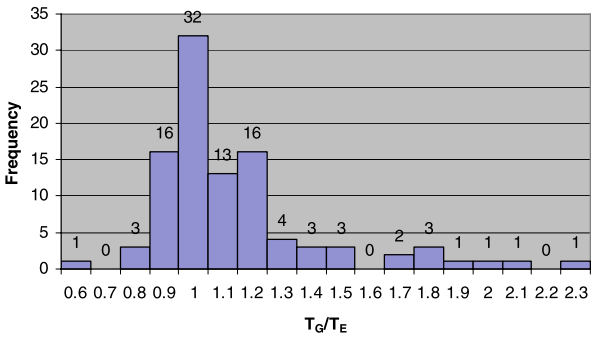


Fig. 8 Frequency distribution of T_G/T_E

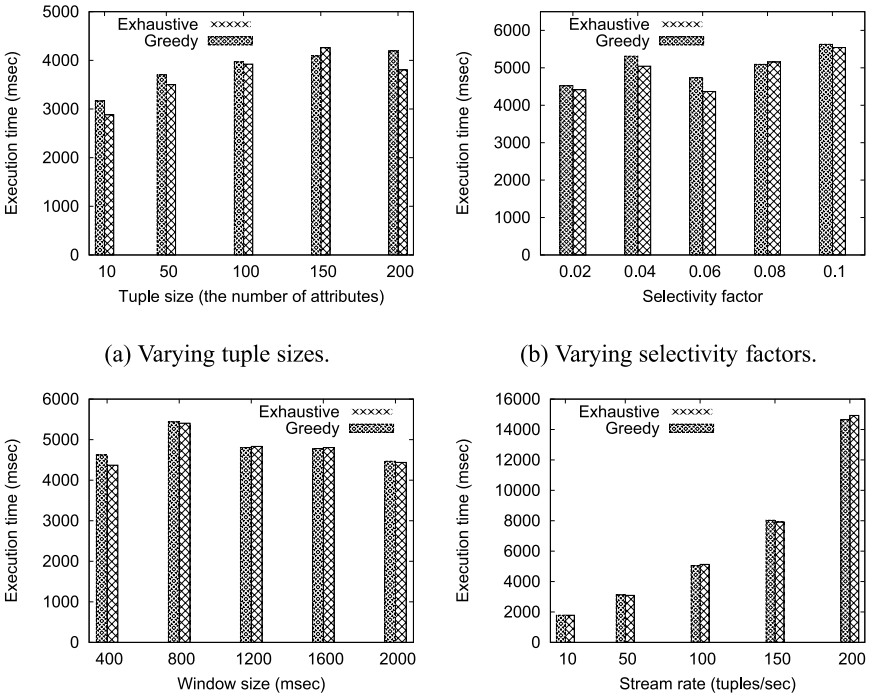
the ratio is only 1.14 even for the fully-connected topology, which means less than 15% overhead in the execution time on average for the greedy algorithm.

Figure 8 shows the frequency distribution of the ratio (rounded to the first fractional digit). We see that the ratio has an exponential distribution skewed to the value of 1.0. Note that, although theoretically T_G/T_E cannot be smaller than 1.0, some values empirically measured are. This is due to the fluctuation of the elapsed time measured in the computing platform.

It is noteworthy that the greedy algorithm is optimal if the values of each parameter shown in Table 2 are the same across all nodes. This is because then all edge weights of the join graph (G) in Algorithm 10 are the same and, consequently, the algorithm finds a minimum spanning tree of the join graph. Figure 9 demonstrates this point for an arbitrary set of parameter values. The measured execution times are the same except for some small differences due to fluctuations in the measurements. (Given the parameter setting, the difference is less than 2%, 6%, and 10% in 12, 17, and 20, respectively, out of the 20 cases.)

Real dataset

We also conduct the experiments using real data streams. The real data streams used are the daily access logs of requests made to the 1998 World Cup web site [2]. The access log from each day is considered a stream. Each entry in an access log rep-



(a) Varying tuple sizes.

(b) Varying selectivity factors.

(c) Varying window sizes.

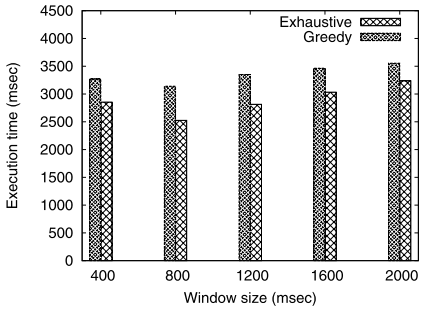
(d) Varying stream rates.

The default settings are as follows: tuple size = 500 bytes (amounting to the number of attributes = 100); stream rate = 100 tuples/sec; network bandwidth = 1024 Kbps; latency = 25 msec; selectivity factor = 0.06; window size = 1200 msec; execution interval = 100 msec; maximum delay = 80 msec. The execution time is an average obtained from ten repeated runs.

Fig. 9 Comparison of the execution time between the greedy algorithm and the exhaustive algorithm with identical parameter values across all nodes

resents a request made to the site and makes a tuple in the corresponding stream. The attributes of each tuple are timestamp, clientID, objectID, size, method, status, type, and server. The number of requests per second is very large, so we have scaled down the timeline by a factor of ten to reduce the stream rate of each stream. In the experiments, we have randomly picked four access logs collected on day 30 through day 33, respectively, and performed a four-way join among them on the clientID attribute.

Figure 10 shows the results for the comparison between the two algorithms. Note that, unlike the synthetic data set case (see Fig. 9), we can vary only the window size, as the stream rate, tuple size and selectivity factor are fixed as constant or average values in the real data set. As shown in the graphs, the execution time of the plan produced by the greedy algorithm is not much longer than that produced by the exhaustive algorithm. The largest difference between the two execution times is at the window size 800, and it is less than 25%. This shows that the greedy algorithm generates efficient plans which are not much worse than the optimal plan.



The tuple size is 142 byetes for all four streams, and the selectivity factors are 0.0024, 0.0017, 0.0015 and 0.001, respectively. The stream rates are 2370, 3670, 3500, and 4200 tuples/sec, respectively, on average. The default settings of the other non-stream parameters are as follows: network bandwidth = 1024 Kbps; latency = 25 msec; execution interval = 100 msec; maximum delay = 80 msec. The execution time is an average obtained from ten repeated runs.

Fig. 10 Comparison of the execution time between the greedy algorithm and the exhaustive algorithm using real data streams

6 Related work

We discuss related work in two areas: distributed data stream query processing and semijoin-based join algorithm in distributed (relational) databases.

6.1 Distributed data stream query processing

Recently there have been many studies on distributed data stream query processing [5, 17, 20, 23, 25, 31, 32, 38, 42, 43, 52, 53]. In these studies, either of the following two types of distributed data stream models is considered: *single (central)* processing site model and *multiple (distributed)* processing site model.

In the single processing site model, all data streams are shipped to the same site for processing. Naturally, all data streams are assumed to have the same schema, and they can be merged into a single stream by the set union operator. With this model, Das et al. [20] address the problem of processing queries including set cardinality expressions. Their work focuses on the accuracy of the set expression cardinality estimate while keeping the data transmission costs at a minimum. Their solutions exploit the global knowledge of the distribution of frequent items as well as the semantics of set expressions to reduce the transmission costs while preserving user-specified error guarantees. With the same model, Olston et al. [38], Gibbons et al. [23], and Keralapura et al. [29] all address the problem of estimating aggregate values with error bounds in distributed data streams. Olston et al.'s work [38] uses "filters" to adapt to changing conditions in order to minimize the transmission overhead from remote nodes to the processing nodes while guaranteeing to provide answers with adequate precision. Gibbons et al. [23] introduce a synopsis data structure for approximating a data stream and present a randomized approximation scheme to estimate the number of distinct values in a sliding window over the union of distributed streams. Keralapura et al. [29] address the problem of counting data elements from different streams at remote nodes and present a method for setting local thresholds at remote nodes

and initiating communication when the number of locally observed data elements exceeds those thresholds.

In the multiple processing site model, data streams are shipped to different sites for processing. In some studies [31, 32, 42, 46] the destination sites are assumed to form a hierarchy. Seshadri et al. [42] address the problem of query optimization in the case different continuous queries are issued at multiple source nodes, and propose a deployment algorithms for placing operators in the queries on different processing nodes to minimize the communication cost. Also using the multiple processing site model, Kumar et al. focus on such issues as resource usage [31] and adaptivity [32] to support scalable distributed stream management. Tang et al. [46] address the CPU resource allocation problem with the objective of maximizing the total of multiple output stream rates and propose solutions for different network topologies. We use multiple processing site model in our work, as it is more general than the single processing model.

Specifically about the *join* processing over distributed data streams, we have found three studies addressing it [30, 54, 56]. Zhang et al. [54] propose *DMJoin* assuming a hierarchy of destination sites. In DMJoin, data streams are forwarded through multiple processing nodes and tuples not satisfying the join condition are filtered out on the way to the final join processing node (i.e., query node at the root of the hierarchy). Zhou et al. [55] present *PMJoin* which partitions a stream into substreams and then forwards those substreams with fewer tuples to the processing site. Their work focuses on developing and studying different stream partitioning schemes. Kriakov et al. [30] use a different join processing model in which each node generates a partition of the stream participating in the join. Their approach is based on incrementally updating the result of Discrete Fourier Transform to reduce data transmission. None of these works on join processing considers the use of distributed semijoin as we do in our work.

6.2 Semijoin-based join algorithm in distributed databases

The semijoin in distributed databases has been introduced by Bernstein and Chiu [12] and Bernstein and Goodman[13] as a technique to reduce communication overhead. As mentioned in Sect. 1, a join can be computed using different semijoin programs which incur different communication overheads. In [12], Bernstein and Chiu examine the questions of which semijoin program is most effective and which type of join queries can be evaluated fully using semijoins. They show that, for tree queries, a full reduction semijoin program can be achieved and a linear searching algorithm can be used to find this semijoin program and that, for cyclic queries, the problem of finding a good semijoin program is intractable and a heuristic approach needs to be used to find it. In [13] Bernstein and Goodman extend their study to include natural join queries and propose an efficient algorithm for finding a full reduction semijoin program for this type of queries.

There are other variants of semijoin proposed since then [15, 27, 33, 37, 40, 41, 48]. Tseng and Chen [48] present a new relational operator called a *hash-semijoin*. The hash-semijoin uses a search filter which presents the projection of a relation in a bit array by hashing the join values. They also propose a method which replaces

some of the semijoins by hash-semijoins in a join sequence. Morrissey and Ogunbadejo [37] investigate the proposed hash-semijoin algorithms and validate them through experiments. Roussopoulos and Kang [27, 41] propose a *two-way semijoin* which includes a backward phase to the semijoin. The backward phase is to reduce the size of the relation that ships its projection. Using the two-way semijoin, they introduce a pipelined N-way join algorithm for joining the reduced relations. Based on the idea of two-way semijoin, Li and Ross [33] present a new two-way semijoin-based join method in which a bit vector (instead of a reduced relation) is sent in the backward phase. Chen and Li [15] introduce a *domain-specific semijoin* for fragmented databases. The operator exploits the semantic information associated with the joining fragmented relations to reduce the size of fragments by unmatching tuples. Perrizo and Chen [40] propose a *composite semijoin* in which multiple join attributes are considered altogether in the projection step of the semijoin, and thus one semijoin operation is done instead of doing several joins between a source site and a destination site.

There also have been techniques developed for utilizing semijoin in distributed query optimization [6, 16, 50]. Wang et al. [50] address the issue of the storage and processing overheads of semijoin and propose a parallel execution method for semijoin to minimize the query response time. Chen and Yu [16] explore an approach to applying a combination of join and semijoin operations to minimize the communication overhead in distributed query processing. Apers et al. [6] address the problem of finding efficient query execution plans with semijoin programs. Their approach considers both processing cost and communication cost and is based on the idea of decomposing a query into simpler queries that can be solved easily and optimally.

All these semijoin techniques and their utilization techniques differ from our work in that they are not for data streams which require the query processing to be incremental and continuous over the network. Specifically, first data stream join is executed by two one-way joins in which new arrival tuples from one stream are matched with the tuples in the window of the other stream, thus instead of projecting a relation, we use partial tuples to reduce the window size of the other stream. Second, windows containing full tuples are updated incrementally as new tuple arrives, and thus windows of partial tuples need to be updated as well, we use bit vector technique to update them efficiently. In addition, linear ordering of multi-way stream join is specific to the data stream, and thus for this, we propose a greedy approach to find an efficient join execution plan based on this ordering.

7 Conclusion

In this paper, we addressed the problem of processing multi-way stream joins over distributed data streams with semijoins. In a distributed environment, including semijoin in the consideration for query execution plans allows for more efficient query processing over the network. Using the notion of a semijoin as a basis, in this paper we first proposed a model for processing distributed stream joins and, then, based on the model, developed one-way join algorithms and, using them as the building blocks, developed a multi-way join algorithm. The one-way join algorithms are based on two alternative join methods (simple join vs. semijoin-based join), two alternative join

placements for each join (source node vs. destination node), and, for the semijoin-based join, two alternative semijoin programs (one step vs. two steps). The multi-way join algorithm assumes the linear ordering of joins.

The streaming nature of the data make the join algorithms significantly more complicated. Thus, in this paper particular attention was given to the correctness of join processing in the face of transmission delay due to network latency and the efficiency of join processing in the face of incremental and continuous arrival of tuples at distributed nodes. We also proposed an optimization algorithm using a greedy heuristic for finding an efficient join execution plan. Finally, we conducted thorough experiments to study the performance of different join algorithms and show the efficiency of our proposed greedy algorithm.

Semijoin is well known as an effective operator for reducing the communication cost for join processing in distributed databases, but it has never been considered for distributed data streams. To our knowledge, this is the first work done to comprehensively address semijoin-based join processing in a distributed data stream environment.

There are several directions for future work. First, in our work we focused on efficiently processing distributed stream joins in which the exact join output is generated. An interesting problem for future work is to give a good approximation for the join output in the case of insufficient system resources. The challenge in this case lies in that the processing is distributed. Finding an adequate quality metric of the approximate result in a distributed environment and computing it efficiently will bring interesting challenges. Second, in data stream processing systems, queries are continuous and long-running and the stream characteristics may vary significantly over time. Thus, query processing needs to adapt to the changes of the stream characteristics. We are currently working on the problem of adaptively processing stream joins in a distributed environment. Third, Bloomjoin is another interesting approach that has been used in distributed databases for reducing the transmission cost. The Bloomjoin approach uses a Bloom filter which is an m-bit vector that is constructed by mapping each join attribute value to an integer from 1 to m using a hash function. As stated by Li and Ross in [33], "due to the nature of hash collisions present in a Bloom filter, Bloomjoin can be viewed as a *lossy* implementation of semijoin." Developing a stream join method using a Bloom filter and integrating it into the distributed stream framework for an efficient (but not lossless) stream join plan will be an interesting exercise. Fourth, as mentioned in Sect. 2.2, the periodic heartbeat/punctuation solution can be incorporated into our work to avoid the risk of incorrect result even in the case of the network latency exceeding δ abruptly. To achieve this, a new join processing model must be used. In the conventional join processing model used in this paper, each stream invalidates its own window tuples independently of the other streams. In the new model, the invalidation of tuples should be driven by the arrival of tuples from another stream. Developing multi-way join and semijoin-based join algorithms to work under the new model with the distributed shipping of heartbeats or punctuations will be an interesting study.

Acknowledgements The authors express their special thanks to the anonymous reviewers for their comments and guidance which were critical to improve the paper to its current shape. The material presented in this paper is based upon work supported by the National Science Foundation under Grant No. IIS-0415023.

Appendix A: Equivalence of the join algorithms

In this section, we show the equivalence of different join execution plans of a join query.

Claim 1 Given the definition of a multi-way stream join execution plan as stated in Definition 1, all alternative join execution plans of a multi-way stream join query over distributed data streams are equivalent.

Proof As stated in Definition 1, a multi-way join is computed as a sequence of one-way joins (Sect. 2). Therefore, it suffices to show the equivalence of the six one-way join algorithms presented in Sect. 3. Besides, since the join algorithms are executed in an incremental manner, we only need to show that the output results of the join algorithms are the same at any point in time or, more specifically, after every τ time units.

It is straightforward that SP-S and SP-D are equivalent because they perform the same join execution $B_1 \ltimes W_2$ at any point in time $t + \delta$ for every τ time units (where t is the time that the join re-execution begins) and are different only in the processing node.

Next, we show that the four semijoin-based join algorithms (i.e., SM-S1, SM-S2, SM-D1, SM-D2) are equivalent to SP-S. Let us use induction for this proof. That is, for each semijoin-based join algorithm, we prove that if it generates the same output as SP-S at time $t + \delta$ then it generates the same output as SP-S at time $t + \delta + \tau$ as well.

- SM-S1 $\equiv$ SP-S: In SM-S1 (see Fig. 4(a) and Algorithm 3), up to the point of time $t + \delta$, $W_2'@N_1$ contains all tuples in W_2 that match the tuples in S_1. At the time $t + \tau$, K_1 is computed and shipped to N_2. N_2 receives K_1 at time $t + \tau + \delta$. $\Delta W_2'$, containing all tuples that match the tuples in B_1 and have not been shipped to N_1, is computed and shipped to N_1 to be inserted into W_2'. Then, after this insertion, W_2' contains all tuples of W_2 that match the tuples in B_1. Note that expired tuples are removed from W_2 at time $t + \tau + \delta$ (see Constraint 2 in Sect. 2.2) and, thus, expired tuples are removed from W_2' at a time later than the time $\Delta W_2'$ arrives at N_1. Hence, the join $B_1 \ltimes W_2'$ returns the same result as $B_1 \ltimes W_2$ after the time $t + \delta + \tau$.
- SM-S2 $\equiv$ SP-S: In SM-D2 (see Fig. 4(b) and Algorithm 4), up to the point of time $t + \delta$, V_2 contains all distinct partial tuples resulting from the projection of W_2, and M_2 contains the same partial tuples as V_2 and, additionally, the full tuples that match the tuples in W_1. After the time $t + \tau$, $\Delta V_2'$, containing the matching partial tuples of which the full tuples are not available in M_2, is computed and shipped to N_2 to obtain full tuples. $\Delta V_2'$ arrives at N_2 no later than $t + \tau + \delta$. Then, $\Delta W_2'$, the set of full tuples thus obtained, is computed and returned to N_1. Note that expired tuples are removed from W_2 at time $t + \tau + \delta$ (see Constraint 2 in Sect. 2.2) and, thus, $\Delta W_2'$ still contains all the matching W_2-tuples. After M_2 is updated with $\Delta W_2'$, M_2 contains all tuples of W_2 that match the tuples in B_1. Hence, the join $B_1 \ltimes M_2$ returns the same result as $B_1 \ltimes W_2$ after the time $t + \delta + \tau$.

- SM-D1 ≡ SP-S: In SM-D1 (see Fig. 4(c) and Algorithm 5), up to the point of time $t+\delta$, V_2 contains all distinct partial tuples resulting from the projection of W_2 and $V_2@N_1$ contains the same partial tuples as V_2. After the time $t+\delta+\tau$, B_1', the result of the semijoin $B_1 \ltimes V_2$, contains all full tuples that match the tuples in W_2. This is because $V_2@N_1$ contains the same partial tuples as V_2 and, thus, contains all possible matching partial tuples of W_2. B_1' arrives at N_2 no later than $t+\tau+\delta$. Hence, the join $B_1' \bowtie W_2$ returns the same output as $B_1 \bowtie W_2$ after the time $t+\delta+\tau$.
- SM-D2 ≡ SP-S: In SM-D2 (see Fig. 4(d) and Algorithm 6), at time $t+\delta+\tau$, K_1, which contains all partial tuples of B_1, is computed and shipped to N_2. K_1 arrives at N_2 no later than $t+\delta+\tau$. Then, K_1', which contains only the partial tuples that have matching tuples in W_2, is computed and shipped to N_1 to obtain their full tuples. Matching tuples in W_2 are saved into *Temp*. Then, B_1', containing all the full matching tuples in B_1, is shipped to N_2. Hence, the join $B_1' \bowtie$ *Temp* returns the same result as $B_1 \bowtie W_2$ after the time $t+\delta+\tau$.

Appendix B: Cost formulas

In this section we present the set of cost formulas used to estimate the cost of a distributed stream join execution plan. Since queries are continuous, we use the unit time cost [28], defined as the execution time per time unit, as the cost metric. For a given JEP, its cost is the total execution time which is the sum of the total query processing time at all nodes and the total transmission time between nodes. The transmission time in turn is the summation of the network latency and the data transfer time.

As mentioned in Sect. 4, a join execution plan (JEP) is the set of per-stream join sequences where each sequence specifies one-way joins executed in a linear order. The total execution time of a JEP, C_{JEP}, is computed as the summation of the total execution times of individual join sequences, C_{P_i} $(i=1,\ldots,m)$. The total execution time of a join sequence is in turn computed as the summation of the total execution time of individual one-way joins in the sequence, $C_{A_{i_j}}$ $(j=1,\ldots,m-1)$. Thus, for an m-way join, the total execution time of its JEP is computed as:

$$C_{JEP} = \sum_{i=1}^{m} C_{P_i} = \sum_{i=1}^{m} \sum_{j=1}^{m-1} C_{A_{i_j}} \tag{1}$$

where A_{i_j} is one of the six one-way join algorithms (i.e., SP-S, SP-D, SM-S1, SM-S2, SM-D1, SM-D2) described in Sect. 3.1. The cost formulas for computing $C_{A_{i_j}}$ $(j=1,\ldots,m-1)$ for these alternative methods are as shown below. Table 4 summarizes the notations used in these formulas. We believe the terms in the formulas are evident from the algorithms.

Simple join at the source node:

$$\begin{aligned} C_{SP\text{-}S} &= C_{update}[W_1] + C_{probe}[W_2] + C_{ship}[W_2] + C_{update}[W_2@N_1] \\ &= r_1 \times c_u + r_1 \times c_p \times w_2 + \frac{r_2}{wp_2} \times S(wp_2 \times F_{S_2}) + r_2 \times c_u \end{aligned}$$

Simple join at the destination node:

$$\begin{aligned} C_{SP\text{-}D} &= C_{update}[W_1] + C_{ship}[B_1] + C_{probe}[W_2] + C_{update}[W_2] \\ &= r_1 \times c_i + \frac{r_1}{wp_1} \times S(wp_1 \times F_{S_1}) + r_1 \times c_p \times w_2 + r_2 \times c_u \end{aligned}$$

One-step semijoin-based join at the source node:

$$\begin{aligned} C_{SM\text{-}S1} &= C_{update}[W_1] + C_{ship}[K_1] + C_{probe}[W_2] + C_{ship}[\Delta W_2'] + C_{probe}[B_1] \\ &\quad + C_{update}[W_2] + C_{update}[W_2'] \\ &= r_1 \times c_i + \frac{r_1}{wp_1} \times S(m_1 \times P_{S_1}) + \frac{r_1}{wp_1} \times m_1 \times c_p \times w_2 + \frac{r_1}{wp_1} \\ &\quad \times S(wp_2 \times f_{21} \times F_{S_2}) \\ &\quad + \frac{r_1}{wp_1} \times wp_2 \times f_{21} \times c_i + r_2 \times c_i + r_1 \times c_p \times w_2 \times f_{21} + \frac{r_2}{wp_2} \times S(c) \end{aligned}$$

Two-step semijoin-based join at the source node:

$$\begin{aligned} C_{SM\text{-}S2} &= C_{update}[W_1] + C_{probe}[M_2] + C_{ship}[\Delta V_2'] + C_{probe}[W_2] + C_{ship}[\Delta W_2'] \\ &\quad + C_{update}[M_2] + C_{probe}[M_2] + C_{update}[W_2] + C_{update}[V_2] \\ &\quad + C_{ship}[\{\mathbf{v}_C, \mathbf{v}_J\} \cup \mathbf{s}_F] + C_{update}[M_2] \\ &= r_1 \times c_i + r_1 \times c_h + \frac{r_1}{wp_1} \times S(k_2 \times (1 - f_{12}) P_{S_2}) + \frac{r_1}{wp_1} \times k_2 \times (1 - f_{12}) \\ &\quad \times c_p \times w_2 \\ &\quad + \frac{r_1}{wp_1} \times S(k_2 \times (1 - f_{12}) \times F_{S_2}) + r_2 \times c_p \times k_2 + r_1 \times c_h + r_2 \times c_i \\ &\quad + \frac{r_2}{wp_2} \times m_2 \times (1 - f_{21}) \times c_i + \frac{r_2}{wp_2} \times S(m_2 \times (1 - f_{21}) \times P_{S_2} \\ &\quad + wp_2 \times f_{21} \times F_{S_2}) + \frac{r_2}{wp_2} \times m_2 \times (1 - f_{21}) \times c_u \end{aligned}$$

One-step semijoin-based join at the destination node:

$$\begin{aligned} C_{SM\text{-}D1} &= C_{update}[W_1] + C_{probe}[V_2@N_1] + C_{ship}[B_1'] + C_{probe}[W_2] + C_{update}[W_2] \\ &\quad + C_{update}[V_2] + C_{update}[V_2@N_1] \\ &= r_1 \times c_i + r_1 \times c_p \times k_2 + \frac{r_1}{wp_1} \times S(wp_1 \times f_{12} \times F_{S_1}) \\ &\quad + r_1 \times f_{12} \times c_p \times w_2 + r_2 \times c_p \times k_2 + r_2 \times c_i \\ &\quad + 2\frac{r_2}{wp_2} \times m_2 \times (1 - f_{21}) \times c_i + \frac{r_2}{wp_2} \times S(m_2 \times (1 - f_{21}) \times P_{S_2}) \end{aligned}$$

Two-step semijoin-based join at the destination node:

$$
\begin{aligned}
C_{SM\text{-}D2} &= C_{update}[W_1] + C_{ship}[K_1] + C_{probe}[W_2] + C_{ship}[K_1'] + C_{probe}[B_1] \\
&\quad + C_{ship}[B_1] + C_{probe}[W_2] + C_{update}[W_2] \\
&= r_1 \times c_i + \frac{r_1}{wp_1} \times S(m_1 \times P_{S_1}) + \frac{r_1}{wp_1} \times m_1 \times c_p \times w_2 \\
&\quad + \frac{r_1}{wp_1} \times S(m_1 \times f_{12} \times P_{S_1}) \\
&\quad + \frac{r_1}{wp_1} \times m_1 \times f_{12} \times c_p \times wp_1 + \frac{r_1}{wp_1} \times S(wp_1 \times f_{12} \times F_{S_1}) \\
&\quad + \frac{r_1}{wp_1} \times wp_1 \times f_{12} \times c_p \times w_2 + r_2 \times c_u
\end{aligned}
$$

Table 4 Notations used in the cost formulas in Appendix B

Notation	Meaning
c_p	Per-tuple probing cost.
c_u	Per-tuple update cost.
c_h	Per-tuple hashing cost.
$C_{probe}[X]$	The cost of probing a set of tuples X.
$C_{update}[X]$	The cost of updating a set of tuples X.
$C_{ship}[X]$	The cost of shipping a set of tuples X.
r_i	The stream rate of S_i, i.e., the average number of tuples arriving at stream S_i in unit time.
r_{ij}	The transmission rate between two nodes N_i and N_j, i.e., total amount (bytes) of data transmitted per unit time.
w_i	The number of tuples in the window W_i. (Note $w_i = r_i \times T_i$.)
wp_i	The number of tuples in the buffer B_i. (Note $wp_i = r_i \times \tau$.)
f_i	The selectivity factor of S_i. (Note $f_i = \frac{1}{d_i}$ where d_i is the average number of the distinct values of the join attribute in S_i.)
f_{ij}	The join selectivity factor of the semijoin from S_i to S_j. (Note $f_{ij} = \min(1, \frac{f_i}{f_j})$, assuming that the set of distinct join attribute values in S_i is a subset of the set of distinct join values in S_j.
F_{S_i}	The size (bytes) of a full tuple in the stream S_i.
P_{S_i}	The size (bytes) of a partial tuple in the stream S_i.
k_i	The number of distinct join attribute values in W_i. (Note $k_i = \min(\frac{1}{f_i}, \frac{w_i}{f_i} * 100)$.)
m_i	The number of distinct join attribute values in B_i. (Note $m_i = \min(\frac{1}{f_i}, \frac{wp_i}{f_i} * 100)$.)
$S(x)$	A function that returns the time it takes to transfer x bytes of data over the network. (Note $S(x) = latency + \frac{x}{bandwidth}$.)
$\mathbf{v}_J$	Distinct join attribute value vector (from V_2) (see Sect. 3.2.2).
$\mathbf{v}_C$	Distinct join attribute count vector (from W_2) (see Sect. 3.2.2).
$\mathbf{s}_F$	The set of full tuples (to be inserted into M_2) (see Sect. 3.2.2).

References

1. VMWare Workstation 6.0: http://www.vmware.com/
2. 1998 World Cup Web Site Access Logs: http://ita.ee.lbl.gov/html/contrib/WorldCup.html
3. A Report of Highspeed Internet Access in the United States made by FCC (Federal Communications Commission), March 2008: http://hraunfoss.fcc.gov/edocs_public/attachmatch/DOC-280906A1.pdf
4. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
5. Amini, L., Jain, N., Sehgal, A., Silber, J., Verscheure, O.: Adaptive control of extreme-scale stream processing systems. In: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, no. 71 (CD) (2006)
6. Apers, P.M.G., Hevner, A.R., Yao, S.B.: Optimization algorithms for distributed queries. IEEE Trans. Knowl. Data Eng. **9**(1), 57–68 (1983)
7. Arasu, A., Babcock, B., Babu, S., McAlister, J., Widom, J.: Characterizing memory requirements for queries over continuous data streams. ACM Trans. Database Syst. **29**(1), 162–194 (2004)
8. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the 21st ACM Symposium on Principles of Database Systems, pp. 1–16. ACM, New York (2002)
9. Babu, S., Arasu, A., Widom, J.: CQL: A language for continuous queries over streams and relations. In: Proceedings of the 8th International Symposium on Database Programming Languages, pp. 1–19. Springer, Berlin (2003)
10. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: Proceedings of the 23rd International Conference on Management of Data, pp. 407–418 (2004)
11. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive caching for continuous queries. In: Proceedings of the 21st International Conference on Data Engineering, pp. 118–129 (2005)
12. Bernstein, P.A., Chiu, D.-M.W.: Using semi-joins to solve relational queries. J. ACM **28**(1), 25–40 (1981)
13. Bernstein, P.A., Goodman, N.: Power of natural semijoins. SIAM J. Comput. **10**(4), 751–771 (1981)
14. Ceri, S., Pelagatti, G.: Distributed databases: Principles and systems (1984)
15. Chen, J.S.J., Li, V.O.K.: Domain-specific semijoin: a new operation for distributed query processing. Int. J. Inf. Sci. **52**(2), 165–183 (1990)
16. Chen, M.-S., Yu, P.S.: Combining join and semi-join operations for distributed query processing. IEEE Trans. Knowl. Data Eng. **5**(3), 534–542 (1993)
17. Cormode, G., Muthukrishnan, S., Zhuang, W.: What's different: Distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In: Proceedings of the 22nd International Conference on Data Engineering, no. 57 (CD) (2006)
18. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of the 22nd International Conference on Management of Data, pp. 647–651. ACM, New York (2003)
19. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: Proceedings of the 22nd International Conference on Management of Data/Principles of Database Systems, pp. 40–51. ACM, New York (2003)
20. Das, A., Ganguly, S., Garofalakis, M.N., Rastogi, R.: Distributed set expression cardinality estimation. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 312–323 (2004)
21. Gedik, B., Wu, K.-L., Yu, P.S., Liu, L.: A load shedding framework and optimizations for m-way windowed stream joins. In: Proceedings of the 23rd International Conference on Data Engineering, pp. 536–545 (2007)
22. Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G., Elmagarmid, A.K.: Incremental evaluation of sliding-window queries over data streams. IEEE Trans. Knowl. Data Eng. **19**(1), 57–72 (2007)
23. Gibbons, P.B., Tirthapura, S.: Distributed streams algorithms for sliding windows. In: Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 63–72. ACM, New York (2002)
24. Golab, L., Ozsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 500–511. ACM, New York (2003)

25. Gorawski, M., Marks, P.: Fault-tolerant distributed stream processing system. In: Proceedings of a Workshop of 17th International Conference on Database and Expert Systems Applications, pp. 395–399 (2006)
26. Gu, X., Yu, P.S., Wang, H.: Adaptive load diffusion for multiway windowed stream joins. In: Proceedings of the 23rd International Conference on Data Engineering, pp. 146–155 (2007)
27. Kang, H., Roussopoulos, N.: Using 2-way semijoins in distributed query processing. In: Proceedings of the 3rd International Conference on Data Engineering, pp. 644–651. IEEE Comput. Soc., Los Alamitos (1987)
28. Kang, J., Naughton, J.F., Viglas, S.D.: Evaluating window joins over unbounded streams. In: Proceedings of the 19th International Conference on Data Engineering, pp. 341–352. IEEE Comput. Soc., Los Alamitos (2003)
29. Keralapura, R., Cormode, G., Ramamirtham, J.: Communication-efficient distributed monitoring of thresholded counts. In: Proceedings of the 25th International Conference on Management of Data, pp. 289–300 (2006)
30. Kriakov, V., Delis, A., Kollios, G.: Approximate data stream joins in distributed systems. In: Proceedings of the 27th IEEE International Conference on Distributed Computing Systems, no. 5 (CD) (2007)
31. Kumar, V., Cooper, B.F., Cai, Z., Eisenhauer, G., Schwan, K.: Resource-aware distributed stream management using dynamic overlays. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, pp. 783–792 (2005)
32. Kumar, V., Cooper, B.F., Schwan, K.: Distributed stream management using utility-driven self-adaptive middleware. In: Proceedings of the 2nd International Conference on Autonomic Computing, pp. 3–14 (2005)
33. Li, Z., Ross, K.A.: Perf join: An alternative to two-way semijoin and bloomjoin. In: Proceedings of the 4th International Conference on Information and Knowledge Management, pp. 137–144 (1995)
34. Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., Maier, D.: Out-of-order processing: a new architecture for high-performance stream systems. In: Proceedings of the 34th International Conference on Very Large Data Bases, pp. 274–288 (2008)
35. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proceedings of the 21st International Conference on Management of Data, pp. 49–60 (2002)
36. Moerkotte, G., Neumann, T.: Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In: Proceedings of the 32th International Conference on Very Large Data Bases, pp. 930–941 (2006)
37. Morrissey, J.M., Ogunbadejo, O.: Combining semijoins and hash-semijoins in a distributed query processing. In: Proceedings of 1999 IEEE Canadian Conference on Electrial and Computer Engineering, pp. 122–126. IEEE Comput. Soc., Los Alamitos (1999)
38. Olston, C., Jiang, J., Widom, J.: Adaptive filters for continuous queries over distributed data streams. In: Proceedings of the 22nd International Conference on Management of Data, pp. 563–574 (2003)
39. Ozsu, M.T., Valduriez, P.: Principles of distributed database systems (1999)
40. Perrizo, W., Chen, C.-S.: Composite semijoins in distributed query processing. Int. J. Inf. Sci. **50**(2), 197–218 (1990)
41. Roussopoulos, N., Kang, H.: A pipeline n-way join algorithm based on the 2-way semijoin program. IEEE Trans. Knowl. Data Eng. **3**(4), 486–495 (1991)
42. Seshadri, S., Kumar, V., Cooper, B.F.: Optimizing multiple queries in distributed data stream systems. In: Proceedings of Workshop of the 22nd International Conference on Data Engineering, no. 25 (CD) (2006)
43. Sharfman, I., Schuster, A., Keren, D.: A geometric approach to monitoring threshold functions over distributed data streams. In: Proceedings of the 25th International Conference on Management of Data, pp. 301–312 (2006)
44. Srivastava, U., Munagala, K., Widom, J.: Operator placement for in-network stream query processing. In: Proceedings of the 24th Symposium on Principles of Database Systems, pp. 250–258 (2005)
45. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. VLDB J. **6**(3), 191–208 (1997)
46. Tang, A., Liu, Z., Xia, C.H., Zhang, L.: Distributed resource allocation for stream data processing. In: Proceedings of the 2nd International Conference on High Performance Computing and Communications, pp. 91–100 (2006)
47. Tran, T.M., Lee, B.S., Bovee, M.W.: Why not semijoins for streams, when distributed? In: Proceedings of the 2nd International Conference on Digital Telecommunication, no. 27 (CD) (2007)

48. Tseng, J.C.R., Chen, A.L.P.: Improving distributed query processing by hash-semijoins. J. Inf. Sci. Eng. **8**(4), 525–540 (1992)
49. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of the 29th International Conference on Very Large Data Bases, pp. 285–296 (2003)
50. Wang, C., Chen, A.L.P., Shyu, S.-C.: A parallel execution method for minimizing distributed query response time. IEEE Trans. Parallel Distrib. Syst. **3**(3), 325–333 (1992)
51. Wang, S., Rundensteiner, E.A., Ganguly, S., Bhatnagar, S.: State-slice: New paradigm of multi-query optimization of window-based stream queries. In: Proceedings of 28th International Conference on Very Large Data Bases, pp. 619–630 (2006)
52. Xia, T., Jin, C., Zhou, X., Zhou, A.: Filtering duplicate items over distributed data streams. In: Proceedings of the 6th International Conference on Web-Age Information Management, pp. 779–784 (2005)
53. Zhang, D., Li, J., Wang, W., Guo, L., Ai, C.: Processing frequent items over distributed data streams. In: Proceedings of the 7th Asia-Pacific Web Conference, pp. 523–529 (2005)
54. Zhang, D., Li, J., Kimeli, K., Wang, W.: Sliding window based multi-join algorithms over distributed data streams. In: Proceedings of the 22nd International Conference on Data Engineering, no. 139 (CD) (2006)
55. Zhou, Y., Yan, Y., Ooi, B.C., Tan, K.-L., Zhou, A.: Optimizing continuous multijoin queries over distributed streams. In: Proceedings of the 14th International Conference on Information and Knowledge Management, pp. 221–222 (2005)
56. Zhou, Y., Yan, Y., Yu, F., Zhou, A.: PMJoin: Optimizing distributed multi-way stream joins by stream partitioning. In: Proceedings of the 9th International Conference on Database Systems for Advanced Applications, pp. 325–341 (2006)