Why Not Semijoins for Streams, When Distributed?

Tri Tran Byung Suk Lee
Department of Computer Science
University of Vermont
Burlington, VT 05405
{ttran, bslee}@cems.uvm.edu

Matthew W. Bovee
School of Business Administration
University of Vermont
Burlington, VT 05405
bovee@bsad.uvm.edu

Abstract

This paper addresses the semijoin-based window join algorithm over distributed data streams. In distributed stream query processing, data streams arriving at remote sites need to be shipped to the processing site for query execution. This typically introduces high communication overhead over the network. Our observation is that semijoin, effective to reduce communication overhead in distributed database query processing, can be also effective in distributed stream *query processing. The challenge, of course, lies in the streaming nature of tuples, the processing of which is fundamentally different from processing a set of tuples. We address this challenge by first adapting the window-based stream join to a distributed environment. The resulting join algorithm (called* simple join*) uses the idea of exporting a window to the query processing site. We then adopt the semijoin to reduce the communication overhead (in return for a marginal increase of the processing overhead). The resulting* semijoin-based join *algorithm uses the ideas of a* mirror window *and a* partial tuple*. That is, it creates a copy of a remote window at the processing site and sends a partial tuple to probe for matching tuples before sending a full tuple. Finally, we analyze the two join algorithms using our proposed cost models and verify the analysis results through a set of experiments.*

1 Introduction

Recently, distributed stream joins have been gaining attention in the research community [24, 26]. A distributed stream join is different from a local stream join in that data streams arriving at remote sites need to be shipped over a communication network to the processing site for join execution. Communication overheads of this shipment can be very high in many applications, and this brings a need for a technique to reduce the communication overheads. A few examples of distributed stream joins are given here.

Example 1. In network traffic monitoring [26], suppose we want to monitor the traffic of data packets passing through two particular routers in order to find packets with the same destination address. The two routers can be considered two sites, and data packets going through the routers can be considered data streams (S_1 and S_2). Each data packet contains a destination IP address $dest$. This monitoring task then can be specified as a distributed stream join query $S_1 \bowtie_{S_1.dest=S_2.dest} S_2$. □

Example 2. In news stream filtering [15], suppose we want to find recent articles on the same topic published by two news network services, say, Associated Press and Reuters. The two news network services can be considered two sites, and articles generated from the news network services can be considered news streams (S_A for Associated Press, and S_R for Reuters). Each news article in a stream is tagged with a set of weighted keywords SK. The monitoring task then can be specified as a distributed stream join query $S_A \bowtie_{S_A.SK=S_R.SK} S_R$ where $S_A.SK = S_R.SK$ is a set equality comparison. □

In a data stream system, data arrive as a continuous sequence of tuples, and windows are needed to limit the number of tuples processed. We thus assume a *window-based* join in this paper. (The window-based join algorithm over local data streams has been proposed by Kang et al.[10].) A simple way of applying a window-based join to distributed data streams is to ship all tuples of remote data streams to the processing site and execute the join on that site. This method, however, incurs high communication overheads if the volume of shipped data is large. We call this method the *window-based distributed stream simple join (SPJ)*.

In distributed databases, semijoin is well known as an effective operator to decrease the communication overheads of join queries [2, 3, 4, 14, 21]. A semijoin from relation R_1 to relation R_2, denoted as $R_2 \ltimes R_1$, is equivalent to $\Pi_{Attr(R_2)}(R_1 \bowtie R_2)$ where $Attr(R_2)$ denotes all attributes of R_2. With 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" (or strategies) [17]: $R_1 \bowtie (R_2 \ltimes R_1)$, $(R_1 \ltimes R_2) \bowtie R_2$, and $(R_1 \ltimes R_2) \bowtie (R_2 \ltimes R_1)$.

Computing join using a semijoin program may incur lower communication overhead due to relation size reductions caused by semijoin. 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 this projection to R_2's site and joining with R_2. The result of the semijoin is a reduced R_2 with

Second International Conference on Digital Telecommunications (ICDT'07)
0-7695-2910-0/07 $20.00 © 2007 IEEE

only the tuples that contribute to the final join with R_1. If this reduction of R_2 is larger than the projection of R_1, then using the semijoin incurs lower communication overhead.

Intuitively, the semijoin technique should be effective in distributed *data stream* joins as well, but to our knowledge, there has been no concrete research done on that. By applying the ideas of the semijoin and the window-based stream join in a distributed data stream environment, we introduce the *distributed stream semijoin-based join (SMJ)* algorithm in this paper. Applying the ideas poses two main technical challenges: adapting local windows to *distributed* streams and adapting the semijoin technique to data *streams*.

Adapting windows to distributed streams: In a centralized stream system all windows on streams are at the same site, but in a distributed stream system they are at different sites. Thus, in order to perform a window-based stream join on distributed streams, we need to maintain a copy of each window at the processing site. In the SPJ algorithm, we export the window and maintain only its copy at the processing site. In the SMJ algorithm, we maintain both the original window at the stream site and its copy at the processing site. We call this copy a *mirror window*.

Adapting semijoin to data streams: A straightforward way to apply the semijoin in a distributed data stream environment would be to treat the window on each stream as a relation and ship a projection of a window at one site N_i to another site N_j. However, since queries are continuous and long-running and windows are updated continuously, shipping the projection of a window from N_i to N_j and shipping the semijoin result for the final join should be done for every new arriving tuple. This would be very expensive. To handle this problem, we ship only a *partial* tuple (tuple consisting of timestamp and join attribute only) of each new arriving tuple from N_i to N_j and save the tuple in a mirror window. If N_j is the query site, then the *full* tuple is shipped (to produce the join output) only if there is a matching tuple for the partial tuple. The communication overhead is decreased if the decrease of overhead for not sending non-matching full tuples is larger than the increase of overhead for sending partial tuples for every new tuple.

Similarly to the distributed database case, SMJ can be executed using different semijoin programs. In this paper, we present the algorithm using one of them (corresponding to the strategy $R_1 \bowtie (R_2 \ltimes R_1)$ in distributed database). Our objective is to present the idea of using semijoin technique in distributed stream join and to show that SMJ can be more efficient than SPJ. The problem of finding the best semijoin program is out of the scope of this paper.

We develop cost models of the two join algorithms. The cost metric is the total execution time, which includes the processing time at all sites and the transmission time. We then use the cost models to evaluate the join performances. We analyze the cost models to show qualitatively that SMJ costs less than SPJ when the join selectivity is low and when partial tuples are small compared with full tuples. We also implement the two join algorithms and conduct a set of experiments to show the join performances quantitatively and verify our analysis.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the distributed data stream model and the window stream join model. Section 4 proposes the two join algorithms SPJ and SMJ. Section 5 evaluates the performances of the join algorithms. Section 6 concludes the paper and discusses future work.

2 Related Work

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

Recently there have been many studies on distributed data stream query processing [16, 7, 8, 19, 9, 25, 22, 6, 18, 13, 12]. In these, two types of distributed data stream models are considered based on the number of processing sites: *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 a set union operator. With this model, Das et al. [7] address the problem of processing queries including set cardinality expressions, and Olston et al. [16], Gibbons et al. [8], and Keralapura et al. [11] all address the problem of finding frequent items in distributed data streams.

In the multiple processing site model, data streams are shipped to different sites for processing. In some studies [18, 13, 12] the destination sites are assumed to form a hierarchy and methods have been proposed for placing operators to minimize the communication cost. Other studies examine such issues as load balancing [23], fault tolerance [9], and distributed architecture [5]. In this paper we consider the more general multiple processing site model.

Specifically about *join* processing in distributed data streams, to our knowledge only two studies have addressed it [24, 26]. Neither involved the use of distributed semijoin. Zhang et al. [24] 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 join final processing node (i.e., query node at the root of the hierarchy). Zhou et al. [26] present PMJoin using the heuristic of partitioning a stream into substreams and then forwarding those substreams with fewer tuples to the processing site.

The semijoin in distributed databases has been introduced by Bernstein and Chiu[2] and Bernstein and Goodman[3] as a technique to reduce communication overhead. Other papers address different issues of semijoin, such as the improvement of semijoin [14] and the utilization of semijoin in query optimization [4, 21]. Li and Ross [14] 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. These semijoin techniques differ from our work presented here in that they do not consider distributed data streams.

3 Distributed Stream Join Model

In this section, we describe a distributed stream join model assumed in our work. It comprises the distributed data stream model and the window stream join model.

Distributed data stream model: Consider a set of n nodes (or sites) $N_1, N_2, \ldots, N_n$ connected through a communication network. In each node N_i, n_i data streams $S_1^i, S_2^i, \ldots, S_{n_i}^i$ are arriving in the form of an ordered sequence of tuples. Each tuple in the stream has a timestamp, TS, and a join attribute, J, as part of its schema. Each data stream S_j^i ($i = 1 \ldots n, j = 1 \ldots n_i$) has a stream rate λ_j^i which is defined as the number of tuples generated (or arriving) in the stream per time unit (second). Two nodes N_k and N_l are connected via a link with a transmission rate $\lambda_{k,l}$ which is defined as the amount of data transmitted per time unit (bytes/second).

Window stream join model: We assume the window-based join proposed in [10]. A window can be either tuple- or time-based. If tuple-based, a window size is the number of tuples in the window; if time-based, a window size is the time interval from the current time point to the past. We denote a window of stream S_j by W_j (the site id is irrelevant here). A two-way window join between two streams S_1 and S_2 with windows W_1 and W_2, respectively, is computed as follows. For each new tuple s_1 arriving in S_1, matching tuples are found from the window W_2 and then output. Then, the new tuple s_1 is inserted into W_1 and any expired tuples are removed from W_1. The computation is symmetric for a new arriving tuple in S_2.

In this paper, we consider the problem of window-based join between two nodes N_1 and N_2. The query result is produced at node N_2. For the simplicity of presentation, we assume only one stream S_1 with window W_1 at node N_1 joining with another stream S_2 with window W_2 at node N_2. As mentioned in Section 1, we assume that a semijoin is performed by shipping S_1 at N_1 to N_2 and joining with S_2 at N_2. That is, the SMJ algorithm considers the strategy of shipping partial tuples of S_1 at N_1 to N_2.

4 Distributed Stream Join Algorithms

In this section we present the distributed stream join algorithms: simple join (SPJ) and semijoin-based join (SMJ).

4.1 Simple Join

The idea of the SPJ algorithm is to export the window W_1 and maintain it (denoted as W_1') at node N_2 by shipping every new tuple of stream S_1 to N_2. The join is done at N_2 as a local window join with windows W_1' and W_2. For each new tuple arriving, the join proceeds in the following steps.

For each new tuple s_1 at node N_1:

1. N_1 ships s_1 to N_2.
2. N_2 receives s_1, probes W_2 for matching tuples and outputs the result.
3. N_2 updates W_1' by inserting s_1 and removing any expired tuples.

For each new tuple s_2 at node N_2:

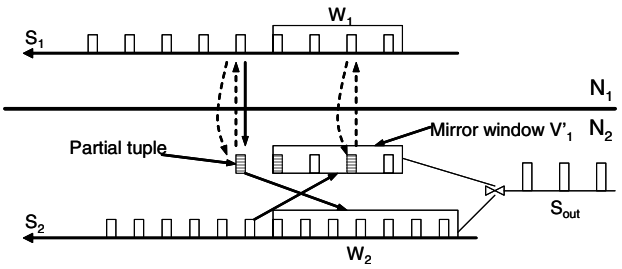


Figure 1. Semijoin-based join processing.

1. N_2 probes W_1' for matching tuples and output the result.
2. N_2 updates W_2 by inserting s_2 and removing any expired tuples.

A major disadvantage of the SPJ algorithm is that it may have high communication overheads. That is, if the stream rate λ_1 of S_1 is high and tuple size of S_1 is large. In the next subsection, we introduce the SMJ algorithm that, depending on the statistics of the joined streams, can reduce the communication cost significantly.

4.2 Semijoin-based join

The SMJ algorithm aims at reducing the communication overhead of transmitting tuples between node N_1 and N_2. The idea is that for each new tuple s_1, a *partial* tuple ps_1 which includes only the timestamp and the join attribute is shipped to N_2 and used to probe W_2 for a matching tuple and, if a matching tuples is found, then the full tuple s_1 is shipped to N_2 and matched with tuples in W_2 to produce output tuples. Fig. 1 illustrates the processing of the SMJ algorithm. In node N_2, we maintain a mirror of window W_1, denoted as V_1'. V_1' is different from W_1' used in the SPJ algorithm in that it contains two types of tuples: partial tuples and full tuples. A partial tuple ps_1 is stored if there is no matching tuple found in W_2 for ps_1; otherwise, a full tuple s_1 is stored. For each new tuple arriving, the join proceeds in the following steps.

For each new tuple s_1 at node N_1:

1. N_1 ships a partial tuple ps_1 to N_2.
2. N_1 updates W_1 by inserting s_1 and removing any expired tuples.
3. N_2 receives ps_1 and probes W_2 for the *first* matching tuple. (ps_1 is not inserted into V_1' yet.)
4. If a matching tuple is found then
   (a) N_2 sends a request for the full tuple s_1 back to N_1.
   (b) N_1 receives a request for s_1, probes W_1 for the full tuple s_1, and then sends the found s_1 to N_2.
   (c) N_2 receives s_1, probes W_2 for *all* matching tuples and outputs the joined matching tuples, and then updates V_1' by inserting s_1 and removing any expired tuples.

   else (i.e., a matching tuple is not found)
   (a) N_2 updates V_1' by inserting ps_1 and removing any expired tuples.

For each new tuple s_2 at node N_2:

1. N_2 probes V_1' for the *first* matching tuple.
2. If a matching tuple is found and it is a partial tuple ps_1 then
   (a) N_2 sends a request for *all* full tuples that have the same join attribute value as ps_1 to N_1.
   (b) N_1 receives a request for full tuples, probes W_1 for the full tuples $(\{s_{1_1}, s_{1_2}, \cdots, s_{1_k}\})$, and then ships them to N_2.
   (c) N_2 receives the full tuples $\{s_{1_1}, s_{1_2}, \cdots, s_{1_k}\}$, outputs the joined matching tuples $\{s_{1_1}||s_2, s_{1_2}||s_2, \cdots, s_{1_k}||s_2\}$, and then updates V_1' by replacing all of their partial tuples ps_1 by $\{s_{1_1}, s_{1_2}, \cdots, s_{1_k}\}$.

   else (i.e., if the matching tuple is a full tuple s_1, so there may be more matching full tuples in V_1')
   (a) N_2 probes V_1' for all matching full tuples and outputs the joined matching tuple $s_1||s_2$.

The communication between the two nodes is essentially asynchronous and, therefore, the actual implementation of these algorithms may well be event-driven.

5 Performance Study

In this section we first build the cost models of the two join algorithms and analyze their performances based on the cost models. Then, we compare the performances of the two join algorithms through experiments conducted using a program implementing the algorithms.

5.1 Analysis

The cost models we develop estimate the total execution times of join algorithms. Since queries are continuous in our work, we use the unit time cost [10], defined as the total execution time per time unit, as a cost metric. The execution time is the total of processing time at each node (N_1, N_2) and transmission time between the two nodes. That is, $C_{join} = C_{proc\_at\_N_1} + C_{proc\_at\_N_2} + C_{trans}$. We assume that the transmission cost is composed of transmission time and latency where the latency is a constant. We also assume that the stream rates are temporal averages and that the rates do not change much over time.

Generic unit-time cost models for SPJ and SMJ are formulated as shown below. We believe the terms in the formulas are evident from the join processing steps outlined in Section 4.1 and Section 4.2.

$$C_{SPJ} = \lambda_1[C^*_{ship\_s_1} + C_{probe\_W_2} + C_{update\_W_1'}] + \lambda_2[C_{probe\_W_1'} + C_{update\_W_2}]$$

$$C_{SMJ} = \lambda_1[C_{ship\_ps_1} + C_{update\_W_1} + C^*_{probe\_W_2} + C_{send\_request} + C^*_{probe\_W_1} + C^*_{ship\_s_1} + C_{probe\_W_2} + C_{update\_V_1'}] + \lambda_2[C^*_{probe\_V_1'} + C_{send\_request} + C_{probe\_W_1} + C_{ship\_s_1} + C^*_{update\_V_1'} + C_{update\_W_2}]$$

In the above formulas, the cost terms marked with '*' are different from those without '*'. Specifically, $C^*_{probe\_W_1}$, $C^*_{probe\_W_2}$ and $C^*_{probe\_V_1'}$ are the costs of finding only the *first* matching tuple in W_1, W_2, and V_1', respectively, whereas $C_{probe\_W_1}$ and $C_{probe\_W_2}$ are the costs of finding *all* matching tuples in W_1 and W_2, respectively. $C^*_{ship\_s_1}$ is the cost of shipping a *single* full tuple with the same join attribute value whereas $C_{ship\_s_1}$ is the cost of shipping *all* full tuples. $C^*_{update\_V_1'}$ is the cost of updating V_1' by replacing all partial tuple in V_1' by the corresponding full tuples received from N_1.

Now, we compare analytically the performances of the two join algorithms using the cost models. For this, we compare the transmission costs and the processing costs separately and, then, combine the two to make a conclusion.

First, for the transmission costs, let l be the latency of a transmission cost, d_F be the size of a full tuple and d_P be the size of a partial tuple (both in bytes). Let d_V be the number of distinct values of the join attribute, and let $|W_1|$ and $|W_2|$ be the number of tuples in the window W_1 and W_2, respectively. Then, the transmission costs of the two join methods are formulated as follows.

$$C_{SPJ\_trans} = \lambda_1 * C_{ship\_s_1} = \lambda_1(l + \frac{d_F}{\lambda_{1,2}}) \quad (1)$$

$$\begin{aligned} C_{SMJ\_trans} &= \lambda_1(C_{ship\_ps_1} + C_{send\_request} + C^*_{ship\_s_1}) \\ &\quad + \lambda_2(C_{send\_request} + C_{ship\_s_1}) \\ &= \lambda_1[(l + \frac{d_P}{\lambda_{1,2}}) + p_2(l + \frac{d_P}{\lambda_{1,2}} + l + \frac{d_F}{\lambda_{1,2}})] \\ &\quad + \lambda_2[(1 - p_2) * p_1 * (l + \frac{d_P}{\lambda_{1,2}} + l + \frac{|W_1|}{d_V} * \frac{d_F}{\lambda_{1,2}})] \end{aligned} \quad (2)$$

In these formulas, p_1 and p_2 are the probabilities of finding a matching tuple in W_1 and W_2, respectively. The probabilities p_1 and p_2 can be estimated using the number of distinct values d_V and window sizes $|W_1|$ and $|W_2|$ as $p_i = 1 - (\frac{d_V - 1}{d_V})^{|W_i|}$ for $i = 1, 2$. This is because the probability of its complement $(1 - p_i)$ (i.e., the probability of no matching tuple found given a specific value v) can be estimated by the fraction of number of possibilities placing $d_V - 1$ distinct values (without v) and the number of possibilities placing d_V distinct values in $|W_i|$ positions.

As we can see from the above formulas, the transmission cost of SMJ depends on p_i $(i = 1, 2)$. Moreover, p_i is a function of d_V and W_i. If d_V is sufficiently large relative to W_i so that $p_i \approx 0$ holds, then $C_{SMJ\_trans} - C_{SPJ\_trans} \approx \frac{\lambda_1}{\lambda_{1,2}}(d_P - d_F)$. The value of this difference is less than (or equal to) 0 because $d_P \leq d_F$. The absolute value of this difference increases linearly with $|d_P - d_F|$, that is, the higher the $|d_P - d_F|$ is, the more benefit SMJ brings.

In contrast, if W_i is sufficiently large relative to d_V so that $p_i \approx 1$ holds, then $C_{SMJ\_trans} - C_{SPJ\_trans} \approx 2\lambda_1(l + \frac{d_P}{\lambda_{1,2}})$. The value of this difference is greater than 0 because all terms are positive. The difference, however, is typically small compared with C_{SMJ_trans} and C_{SPJ_trans} because both the latency and the partial tuple size are typi-

cally small.

Next, for the processing costs, let us subtract the processing cost of SPJ from the processing cost of SMJ.

$$C_{SMJ\_proc} - C_{SPJ\_proc} = \lambda_1(C_{update\_W_1} + C^*_{probe\_W_2} + C^*_{probe\_W_1}) + \lambda_2(C^*_{probe\_V'_1} + C^*_{update\_V'_1})$$

Note that during this subtraction $C_{update\_W'_1}$ and $C_{update\_V'_1}$ cancel out because W'_1 and V'_1 have the same number of tuples.

From the result of this subtraction, we see that $C_{SMJ\_proc} > C_{SPJ\_proc}$ because all the cost terms are positive numbers. In the formula above, the values of $C^*_{probe\_W_1}$, $C^*_{probe\_W_2}$ and $C^*_{probe\_V'_1}$ are very small (e.g., $C^*_{probe\_W_1} \approx 0.2msec$ in the case of our experiments in Section 5.2) since they are the costs of finding the first or the only matching tuple in W_1, W_2, and V'_1, respectively. Besides, the values are constants with respect to the window sizes. That is, if a hashing technique is used for the probing, the costs are independent of the window sizes and, even if a linear scanning is used, the costs are still near constants because, the way the two join algorithms work, the matching tuple is at or near the beginning of the windows at the time of the probing. The value of $C_{update\_W_1}$ is small as well because it is the cost of inserting a tuple into the window buffer and removing any expired tuples. The value is also a constant with respect to the window size. From this analysis of the cost terms, we have $C_{SMJ\_proc} - C_{SPJ\_proc} \approx \lambda_2 * C^*_{update\_V'_1}$

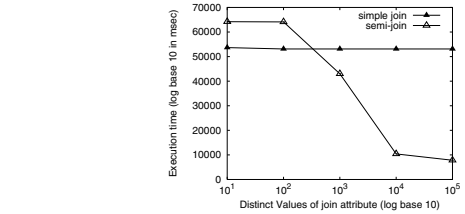
The value of $C^*_{update\_V'_1}$ depends on the probability of finding partial tuples in V'_1 (i.e., $(1-p_2)*p_1$). Similarly to the analysis of transmission costs, we have two cases. If d_V is sufficiently large relative to W_i (i.e., $p_i \approx 0$), then $C^*_{update\_V'_1} \approx 0$. If W_i is sufficiently large relative to d_V (i.e., $p_i \approx 1$), then $C^*_{update\_V'_1} \approx 0$ as well. In both cases, $C^*_{update\_V'_1} \approx 0$, thus the difference between the processing costs of SMJ and SPJ is very small.

In summary, taking both transmission time and processing time into consideration, SMJ is preferred when the number of distinct values of join attribute is sufficiently large (hence, the join selectivity is low) and/or the partial tuple size is much smaller than the full tuple size.

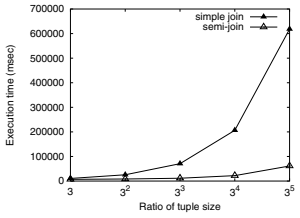
We believe these two conditions are common in many applications. For example, in the network traffic monitoring application (Example 1), common packet sizes are 44, 552, 576 and 1500 bytes[20], while a partial tuple including the timestamp and the join attribute (i.e., destination IP address) is only about 12 bytes (source: TCP extension specifications [1]). In the news article monitoring application (Example 2), the size of each news article tagged with a set of weighted keywords may range from 1000 bytes to 10 megabytes while a partial tuple may have few hundred bytes (including timestamp and the set of weighted keywords).

5.2 Experiments

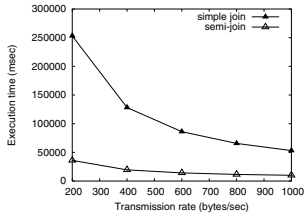
We have built a program that implements the SPJ and SMJ algorithms. In the program, transmission cost is simulated by calling a *sleep* function with the delay computed using the formulas (1) and (2). Only binary join is supported currently, and windows are tuple-based. The program takes as inputs the data streams generated using our data generator, the join window sizes (i.e., number of tuples), and the join algorithm type (SPJ or SMJ). It then returns the total time (in milliseconds) of executing the join algorithm on the input data streams. The program is written in Java 2 SDK 1.4.2, and runs on a Linux PC with Pentium IV 1.6GHz processor and 512MB RAM.



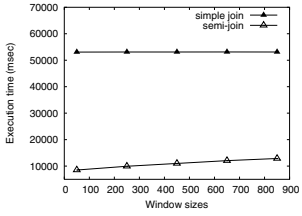
(a) Varying number of distinct join attribute values (d_V).



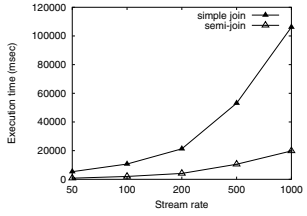
(b) Varying tuple size ratio ($\frac{d_F}{d_P}$).



(c) Varying transmission rate ($\lambda_{1,2}$).



(d) Varying window sizes ($|W_1|, |W_2|$).



(e) Varying stream rate (λ_1).

Figure 2. Execution times of SPJ and SMJ.

The objective of our experiments is to compare the two join algorithms in light of the analysis presented in Section 5.1. For this, we perform five sets of experiments by varying each of the following five parameters: number of distinct join attribute values, ratio of full tuple size and partial tuple size, transmission rate, window size, and stream rate. While one parameter is varied, the others are fixed to the following default values: window sizes ($|W_1| = 500$, $|W_2| = 500$ tuples), stream rates ($\lambda_1 = 500$, $\lambda_2 = 500$ tuples/second), transmission rate ($\lambda_{1,2} = 1000$ bytes/second), latency ($l = 5$ msec), number of distinct join values ($d_V = 10000$), and tuple sizes ($d_F = 20$, $d_P = 2$ attributes, where each attribute size equals 5 bytes).

Fig. 2 shows results of the five experiments. We have implemented the window-based join with both nested loop

join and hash join. Due to space limit, however, we show only the results of hash joins. The results of nested loop joins show the same trends except that the costs are higher. The figure shows that the performance advantage of SMJ increases with the increase of the number of distinct values and the ratio of tuple sizes. These confirm the analysis results from the cost models (in Section 5.1).

6 Conclusion and Future Work

In this paper, we addressed the problem of window-based join processing on distributed data streams. Specifically, we proposed two distributed join algorithms: simple join and semijoin-based join. These join algorithms need all join windows to be either exported to or mirrored at a remote query processing site. Additionally, the semijoin-based algorithm employs the idea of sending a partial tuple to the query processing site. We developed cost models of the two join algorithms using total execution time as the cost metric, and through analysis using the cost models and a set of experiments, concluded that semijoin-based join is typically less costly than the simple join. To our knowledge, this is the first work done to use semijoins in distributed window stream join processing. We believe this work sets a foundation for further research in distributed stream join query optimization.

There are a number of future works in progress or in plan. First, we are currently working to improve the efficiency of join algorithms, using such ideas as shipping a block of tuples instead of individual tuples and reducing the size of mirror windows with a bitmap structure. Second, in this paper we considered only two-way joins and only one of many possible semijoin reduction programs. We plan to build a distributed join query optimizer which considers all possible semijoin reduction programs for multi-way joins and chooses the best one. Third, we plan to conduct experiments in a real distributed environment with real data streams. Fourth, we assumed there is no transmission delay between nodes and no processing overload in a node. We will relax these assumptions and study how to handle those problems through, for example, approximating the tuples missed (due to delay) or lost (due to overload).

Acknowledgments

This research has been supported by US National Science Foundation through Grant No. IIS-0415023. We thank the anonymous reviewers for their constructive comments.

References

[1] The TCP extentions (RFC1323) published by the Internet Engineering Task Force: http://www.ietf.org/rfc/rfc1323.txt?number=1323.

[2] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

[3] P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM J. Comput.*, 10(4):751–771, 1981.

[4] M.-S. Chen and P. S. Yu. Combining join and semi-join operations for distributed query processing. *IEEE TKDE*, 5(3):534–542, 1993.

[5] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, page 23, 2003.

[6] G. Cormode, S. Muthukrishnan, and W. Zhuang. What's different: Distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In *ICDE*, page 57, 2006.

[7] A. Das, S. Ganguly, M. N. Garofalakis, and R. Rastogi. Distributed set expression cardinality estimation. In *VLDB*, page 312, 2004.

[8] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *SPAA*, page 63, 2002.

[9] M. Gorawski and P. Marks. Fault-tolerant distributed stream processing system. In *DEXA Workshops*, page 395, 2006.

[10] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, page 341, 2003.

[11] R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD*, page 289, 2006.

[12] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, page 783, 2005.

[13] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *ICAC*, page 3, 2005.

[14] Z. Li and K. A. Ross. Perf join: An alternative to two-way semijoin and bloomjoin. In *CIKM*, page 137, 1995.

[15] H. Oh'Uchi, T. Miura, and I. Shioya. Querying on news stream by using random projection. *ICITA*, 01:185–190, 2005.

[16] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, page 563, 2003.

[17] M. T. Ozsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. 1999.

[18] S. Seshadri, V. Kumar, and B. F. Cooper. Optimizing multiple queries in distributed data stream systems. In *ICDE Workshops*, page 25, 2006.

[19] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *SIGMOD*, page 301, 2006.

[20] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, Nov./Dec. 1997.

[21] C. Wang, A. L. P. Chen, and S.-C. Shyu. A parallel execution method for minimizing distributed query response time. *IEEE TPDS*, 3(3):325–333, 1992.

[22] T. Xia, C. Jin, X. Zhou, and A. Zhou. Filtering duplicate items over distributed data streams. In *WAIM*, page 779, 2005.

[23] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. B. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, page 775, 2006.

[24] D. Zhang, J. Li, K. Kimeli, and W. Wang. Sliding window based multi-join algorithms over distributed data streams. In *ICDE*, page 139, 2006.

[25] D. Zhang, J. Li, W. Wang, L. Guo, and C. Ai. Processing frequent items over distributed data streams. In *APWeb*, page 523, 2005.

[26] Y. Zhou, Y. Yan, B. C. Ooi, K.-L. Tan, and A. Zhou. Optimizing continuous multijoin queries over distributed streams. In *CIKM*, page 221, 2005.