Distributed Adaptive Windowed Stream Join Processing

Tri Minh Tran
Department of Computer Science
University of Vermont
Burlington VT 05405, USA
ttran@cems.uvm.edu

Byung Suk Lee
Department of Computer Science
University of Vermont
Burlington VT 05405, USA
bslee@cems.uvm.edu

Abstract

This paper presents an adaptive framework for processing a window-based multi-way join query over distributed data streams. The framework integrates distributed plan modification and distributed plan migration within the same scope by using a building block called the *node operator set (NOS)*. An NOS is housed in each node that participates in the join execution, and specifies the set of atomic operations to be performed locally at the host node to execute its share of the global execution plan. The plan modification and migration techniques in this paper are presented for the case of updating the NOSs centralized at a single node and the case of updating them distributed at each node. The plan modification is triggered by the change of stream statistics and adjusts the join execution order and placement greedily to satisfy a cost invariant. The plan migration uses what is called the *distributed track strategy* to accelerate the migration of window extents to new nodes. The migration of all window extents is synchronized. Experiments confirm the effectiveness of the developed adaptive framework on reducing the join execution cost and indicate a small additional adaptation-overhead for distributing the NOS update.

Keywords: multi-way join query, adaptive query processing, data streams, distributed database management systems.

1 Introduction

Distributed data stream processing (Amini, Jain, Sehgal, Silber, and Verscheure 2006; Cormode, Muthukrishnan, and Zhuang 2006; Das, Ganguly, Garofalakis, and Rastogi 2004; Kumar, Cooper, and Schwan 2005; Kumar, Cooper, Cai, Eisenhauer, and Schwan 2005; Olston, Jiang, and Widom 2003; Seshadri, Kumar, and Cooper 2006; Sharfman, Schuster, and Keren 2006) is a fast growing research area in the data stream field. The driving force behind this growth is the widely deployed and utilized diverse distributed computing environments such as the telecommunication networks, web, sensor networks, and P2P networks as well as the evermore performance-demanding intelligence and monitoring applications in various sectors of the society.

In this paper, we focus on multi-way window-based stream join query which is an important class of queries in distributed stream applications. For example, in network packet monitoring, the network administrator may want to monitor the traffic of data packets passing though different routers with the objective of finding packets with the same destination IP address. For this task, a distributed stream join query is needed to join the streams of packets from those routers. As another example, in building-monitoring using sensor networks, one may want to keep track of the temperature, humidity, and light intensity measured by sensors in a room. The sensor readings of each measurement type are sent to their respective sinks as a stream. The monitoring task in each room can be specified as a distributed stream join query that joins on the same room id from three sensor reading streams. Similar distributed join queries are also needed in many other stream applications such as financial stock ticker analysis, telephone call monitoring, and news article filtering.

An important aspect of query processing today is the adaptivity, that is, adjusting the query execution plan adaptively to the changing data profile and system environment. In light of data stream query processing, the fluctuations of stream statistics (e.g., stream rates, join selectivity) or available system resources (e.g., memory, CPU time) are the changes to adapt to. This paper focuses on the former, i.e., stream statistics.

To the best of our knowledge, all existing research on adaptive stream join processing have been done in the centralized environment (Babu, Motwani, Munagala, Nishizawa, and Widom 2004; Babu, Munagala, Widom, and Motwani 2005; Zhu, Rundensteiner, and Heineman 2004) and none in the distributed environment. In the distributed environment, a different query processing model is needed because some or all join steps are performed at different nodes across the network and the communication overhead for these join steps should be taken into consideration in query execution planning, and thus the solutions developed in the centralized environment are not applicable.

Moreover, there is a division in the scope of the existing work. Adaptive query processing framework encompasses query plan *modification* and query plan *migration*. Query plan modification involves the process of updating current execution plan to a new, better plan, and query plan migration handles the switch from the current execution plan to the new plan. As far as we know, however, there does not exist any work done on adaptive stream query processing with both in one scope. All the existing work address either the query plan modification (Babu, Motwani, Munagala, Nishizawa, and Widom 2004; Babu, Munagala, Widom, and Motwani 2005) or the query plan migration (Zhu, Rundensteiner, and Heineman 2004), not to mention they are not distributed. This disconnection naturally misses out the interaction between the two key aspects of adaptive query processing.

This paper aims to advance the state of the art by providing a solution to the distributed plan modification and migration problem for executing stream join queries adaptively within the same framework as the stream statistics change in a distributed environment.

The adaptation of query processing in this paper is triggered by an event defined as a significant change of stream statistics such as stream rate, join selectivity, and tuple size. One challenge in achieving this adaptivity in a *distributed* environment comes from the fact that the query execution plan specifies only the steps for executing a query but not specifically what each node needs to do. Thus, each node has to make its own local decision on what part of the distributed global plan it needs to execute, when to adjust its own part, and how to adjust it.

Our approach to meeting this challenge is to use an abstract data type called the *node operator set (NOS)*. An instance of the NOS is stored in each node and used to specify what each node needs to do toward generating the global query result. Specifically, the elements of an NOS are the basic operators executed locally at the node as part of a distributed join execution plan. The operators include a one-way join operator, a window-update operator, and a tuple-shipping operator, and collectively represent the state of the node during the event-driven distributed join execution. Each node has a set of operators like these for executing its part of all execution plans.

Using the notion of NOS as a common building block, we develop an integrative solution that covers both distributed query plan modification and distributed query plan migration in an adaptive framework. The distributed plan modification is centered on updating the NOSs housed at individual nodes. The update of NOSs can be done either at a central node or distributed at individual nodes. In the centralized update, one node (usually the query site) maintains and updates the NOSs and sends updated NOSs to the nodes housing them (referred to as the *affected nodes*). In the distributed update, individual nodes communicate with their neighboring nodes and update their own NOSs locally if necessary. We address both update approaches in this paper. Both approaches are founded on the same greedy algorithm for plan generation, and thus always generate the same plan.

The distributed plan migration is executed at each node affected by the plan modification. The key requirement is to switch the current NOS to a new NOS locally without duplicating or missing any output tuple globally. The challenge comes from the fact that the switch-over is distributed, that is, when the plan changes, windows need to move to new nodes without disrupting the join execution. We call it the *distributed track strategy*. The approach used is to run the old NOSs and new NOSs together until the new NOSs start generating the query output using full window contents and then have all affected nodes migrate to the new plan synchronously. In order to avoid the delay incurred until the new NOSs are ready, any window that needs to move to a different node during the switch-over is copied over as soon as possible so that the new NOSs do not have to start with empty windows. To enable the synchronized migration, we use a somewhat simple *two-phase migration* protocol.

A set of experiments has been done to evaluate how effective the proposed adaptive framework is compared with a non-adaptive one and to observe the extra adaptation-overhead of the distributed NOS-update approach (as opposed to the central) in return for the distributivity. The results show a significant reduction in the execution cost in the adaptive case despite substantial fluctuations of the stream rate and show an arguably small additional overhead of the distributed update approach.

Main contributions of this paper include, first, introducing the notion of a node operator set (NOS) which is a set of operators stored in each node to execute a join execution plan; second, proposing distributed plan modification and migration techniques with both centralized and distributed NOS update approaches; third, studying the performance of the adaptive framework through a set of experiments. To the best of our knowledge, this is the first work done to address the integrated problem of plan modification and plan migration for adaptively processing distributed windowed stream joins.

The rest of this paper is organized as follows. Section 2 presents the processing model and the cost model of distributed windowed stream join queries. Section 3 presents a basic plan generation algorithm for finding an efficient join execution plan. Section 4 introduces the concept of the node operator set and discusses the proposed distributed plan modification and migration techniques based on centralized and distributed NOS updates. Section 5 evaluates the performance of the proposed adaptive framework. Section 6 discusses related work. Section 7 concludes the paper and suggests future work.

2 Preliminaries

This section provides two models associated with distributed stream join processing: query processing model and cost model.

2.1 Distributed stream join processing model

Queries are distributed window-based multi-way stream joins. A stream S_i is a sequence of tuples arriving in order. Each tuple in the stream has a timestamp, *ts*, and a join attribute, *J*, as part of the schema. In a distributed environment, a set of nodes (or sites) $N_1, N_2, \ldots, N_n$ are connected through a communication network. The model assumes only one stream per node, assuming all local processing (e.g., selection, projection) has been done at each node. It also assumes that the timestamp is synchronized across all nodes in the network.

All joins are window-based. For a given multi-way join, $S_1 \bowtie S_2 \bowtie \ldots \bowtie S_m$, there is a window W_i on each stream S_i ($i = 1, 2, \ldots, m$). This join (Golab and Ozsu 2003) is processed as follows. For each new tuple s_i arriving in S_i, probe the other $m-1$ windows in sequence and output the matching tuples, and then insert s_i into W_i and remove any expired tuples from W_i. Any type of join condition – equijoin or non-equijoin – is supported in our model. Note that, in a distributed environment, the join processing model needs to consider other issues as well, such as the node synchronization over network latency and the batch processing of buffered tuples. These issues have already been addressed in our prior work on distributed stream join processing (Tran and Lee 2010), and thus are not addressed in this paper.

A distributed multi-way join execution plan is characterized by join ordering, join placement, and join method (Ceri and Pelagatti 1984). First, *join ordering* determines the sequence of joins in a multi-way join. We assume *linear* ordering (Viglas, Naughton, and Burger 2003) which has proven to be effective in streaming scenarios (Gedik, Wu, Yu, and Liu 2007; Zhou, Yan, Yu, and Zhou 2006; Babu, Motwani, Munagala, Nishizawa, andWidom 2004; Babu, Munagala, Widom, and Motwani 2005; Srivastava, Munagala, and Widom 2005). For a given m-way join query, linear ordering determines separate join sequences for each stream (which we call a *head stream*), thus m join sequences altogether. The join sequence associated with a head stream S_i is defined as an ordered set O_i of the other $m-1$ streams that are joined in sequence for each new tuple s_i arriving in S_i. That is, $O_i = [S_{i_1}, S_{i_2}, \ldots, S_{i_{m-1}}]$ which is one of the $(m-1)!$ possible orderings of $S_1, \ldots, S_{i-1}, S_{i+1}, \ldots$ and S_m.

Given such a join sequence (or ordering) O_i, a new tuple s_i arriving on the head stream S_i is used to probe the window W_{i_1} on the first stream S_{i_1} in O_i to find matching tuples; if matching tuples are found, then each join output tuple is used to probe the window W_{i_2} on the next stream S_{i_2} in O_i, and the same process repeats until the window $W_{i_{m-1}}$ on the last stream $S_{i_{m-1}}$ in O_i or no matching tuple is found. In this paper we refer to the probing of a window W_{i_k} as a *one-way join* from the output stream of matching tuples from the previous probing (denoted as $S^i_{i_{k-1}}$) to S_{i_k}. (The head stream S_i is equivalent to S_{i_0} here.) Thus, given a linear join sequence O_i, there are $m-1$ one-way joins executed to generate the output for each set of new arrival tuples.

Second, for each join in the sequence, *join placement* determines the node (or site) at which the join is processed. Specifically, a one-way join from S_s at N_s to S_d at N_d can be processed either at N_s by shipping the window W_d on the stream S_d to N_s (called the *source* placement) or at N_d by shipping the new tuples s_s arriving at S_s to N_d (called the *destination* placement). Note that there may be delay or loss of tuple, and thus the join output might be not exact. However, in the scope of this paper, we assume no delay or tuple loss. Since there are $m-1$ one-way joins in a join sequence O_i, there are $m-1$ join placements, one for each one-way join. Thus, for each join ordering O_i, there is an associated sequence of join placement values. We call it the *join placement sequence* of O_i and denote it as $P_i = [p_{i_1}, p_{i_2}, \ldots, p_{i_{m-1}}]$ where $p_{i_k} = \{0,1\}$. Here, p_{i_k} indicates whether the one-way join from $S^i_{i_{k-1}}$ to S_{i_k} is processed at the source node (i.e., $p_{i_k} = 0$) or the destination node (i.e., $p_{i_k} = 1$).

Third, for the join method, we consider only the nested loop join in this work, as considering other join methods only requires expanding the search space for finding an optimal plan and does not affect the plan modification and migration algorithms. Thus, only join ordering and join placement are the factors considered in the join execution plan.

To summarize, the execution plan of a multi-way stream join $S_1 \bowtie \ldots \bowtie S_m$ is defined as follows.

Definition 1: [Join execution plan (JEP)] The plan of an *m*-way join query is a set of *m per-stream execution plans (SEPs)*, that is $\{SEP_1, SEP_2, \ldots, SEP_m\}$. Here each SEP_i $(i = 1,2,\ldots,m)$ is headed by stream S_i and is defined as the pair (O_i,P_i) where $(O_i,P_i) = ([S_{i_1}, S_{i_2}, \ldots, S_{i_{m-1}}], [p_{i_1}, p_{i_2}, \ldots, p_{i_{m-1}}])$.

Example 1: Figure 1 illustrates one possible JEP of a three-way join query $S_1 \bowtie S_2 \bowtie S_3$. It consists of the following three SEPs, one for each head stream S_1, S_2, and S_3.

$SEP_1 = (O_1, P_1) = ([S_2, S_3], [1,1])$

$SEP_2 = (O_2, P_2) = ([S_1, S_3], [0,1])$

$SEP_3 = (O_3, P_3) = ([S_1, S_2], [0,0])$

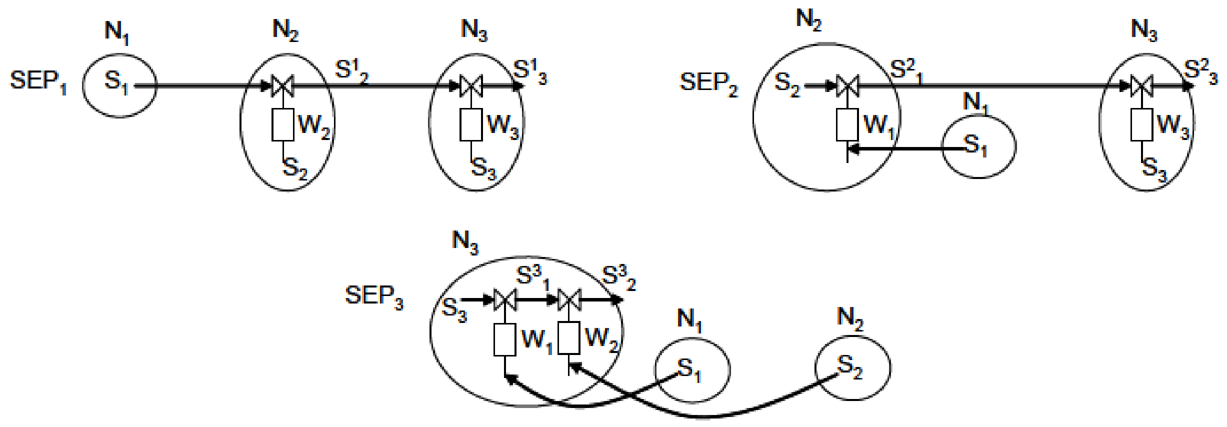


Figure 1. An example join execution plan of a three-way join.

2.2 Distributed stream join cost model

Table 1 summarizes the notations used to build the cost model.

Since data streams arrive continuously and unboundedly, we use the *unit-time cost* model proposed by Kang et al. (Kang, Naughton, and Viglas 2003). That is, the cost is the time it takes to process the tuples arriving in unit time. Based on this, at any point in time, the execution cost of SEP_i is modeled as the sum of the total processing cost and the total transmission cost per unit time over all one-way join steps. That is, the cost of SEP_i, C_{SEP_i}, is computed as the sum of the costs of $m\text{-}1$ one-way joins: $C_{SEP_i} = \sum_{k=1}^{m-1} C^i_{i_{k-1},i_k}$.

The one-way join cost $C^i_{i_{k-1},i_k}$ is formulated as follows. On one hand, if the one-way join is executed at the source $N_{i_{k-1}}$ (i.e., $p_{i_k} = 0$), then the cost comprises the cost of shipping r_{i_k} tuples from N_{i_k} to $N_{i_{k-1}}$ per unit time, the cost of updating W_{i_k} at $N_{i_{k-1}}$ per unit time, and the cost of probing W_{i_k} at $N_{i_{k-1}}$ per unit time. That is,

$$C^i_{i_{k-1},i_k} \,|[p_{i_k} = 0] = T(r_{i_k} \times t_{i_k}) + r_{i_k} \times c_u + r^i_{i_{k-1}} \times c_p \times w_{i_k} \qquad (1)$$

Notation	Meaning (*i = 1,..,m* and *k = 1,..,m-1*)
Join input streams and statistics	
r_i	The stream rate of S_i.
f_i	The join attribute selectivity factor of S_i, i.e., the average fraction of S_i tuples with the same join attribute value.
t_i	The size of a tuple in S_i.
w_i	The number of tuples in the window W_i of S_i.
S_{i_k}	The kth stream in SEP_i
r_{i_k}	The stream rate of S_{i_k}
Join output streams and statistics	
$S^i_{i_k}$	The output stream of the one-way join S_{i_k}. ($S^i_{i_0} = S_i$)
$r^i_{i_k}$	The stream rate of S_{i_k}. ($r^i_{i_0} = r_i$)
$f^i_{i_k}$	The join attribute selectivity factor of S_{i_k}. ($f^i_{i_0} = f_i$)
$t^i_{i_k}$	The size of a tuple in S_{i_k}. ($t^i_{i_0} = t_i$)
$C^i_{i_{k-1},i_k}$	The cost of a one-way join from $S^i_{i_{k-1}}$ to S_{i_k}.
Join cost parameters	
c_p	Per-tuple window-probing cost.
c_u	Per-tuple window-update cost.
c_l	The latency of communication link in the network.
c_r	The transmission rate of communication link in the network.

Table 1. Notations used in the cost model.

where $T(.)$ is a function $T(x) = c_l + x/c_r$ used as a model of the cost of transmitting x bytes of data between two nodes and the value of $r^i_{i_{k-1}}$ is computed $r^i_{i_{k-1}} = r^i_{i_{k-2}} \times min(f^i_{i_{k-2}}, f_{i_{k-1}}) \times w_{i_{k-1}}$ where $f^i_{i_{k-2}}$ is estimated as the larger value between the two selectivity factors of the two streams that are joined, that is, as $max(f^i_{i_{k-3}}, f_{i_{k-2}})$. On the other hand, if the join is executed at the destination N_{i_k} (i.e., $p_{i_k} = 1$), then the cost comprises the cost of shipping $r^i_{i_{k-1}}$ tuples from $N_{i_{k-1}}$ to N_{i_k} per unit time, the cost of updating W_{i_k} at N_{i_k} per unit time, and the cost of probing W_{i_k} at N_{i_k} per unit time. That is,

$$C^i_{i_{k-1},i_k} \,|[p_{i_k} = 1] = T(r^i_{i_{k-1}} \times t^i_{i_{k-1}}) + r_{i_k} \times c_u + r^i_{i_{k-1}} \times c_p \times w_{i_k} \qquad (2)$$

where $t^i_{i_{k-1}} = t^i_{i_{k-2}} + t_{i_{k-1}}$. Given these two cost formulas (Equations 1 and 2), the join placement value p_{i_k} is determined depending on which one incurs the lower cost. In other words, if $C^i_{i_{k-1},i_k} \,|[p_{i_k} = 0] \leq C^i_{i_{k-1},i_k} \,|[p_{i_k} = 1]$, then $p_{i_k} = 0$ and otherwise $p_{i_k} = 1$. Thus, the one-way join cost $C^i_{i_{k-1},i_k}$ is computed as

$$C^i_{i_{k-1},i_k} = min(C^i_{i_{k-1},i_k}|[p_{i_k} = 0], C^i_{i_{k-1},i_k}|[p_{i_k} = 1]) \quad (3)$$

3 Basic Plan Generation Algorithm

Algorithm 1 is a plan generation algorithm which finds SEP_i associated with each head stream S_i. The algorithm is used in two stages in the adaptive query processing framework. The first stage is to generate an initial plan which is disseminated to all participating nodes in the network. The second stage varies depending on the NOS update mechanism (either centralized or distributed, to be discussed in Section 4.2) – the entire algorithm is used as is if centralized whereas only the greedy property is used if distributed.

```
Input: Streams $\{S_1, S_2, ..., S_m\}$ and join graph
Output: Join execution plan $\{(O_1, P_1), (O_2, P_2), ..., (O_m, P_m)\}$
1  begin
2      foreach stream $S_i$ $(i = 1, 2, ..., m)$ do
3          $O_i = \emptyset$;
4          $P_i = \emptyset$;
5          $Temp = \{S_1, S_2, \ldots, S_m\} - S_i$;
6          while $Temp \neq \emptyset$ do
7              Find the stream $S_{i_k}$ in $Temp$ with the smallest associated $C^i_{i_{k-1},i_k}$ computed using Equation 3;
8              Insert $S_{i_k}$ at the end of $O_i$;
9              Determine the join placement value $p_{i_k}$ by comparing $C^i_{i_{k-1},i_k}|[p_{i_k} = 0]$ (Equation 1) and
               $C^i_{i_{k-1},i_k}|[p_{i_k} = 1]$ (Equation 2);
10             Insert $p_{i_k}$ at the end of $P_i$;
11             $Temp = Temp - S_{i_k}$;
12         end
13     end
14 end
```

Algorithm 1: Basic plan generation algorithm.

The algorithm takes the join graph and information about the participating streams as the input and returns the set of SEPs with each input stream as the head stream. For each head stream S_i, O_i and P_i are constructed together using a greedy approach to add the next stream and the next join placement value in each step. The next stream picked in each step is the one S_{i_k} that has the lowest one-way join cost $C^i_{i_{k-1},i_k}$ among streams $S_{i_k}, \ldots, S_{i_{m-1}}$.

$$C^i_{i_{k-1},i_k} \leq C^i_{i_{k-1},i_p} \text{ for all } p \text{ s.t. } 1 \leq k < p \leq m-1 \quad (4)$$

Equation 4 is an invariant condition that should be satisfied throughout the continuous stream join processing despite the change of cost due to the fluctuation of stream statistics. The objective of the adaptive framework in this paper is to maintain the invariant condition online in a distributed environment. The next section presents the techniques developed to achieve it.

Note that the number of possible join execution plans is O(n!) (n is the number of joins in the query) which is exponential. In general, an optimal join plan generation is an intractable problem (Aho, Sagiv and Ullman 1979). For instance, the best known algorithm using dynamic programming approach (e.g., System R) takes $O(n2^{n-1})$. In contrast, the greedy approach used in this paper finds an efficient (but not necessarily optimal) plan in polynomial time $O(n^2)$ in the worst case. Note that this is the same whether the NOS update mechanism is centralized or distributed (to be discussed in Section 4.2), as both are based on the same greedy property.

4 Distributed Adaptive Stream Join Processing

In this section, NOS is explained in Section 4.1 with a focus on its formal definition and the construction algorithm. Section 4.2 describes the distributed plan modification with the centralized NOS-update strategy (Section 4.2.1) and the distributed NOS-update strategy (Section 4.2.2), respectively. Section 4.3 discusses the distributed plan migration strategy.

4.1 Node operator set

Given a JEP, each node extracts the set of relevant operations needed to execute the JEP. There are three possible operators needed at each node, as summarized in Table 2. *Join*(S_i ,S_j) is the one-way join from S_i to S_j upon the arrival of a set of tuples of S_i. *Update*(S_i) is the update of window W_i upon the arrival of a set of tuples of S_i. *Ship*(S_i ,N_k) is the shipment of a set of tuples of S_i to the node N_k. Thus, given the JEP of an *m*-way join, the NOS of a node N_k ($k = 1,2,...,m$) is composed of the sets of operations extracted from every SEP_i ($i = 1,2,...,m$) of the JEP. We denote the set extracted from SEP_i in N_k as NOS^i_k.

Notation	Meaning
join(S_i ,S_j)	Perform one-way join from S_i to S_j .
update(S_i)	Update the window on S_i.
ship(S_i ,N_k)	Ship tuples of S_i to node N_k.

Table 2. Notations of operations.

Definition 2: [Node operator set of N_k (NOS_k)] $NOS_k = NOS^1_k \; U \ldots \; U \; NOS^m_k$ where NOS^i_k is the set of operators needed at the node N_k to execute SEP_i.

Example 2: Continuing from Example 1, the NOS of each node is as follows.

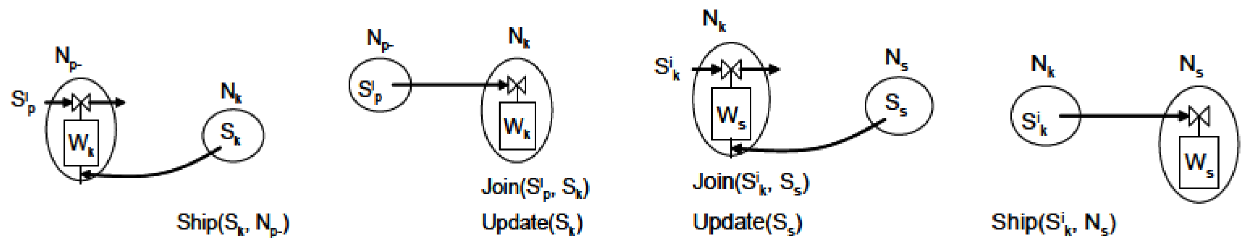
$NOS_1 = NOS^1{}_1 \cup NOS^2{}_1 \cup NOS^3{}_1$
$= \{ship(S_1, N_2)\} \cup \{ship(S_1, N_2)\} \cup \{ship(S_1, N_3)\}$
$= \{ship(S_1, N_2), ship(S_1, N_3)\}$

$NOS_2 = NOS^2{}_1 \cup NOS^2{}_2 \cup NOS^3{}_2$
$= \{join(S_1, S_2), update(S_2), ship(S^1{}_2, N_3), update(S_1), join(S_2, S_1), ship(S^2{}_1, N_3), ship(S_2, N_3)\}$

$NOS_3 = NOS^1{}_3 \cup NOS^2{}_3 \cup NOS^3{}_3$
$= \{join(S^1{}_2, S_3), update(S_3), ship(S^1{}_3\ N_0), join(S^2{}_1, S_3), ship(S^2{}_3, N_0), join(S_3, S_1), join(S^3{}_1, S_2), update(S_1), update(S_2), ship(S^3{}_2\ N_0)\}$

NOS construction algorithm

In order to build the NOS for any give node, the algorithm constructs $NOS^i{}_k$ from each SEP_i and then combines them to obtain NOS_k. Specifically, it locates the stream S_k in O_i and constructs $NOS^i{}_k$ based on the join placement values of the one-way joins to S_k and its succeeding stream, respectively. The details of the algorithm for a given SEP_i are as follows.



(a) $p_k = 0$.   (b) $p_k = 1$.   (c) $p_s = 0$.   (d) $p_s = 1$.

Figure 2. Construction of operators based on join placement.

First, consider the one-way join to S_k in O_i. Let S_p and S_s be respectively the streams immediately preceding and immediately succeeding S_k in O_i. Let N_{P-} be the node at which $S^i{}_p$ is generated. Note that N_{P-} can be any node, not necessarily N_P, because the placement of the one-way join to S_p may not be at N_p. Depending on the join placement (p_k) of the one-way join to S_k, there are two alternative ways to construct the NOS operators. If $p_k = 0$ (Figure 2a), it means the one-way join to S_k is executed at N_{p-}. In this case, N_k needs to ship tuples to N_{p-} for the join execution there and, thus, an operator $ship(S_k, N_{p-})$ is added to $NOS^i{}_k$. If $p_k = 1$ (Figure 2b), it means the one-way join to S_k is executed at N_k. In this case, upon receiving tuples of $S^i{}_p$, N_k joins them with tuples in W_k on S_k and, thus, the operator $join(S^i{}_p, S_k)$ is added to $NOS^i{}_k$; additionally, W_k needs to be maintained at N_k and, thus, the operator $update(S_k)$ is added to $NOS^i{}_k$ as well.

Next, consider the one-way join to S_s in O_i. Like the one-way join to S_k, there are two alternative ways to construct the operators. If $p_s = 0$ (Figure 2c), it means the one-way join to S_s is executed at N_k. In this case, N_s ships tuples to N_k and the join is executed at N_k and, thus, $join(S^i_k, S_s)$ and $update(S_s)$ are added to NOS^i_k. If $p_s = 1$ (Figure 2d), it means the one-way join to S_s is executed at node N_s. In this case, N_k ships the tuples of S^i_k to N_s and, thus, the operator $ship(S^i_k, N_s)$ is added to NOS^i_k; additionally, if S_k is the last stream (Si_{m-1}) in the join sequence, then the ship operator $ship(S^i_k, N_0)$ is added to NOS^i_k as well.

Note that if $p_s = 0$, that is, the one-way join to S_s is executed at N_k, then we need to consider the one-way join to the stream succeeding S_s, denoted as S_{s+1}, as well. The reason for this is that if $p_{s+1} = 0$ then this one-way join is executed at N_k as well and, thus, $join(S^i_s, S_{s+1})$ and $update(S_{s+1})$ also need to be added to NOS^i_k. This step proceeds with all succeeding streams (S_{s+j}, $j = 0,1,\ldots,l \leq m\text{-}k\text{-}1$) until the first one-way join to a stream S_{s+l} with $p_{s+l} = 1$ is encountered or the last stream (Si_{m-1}) is encountered.

Given the NOSs constructed in that manner, the node N_k executes the operators in NOS_k upon the arrival of a set of tuples. The set of tuples comes from either the local node N_k or another node, and belongs to either the local stream S_k or the output stream S^i_p. Each set of arrival tuples is associated with the stream identifier (streamID) of either S_k or S^i_p, depending on where it is coming from. Upon the arrival of a set of tuples, the query executor executes all operators in NOS_k whose first argument matches the streamID.

4.2 Distributed plan modication

When the stream statistics at a node change significantly enough, any of the current SEPs (i.e., SEP_1, SEP_2, ...,SEP_m) may become less efficient and in this case a new SEP needs to be generated to replace the current one for those affected SEP_i ($i = 1,2,\ldots,m$). With the NOS in use for query processing, the plan modification involves updating the NOS at each affected node. In this section we discuss the centralized and the distributed approaches to making these updates.

As mentioned in Section 3, the invariant cost-condition (Equation 4) should always be satisfied by every SEP. Thus, the NOS update is essentially to adjust the NOS of the affected node so that all SEPs satisfy the cost condition again. The adjustment is triggered through a threshold mechanism, that is, when the change in the execution cost at a node exceeds the threshold. The threshold value is a system parameter and can be used to control the plan-modification frequency of the system.

4.2.1 Centralized node-operator-set update

The centralized NOS update is triggered whenever any of the participating nodes reports a change of stream statistics above the threshold. Once triggered, it generates a new JEP (comprising SEPs) based on the new stream statistics provided by the reporting nodes. For this, Algorithm 1 is used. It then extracts from the new SEPs the new NOS_k for each node N_k and if the new NOS_k is different from the old one then ships it to N_k to replace the old one. Algorithm 2 summarizes these steps taken after the NOS update is triggered.

```
begin
    A centralized node $N_q$ runs Algorithm 1 to generate a new JEP (i.e., a set of SEPs);
    foreach node $N_i$ do
        $N_q$ generates a new $NOS_i$ based on the new SEPs;
        if the new $NOS_i$ is different from the old $NOS_i$ then
            $N_q$ ships the new $NOS_i$ to $N_i$ for an update;
        end
    end
end
```

Algorithm 2: Centralized NOS update.

The computation cost of this algorithm in terms of the number of messages sent is *O(n)* where *n* is the number of nodes. This is the worst case cost, in which every node needs to be updated with a new NOS and thus N_q sends a new NOS to every node. Note that, in the algorithm, only those nodes whose old NOSs are different from the new NOSs receive the new NOSs from N_q. They are the only affected nodes. An affected node needs to switch from the old NOS to the new NOS after receiving a new NOS, and this switch-over should be handled online while not disrupting the generation of the correct output. This technique is presented in Section 4.3.

In the centralized update approach, the information about the stream statistics as well as the NOSs of participating nodes is maintained all at the centralized node, and thus there is no cost of gathering the information from participating nodes; each participating node only needs to send a message to report the change of its stream statistics (so the NOS update is triggered) whenever the change exceeds the threshold. However, there always exists a possibility that the central node may be overloaded with too frequent update tasks in a volatile environment with frequent fluctuations of stream statistics. In a situation like this, the distributed NOS update strategy, in which each node communicates with only the neighboring nodes and updates its own NOS locally, is preferable.

4.2.2 Distributed node-operator-set update

The distributed NOS update is triggered locally at each node when its stream statistics change more than a threshold. Then, the plan modification is to reorder the streams in O_i and adjust the join placements in P_i so that the invariant cost-condition is satisfied again for every SEP. This is done at each node through communication with the neighboring nodes.

Let us consider a stream S_{i_k} at node N_{i_k}. Remember that a SEP_i is constructed based on the greedy strategy that a stream S_{i_k} is joined with S_p if and only if S_p has the lowest cost among all the streams that it can join to. A natural outcome of this strategy is that a stream with a lower join cost tends to be placed closer to the beginning of the join sequence and a stream with a higher cost tends to be placed farther from it. Thus, if the change of stream statistics makes $C^i_{i_{k-1},i_k}$ increase, then the position of S_{i_k} in O_i may need to shift toward the end, and if decrease then toward the beginning. Once the position of S_{i_k} changes in O_i, other streams in O_i may need to be reordered as well to make sure the invariant cost-condition is satisfied by all streams. The reordering mechanism is similar to what happens during the insertion sorting.

Algorithm 3 outlines the steps of modifying each SEP through distributed updates of NOSs. The algorithm is executed in two stages. The first stage is to move the stream that triggered the algorithm execution to the right position in O_i. The second stage is to continue to update the other streams affected by the move. The update of NOSs involves three steps. First, it determines the stream S_{i_r} to be swapped with S_{i_k} in the join sequence O_i. Second, it updates $NOS^i_{i_k}$ and $NOS^i_{i_r}$ and also the NOSs of the nodes of $S_{i_{k-1}}$ and $S_{i_{k+1}}$ in the join sequence. Third, it computes the join placements and, if necessary, updates NOS^i_k and NOS_{i_r} to reflect new join placements. Details of each step are presented now.

```
1  begin
2      for each $SEP_i$ do
3          if $C^i_{i_{k-1},i_k}$ has increased then
4              j = k;
5              begin // $S_{i_j}$ moves toward the end of $O_i$
6                  Determine $S_{i_r}$ among $\{S_{i_{j+1}}, S_{i_{j+2}}, .., S_{i_{m-1}}\}$ to be swapped with $S_{i_j}$ in $O_i$;
7                  Update $NOS_{i_r}$ and $NOS_{i_j}$ to reflect the swap and send messages to the nodes of the
                   preceding and succeeding streams of $S_{i_r}$ and $S_{i_j}$, respectively, to update their NOSs;
8                  Update $NOS_{i_r}$ to reflect a new join placement;
9              end
10             Repeat the steps 6 to 8 for $j = k + 1$ to $m - 1$;
11         end
12         else
13             begin // $S_{i_k}$ moves toward the beginning of $O_i$
14                 Determine $S_{i_r}$ among $\{S_{i_1}, S_{i_2}, .., S_{i_{k-1}}\}$ to be swapped with $S_{i_k}$ in $O_i$;
15                 Update $NOS_{i_r}$ and $NOS_{i_k}$ to reflect the swap and send messages to the nodes of the
                   preceding and succeeding streams of $S_{i_r}$ and $S_{i_k}$, respectively, to update their NOSs;
16                 Update $NOS_{i_k}$ to reflect a new join placement;
17             end
18             Repeat the steps 6 to 8 for $j = r + 1$ to $m - 1$ ; // Note: not the steps 14 to 16
19         end
20     end
21 end
```

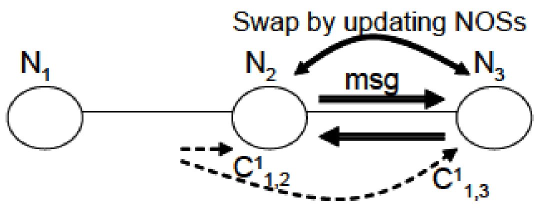
Algorithm 3: Distributed NOS update.

Stage 1: Update on the triggering stream

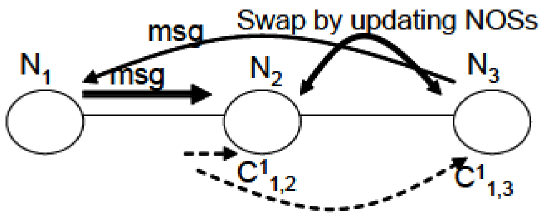
Step 1: Determine the stream to be swapped with S_{i_k} in O_i

If $C^i_{i_{k-1},i_k}$ has increased, one of the succeeding streams $S_{i_{k+1}},\ldots,S_{i_{m-1}}$ replaces S_{i_k} in O_i. Let S_{i_r} be the one that replaces S_{i_k}. Then S_{i_r} must be the one for which $C^i_{i_{k-1},i_r} \leq C^i_{i_{k-1},i_{k+}}$ holds for all i_{k+} in $\{i_k,i_{k+1},\ldots,i_{m-1}\}$. In order to find such an S_{i_r}, streams in O_i are processed sequentially as follows. The initial lowest cost is the new cost of one-way join to S_{i_k}, $C^i_{i_{k-1},i_k}$. N_{i_k} first sends a message to $N_{i_{k+1}}$ with the instruction that $N_{i_{k+1}}$ computes the cost $C^i_{i_{k-1},i_{k+1}}$ and compares it with $C^i_{i_{k-1},i_k}$. If $C^i_{i_{k-1},i_{k+1}} < C^i_{i_{k-1},i_k}$, then the cost of the join to $S_{i_{k+1}}$ is the lowest so far and thus, the lowest-cost stream is updated to $S_{i_{k+1}}$ (at the node $N_{i_{k+1}}$) and the lowest cost is updated to $C^i_{i_{k-1},i_{k+1}}$. Then, $N_{i_{k+1}}$ sends a message to its succeeding stream in O_i with the same instruction. This relay propagates until the message reaches $N_{i_{m-1}}$ while the lowest cost and the lowest-cost stream (and its node) are updated as necessary along the way. Then, $N_{i_{m-1}}$ informs N_{i_r} that S_r should be swapped with S_k.

Figure 3a illustrates the process of determining the new position of S_2 when $C^1_{1,2}$ increases. N_2 first sends a message to N_3 to compute $C^1_{1,3}$ and compare it with $C^1_{1,2}$. If $C^1_{1,3} < C^1_{1,2}$, then N_3 is recorded as the node with the lowest cost. Then N_3 will communicate with N_2 to swap the position.



(a) When the one-way join cost to S_3 has increased.



(b) When the one-way join cost to S_3 has decreased.

Figure 3. Illustration of determining the stream to switch with S_3 in the join sequence O_1.

On the other hand, if $C^i{}_{i_{k-1},i_k}$ has decreased, one of the preceding streams $S_{i_1},\ldots,S_{i_{k-1}}$ replaces S_{i_k} . Let S_{i_r} be the one that swaps with S_{i_k}. Then, $C^i{}_{i_{r-1},i_k} < C^i{}_{i_{r-1},i_r}$ must hold. Finding S_{i_r} needs more than considering the immediate preceding stream $S_{i_{k-1}}$. The reason is that, even if $C^i{}_{i_{k-2},i_k} < C^i{}_{i_{k-2},i_{k-1}}$, it is possible that $C^i{}_{i_{(k-)-1},i_k} < C^i{}_{i_{(k-)-1},i_{k-}}$ holds for some preceding stream $S_{i_{k-}}$ (i_{k-} in $\{i_1,i_2,\ldots,i_{k-1}\}$). Therefore, in order to find S_{i_r}, we need to compare $C^i{}_{i_{j-1},i_j}$ with $C^i{}_{i_{j-1},i_k}$ for all S_{i_j} starting from S_{i_1} until the first S_{i_j} satisfying $C^i{}_{i_{j-1},i_k} < C^i{}_{i_{j-1},i_j}$ is found. This S_j is the wanted S_r. Figure 3b illustrates the process of determining the new position of S_3 when $C^1{}_{1,3}$ decreases.

In order to compute the execution costs using Equation 3 and compare them, each node N_{i_k} maintains metadata which includes the stream rate $r^i{}_{i_{k-1}}$ and the tuple size $t^i{}_{i_{k-1}}$ of $S^i{}_{i_{k-1}}$ and the cost $C^i{}_{i_{k-1},i_k}$. In the case of increased $C^i{}_{i_{k-1},i_k}$, $N_{i_{k+}}$ needs to know $r^i{}_{i_{k-1}}$ and $t^i{}_{i_{k-1}}$ to compute $C^i{}_{i_{k-1},i_{k+}}$, and thus the message sent from N_{i_k} includes the two values. In addition, the message includes the information about the lowest cost and the lowest-cost stream (and its node) so that $N_{i_{k+}}$ can compare the computed $C^i{}_{i_{k-1},i_{k+}}$ with the lowest cost. In the case of decreased $C^i{}_{i_{k-1},i_k}$, $N_{i_{k-}}$ needs to compare its cost $C^i{}_{i_{(k-)-1},i_{k-}}$ with $C^i{}_{i_{k-1},i_k}$, and thus the message sent to $N_{i_{k-}}$ includes r_{i_k}, t_{i_k} and W_{i_k}.

Step 2: Update NOSs to reflect the swap

Since each node maintains its NOS for join execution, in order to reflect the swap between S_{i_k} and S_{i_r} in O_i, the $NOS^i{}_k$ at N_{i_k} and the NOS_{i_r} of N_{i_r} are updated. The update is done by swapping $NOS^i{}_k$ of N_{i_k} and NOS_{i_r} of N_{i_r} and then replacing the streamID of the operators. Specifically, when N_{i_r} receives $NOS^i{}_k$ from N_{i_k}, it replaces its NOS_{i_r} by $NOS^i{}_k$ and changes the streamID of S_{i_k} to the streamID of S_{i_r} in each operator. Similarly, when N_{i_k} receives NOS_{i_r} from N_{i_r}, N_{i_k} replaces its $NOS^i{}_k$ by NOS_{i_r} and changes the streamID of S_{i_r} to the streamID of S_{i_k} in each operator. When N_{i_k} and N_{i_r} finish updating their NOSs, they send messages to the nodes of the preceding stream and succeeding stream to instruct them to update their NOS by changing the streamID in the operators.

Note that the effect of swapping the positions of S_{i_k} and S_{i_r} is to swap the roles of these two streams in O_i and thus to swap NOS^i_k and NOS_{i_r}. The replacement of the streamID of operators is necessary because NOS_{i_r} is at N_{i_k} after the swap of NOSs but there is S_{i_r} at N_{i_k} and, thus, the streamID of S_{i_r} should be changed to the streamID of S_{i_k}. The streamID of S_{i_k} should be replaced by the streamID of S_{i_r} at N_{i_r} for the same reason. The nodes of the preceding stream and succeeding stream need to change the streamIDs because the succeeding stream and the preceding stream (i.e., either S_{i_k} or S_{i_r}) respectively are swapped with the other stream (i.e., either S_{i_r} or S_{i_k}).

Step 3: Update NOSs to reflect a new join placement

As we see from Equation 3, the join placement of a one-way join is determined by the smaller execution cost between the placements at the source and at the destination. This join placement may change when the stream statistics change. Thus, the nodes need to compute the execution costs to determine the new join placement and then update the NOSs if it is different from the old one. Note that the join placement computed during the step 1 (while the stream to be swapped with is found) is not shipped all the way through the nodes in the sequence, as it can be simply recomputed at the swapped nodes.

The join placement may change from the source node to the destination node or the other way around. To accommodate these changes, the join operator and the update operator in the NOS of each affected node should be moved to the other node. Thus, the node from which the join operator is moved out adds a ship operator to its NOS for shipping the input stream of the join operator to the other node.

Example 3 below illustrates updating NOSs to reflect the update of join ordering (O_i) and join placement (P_i).

Example 3: Continuing from Examples 1 and 2, consider $SEP_1 = (O_1, P_1) = ([S_2, S_3], [1,1])$. In this SEP_1, $C^1_{1,2} < C^1_{1,3}$ holds. Now suppose $C^1_{1,2}$ has increased and as a result the invariant cost-condition is violated. Further suppose that the ensuing plan modification tells that S_2 needs to be swapped with S_3 in O_1 and the new join from S_1 to S_3 needs to be executed in the source node. In other words, the new SEP_1 is $(O_1, P_1) = ([S_3, S_2], [0,1])$. In order to implement this change, the NOS in each node is updated as follows.

Before modification:

$NOS^1_1 = \{ship(S_1,N_2)\}$

$NOS^1_2 = \{join(S_1, S_2), update(S_2), ship(S^1_2, N_3)\}$

$NOS^1_3 = \{join(S^1_2, S_3), update(S_3), ship(S^1_3,N_0)\}$

After modification:

$NOS^1_1 = \{join(S_1, S_3), update(S_3), ship(S^1_3, N_2)\}$

$NOS^1_2 = \{join(S^1_3, S_2), update(S_2), ship(S^1_2, N_0)\}$

$NOS^1_3 = \{ship(S_3, N_1)\}$

Stage 2: Update on the other affected streams

After the swap of S_{i_k} and S_{i_r}, depending on whether $C^i_{i_{k-1},i_k}$ has increased or decreased, the positions of the other streams in O_i may need to change as well. Specifically, if $C^i_{i_{k-1},i_k}$ has increased, it may happen that the subsequence $\{S_{i_1},\ldots,S_{i_{k-1}},S_{i_r}\}$ satisfies the invariant cost-condition but the rest $\{S_{i_{k+1}},\ldots, S_{i_{r-1}},S_{i_k},S_{i_{r+1}},\ldots,S_{i_{m-1}}\}$ does not. Thus, the update needs to continue from $S_{i_{k+1}}$. In the same manner, if $C^i_{i_{k-1},i_k}$ has decreased, it may happen that the subsequence $\{S_{i_1},\ldots,S_{i_{r-1}},S_{i_k}\}$ satisfies the invariant cost-condition but the rest of the sequence $\{S_{i_{r+1}},\ldots, S_{i_{k-1}},S_{i_r},S_{i_{k+1}},\ldots,S_{i_{m-1}}\}$ does not because the substitution of S_{i_k} for S_{i_r} affects the join cost to its succeeding streams. Thus, the update needs to continue from $S_{i_{r+1}}$. Each update is performed following the stage 1 steps used in the case of increased cost (i.e., the steps 6 to 8 in Algorithm 3), as any necessary swaps are with another stream succeeding in the sequence.

The complexity of the algorithm in terms of the number of messages sent is $O(n^3)$ where n is the number of nodes, as there are n SEPs (one for each stream) and in the worst case there are n^2 NOS-swaps performed for each sequence.

4.3 Distributed plan migration

As mentioned in Introduction, the main challenge in the distributed plan migration is that windows are distributed over the nodes and thus, when the plan changes, the windows need to move to new nodes without disrupting the join execution. There should be no tuple missing or duplicated in the output. The proposed distributed track strategy is using the parallel track idea proposed by Zhu et al. (Zhu, Rundensteiner, and Heineman 2004) as the basis and exceeds its limit to address the distribution challenge.

In this strategy, window contents are moved to new nodes as soon as possible and each node runs the new NOS and the old NOS in parallel until the new NOS is ready, and at that point the old NOS is dropped and only the new NOS is used for join execution. In order to guarantee no missing tuple in the output, the new NOS is not ready until the old plan and the new plan generate the same output for every SEP. Note that the same output is generated only if for every window its content in the new node is the same as its content in the old node. In order to guarantee no duplicate tuples, join operators in a new plan are not executed until the window extent in the new node is filled up after the parallel runs of the new and old plans.

There are two technical issues to address in the distributed track strategy. One issue is the delay until the window content of the new plan is filled up, until when the plan migration cannot be completed. The delay is as long as the window size, which is inapplicable for large window sizes. Our approach is to jump start the migration by copying the window extent from the old nodes to the new nodes as soon as the new nodes are identified during the plan modification. For this, for each stream the node that is maintaining its window should be kept track of, so that a new node can send a request to the old node and ask for the window content. The details are discussed below. The other issue is to make sure all the affected nodes start their own migration synchronously. This synchronized migration is important because otherwise one node may execute a new plan (after migration) while another node executes an old plan (before migration), thereby corrupting the distributed join result. This kind of synchronization has been well studied in the distributed computing area (e.g., two-phase commit), and we adopt a simple protocol approach described below.

Keeping track of the nodes maintaining windows

The distributed plan migration needs to keep track of which node currently maintains a window so that a request can be sent to the right node to ask for the window content. This issue is handled differently depending on the NOS update approach used in the distributed plan modification (see Section 4.2). In the centralized approach, all NOS updates are determined at the central node and thus the information on the window maintenance node is kept at a central node as well. In the distributed approach, the old node that currently has a window needs to communicate directly with the new node that will need the window. More specific steps are discussed next.

When the centralized NOS update approach is used, the central node N_q first compares the old NOS and the new NOS for each node to determine if a new window (e.g,. W_i) is needed by the new NOS. A new window is needed by an NOS if it contains an update operator (e.g., *update*(S_i)). Then, the central node looks for the update operator in the old NOS. If found, it sends to the node of the old NOS a request to make the window (e.g., W_i) accessible to its new NOS. If not found, then it means the node of the new NOS needs to obtain the window content from another node. To identify the node that currently maintains the window, the central node checks the old NOSs of all nodes to find the NOS that contains the update operator (e.g., *update*(S_i)). Suppose NOS_p is the one. The central node then sends to the node of NOS_p (i.e., N_p) a request to move the window (e.g., W_i) to the node of the new NOS. Additionally, the central node makes the list of all affected nodes by comparing the new NOS and old NOS in each node. This list is needed in the synchronization step to be discussed below.

When the distributed NOS update approach is used, the information on the window maintenance node is recorded while NOSs are updated to reflect the NOS swap and the new join placement (see Step 2 and Step 3 in Section 4.2.2). More specifically, during the swapping of the new NOS in a node N_{i_k} and the old NOS in a node N_{i_r}, if the new NOS contains an update operator (e.g., *update*(S_i)) that is in the old

NOS, then Ni_k sends a request to Ni_r to move the window (e.g., W_i) to Ni_k. Likewise, during the update of an NOS to reflect a new join placement, if an update operator (e.g., *update*(S_i)) is moved from one node (e.g., Ni_p) to another node (e.g., Ni_k), then Ni_k sends to Ni_p a request to move the window (e.g., W_i) to Ni_k. Additionally, each node (e.g., Ni_k) records the list of affected nodes which consists of the swapped node (i.e., Ni_r) and the nodes of the preceding stream and the succeeding stream of Ni_k (i.e., Ni_{k-1} and Ni_{k+1}). This list is used for the synchronization step, discussed next.

Synchronizing the migration among the nodes

The NOS switch-over performed by all affected nodes should be synchronized. For this we use a *two-phase migration* protocol. Each node participating in the migration maintains two states, *local-ready* and *global-ready*, and follows the protocol outlined below.

1. When the window specified in the NOS at the node becomes full, the node enters the *local-ready* state. In this state, it sends a "local-ready" message to all the other affected nodes and waits to receive "local-ready" messages from them.
2. When the number of received "local-ready" messages becomes equal to the number of affected nodes, the first phase is complete, and the node enters the *global-ready* state. In this state, it sends a "global-ready" message to all the other affected nodes and waits to receive "global-ready" messages from them.
3. When the number of received "global-ready" messages becomes equal to the number of the other affected nodes, the second phase is complete, and the node starts performing the migration.

The first phase of this protocol guarantees that all affected nodes are ready to perform the migration and the second phase guarantees that all affected nodes start performing the migration altogether. There may be technical issues pertaining to distributed computing, such as dealing with lost messages or network partitioning, but these issues are beyond the scope of this paper.

This protocol assumes that every affected node knows which the other affected nodes in the plan modification are and the messages are exchanged point to point among the nodes. An alternative would be to elect a coordinator, in which case the total number of messages can be reduced in return for the overhead of electing a new coordinator every time a new plan migration is to be done.

5 Performance Evaluation

There are two objectives in the performance evaluation. One is to see how effective the proposed adaptive processing mechanism is compared with non-adaptive processing. The other is to examine the additional overhead paid by the distributed NOS update approach compared with the centralized approach. In this section we describe the design, setup, and results of these two sets of experiments.

5.1 Experiment design

A prototype distributed data stream processing system has been built for the experiments. It runs on each node and comprises the three modules of optimizer, executor, and communicator. The optimizer generates the initial JEP, modifies the JEP, and extracts NOSs from a JEP. The executor performs the operations in the NOS, monitors the stream statistics, and notifies the optimizer if the change exceeds the threshold. The communicator delivers messages between the optimizers and the executors running on the distributed nodes. The prototype software program is written in Java 2 SDK 1.6.2 and uses TCP/IP as the communication protocol.

We have conducted the experiments in a network environment simulated using VMWare (VMWARE 1998). VMWare allows us to create multiple virtual machines that are connected through a virtual **fully connected** 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 fixed delay in every packet sent out to the virtual network. All virtual machines are created on a 2.0GHz Pentium Core 2 Duo computer with 2GB RAM; each virtual machine is configured to use the same CPU of the host computer and equally divided 192MB RAM of the host computer 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.

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 tuple size (given as the number of 5-byte attributes in the stream schema), and the name, size, and join 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.

The performance metric is the total execution cost 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 NOS and executes it. The execution cost, which includes the processing cost and the communication cost, is recorded at each node and sent to the query node for calculating the total execution cost.

5.2 Experiment results

Experiment 1: Effectiveness of the adaptive processing

The approach in this set of experiments is to compare the total execution costs between the adaptive case and the non-adaptive case while changing the stream statistics. For the purpose of this experiment, it is adequate enough to switch between low and high for the rate of one stream. Thus, we have changed the stream rate at one arbitrary node between 50 tuples/sec and 200 tuples/sec. A four-way join query and an eight-way join query have been used, with a fully-connected join topology.[1] For this, we set up 4 nodes and 8 nodes for the four-way join query and the eight-way join query, respectively.

Figure 4 shows how the execution time of the non-adaptive and the adaptive processing changes as the stream rate changes at the interval of 2000 msec starting from the time 1500 mec. Both four- and eight-way join cases show the same performance curves, except for magnitude of the costs. (The eight-way join involves more nodes in the processing and, thus, the total execution cost is larger.) The costs fluctuate more in the eight-way join case.

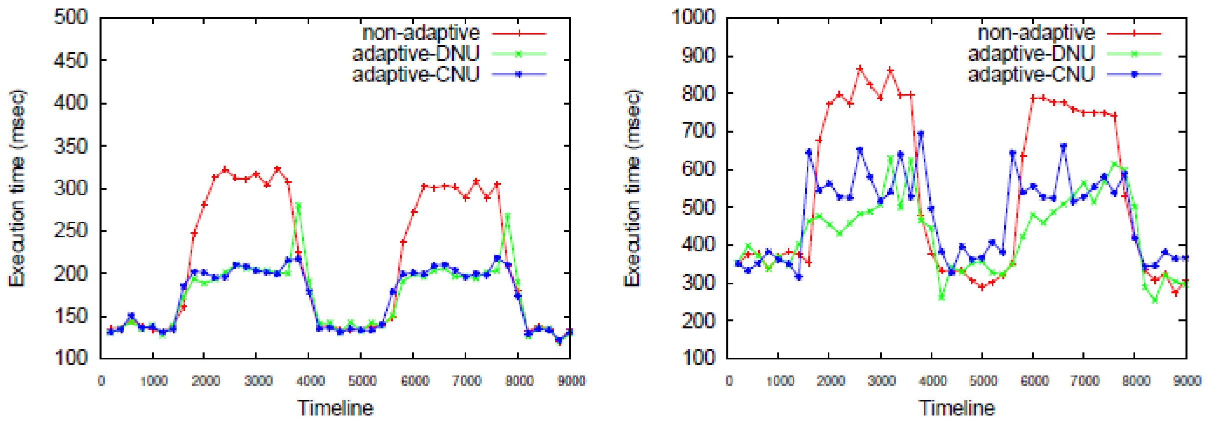
In the figure we see that initially both non-adaptive and adaptive processing use the same execution plan, and thus they show the same execution cost. However, when the stream rate changes from low to high (at time 1500 msec), the non-adaptive processing cost increases by a much larger margin than the adaptive processing cost. Evidently, the reason is that the non-adaptive processing is still using the initial plan, while the adaptive processing has changed it to a more efficient plan. By the same token, when the stream rate changes from high to low (at time 3500 msec), the adaptive processing switches back to the initial plan, which is the same plan the non-adaptive processing is using, and thus they have the same execution cost again.

We also see that there is hardly any difference between the two NOS update approaches (i.e., adaptive-CNU and adaptive-DNU) in the adaptive processing. It confirms that they always produce identical execution plans, since both NOS update approaches are based on the same greedy property.

The spike of the two costs when the stream rate changes from high to low in Figure 4(a) is interesting. This phenomenon can be explained as follows. During the plan migration in this case, a window content on the higher rate stream is copied from one node to another node and then the old NOS is switched to the new NOS for execution; at the beginning of the transition, however, the new NOS (generated for the lower-rate stream) is processing the higher-rate stream tuples still remaining in the window extent; this mismatch causes the cost to increase until the window content is replaced with the right (i.e., lower-rate stream) tuples. Another observation is that the spike in the adaptive-DNU is higher than in adaptive-CNU. The reason is that the former may need several rounds of NOS updates to reach a new efficient plan (see Stage 2 in Section 5.2) and, thus, may be using a less efficient plan than the latter during the stage 2. The spike is less visible in the eight-way join case (Figure 4(b)). This comes from the fact that we vary the rate of only one stream in our experiment setup and, therefore, the transient overhead is due to the migration of one window; as a result, the impact of the transient overhead caused by one out

[1] This join topology is common in stream join queries. Examples are joins on IP addresses for network monitoring applications and joins on sensor IDs for sensor monitoring applications.

of eight windows (in the eight-way join case) is lower than that caused by one out of four windows (in the four-way join case).



(a) A four-way join. (b) An eight-way join.

Adapt-CNU refers to the adaptive processing using the *centralized* NOS update approach, and adapt-DNU refers to the adaptive processing using the *distributed* NOS update approach. The execution time is the average obtained from ten repeated runs. The default settings are as follows: tuples size = 150 bytes; stream rate = 100 tuples/sec; network bandwidth = 1024 Kbps; latency = 5 msec; selectivity factor = 0.01; window size = 500 msec; measurement interval = 200 msec.

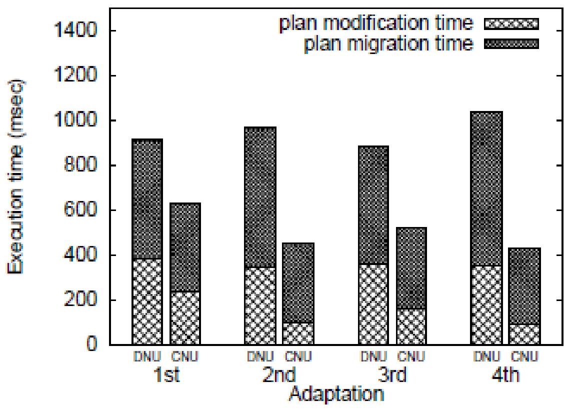
Figure 4. Comparison between adaptive processing and non-adaptive processing.

Experiment 2: Overhead of the adaptive processing

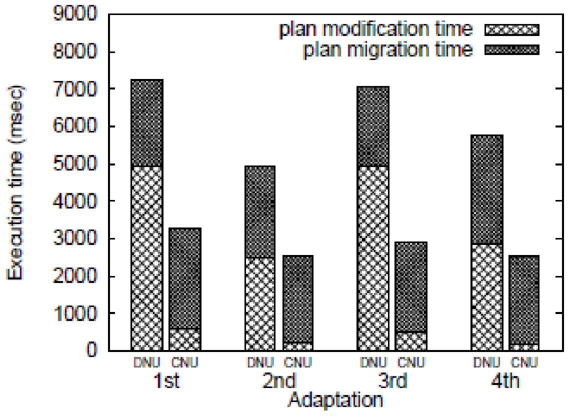
Adaptivity requires plan modification and plan migration every time stream statistic changes at any node participating in the join. The objective of this set of experiments is to examine the overhead in terms of the costs of plan modification and migration. The plan modification cost is measured as the total time all nodes spend to obtain new NOSs, and the plan migration cost is measured as the total time all nodes spend to switch over from old NOSs to new NOSs.

Figure 5 shows the experimental results. Each bar in the figure shows the sum of the plan modification cost and the plan migration cost, with the two distinguished using different patterns. The setup of the experiments is the same as the setup of Experiment 1, and thus the 1st and the 3rd pairs of bars in the figure look similar and so do the 2nd and the 4th. This is evident from the fact that those pairs are for the same changes of stream rates (see the changes at times 1500 msec and 5500 msec and at times 3500 msec and 7500 msec in Figure 4).

The figure shows that the total overhead (considering both plan modification and migration) of the adaptation using the distributed NOS update (DNU) approach is larger than the overhead in the centralized node update (CNU) approach by a factor of approximately 1.4 to 2.5 in the four-way join case and 2.1 to 2.7 in the eightway join case. There is no objective baseline to use to judge how acceptable these additional overheads are. Nonetheless, considering the facts that during query plan modification the NOS update at each node may occur multiple times in DNU but only once in CNU and that with the fully-connected topology there are 6 and 28 alternative join execution plans in the networks of four and eight nodes, respectively, we assert that the additional overhead to be paid for the distributivity is marginal.[2]



(a) A four-way join.



(b) An eight-way join.

The labels of the bars, DNU and CNU, refer to the adaptive processing using, respectively, the distributed and the centralized node update approaches. The default settings and the number of repeated runs are the same as those of Experiment 1 (Figure 4); the first and the third rounds correspond to the times at which the stream rate increases, and the second and the third rounds correspond to the times at which the stream rate decreases.

Figure 5. Plan modification and migration overheads.

We make a number of additional observations in the figure. First, the plan modification cost is significantly higher for the eight-way join than for the four-way join. This is obvious from the fact that there are more nodes involved in the modification process. Second, the plan modification cost is higher when the stream rate increases (the 1st and 3rd bars) than when it decreases (the 2nd and 4th bars). (It shows more clearly in the eight-way join case (Figure 5b).) This difference indicates that, given the parameter settings, there are more nodes whose NOSs are updated when the stream rate increases than when it decreases. Third, the plan modification cost is higher with DNU than CNU. This expected from the fact that the computation cost is $O(n^3)$ for DNU and $O(n)$ for CNU (where n is the number of nodes). Fourth, the plan migration cost is higher with DNU than CNU in the four-way join case (Figure 5a) but lower in the eight-way join case. This difference indicates that the migration overhead in DNU due to multiple rounds of NOS updates is higher than the one-time NOS update overhead of CNU in the four-way join case but lower in the eight-way join case. (The migration cost depends on the number of

[2] We may as well say that the overhead of the DNU approach over the CNU approach is a factor of 0.25 to 0.6 times per node in the four-way join case and 0.26 to 3.34 times per node in the eight-way join case.

shipments of window contents, and this number varies depending on the number of rounds of NOS updates.)

		When the stream rate increases			When the stream rate decreases		
		Number of modification messages	Number of migration messages	Number of synchronization messages	Number of modification messages	Number of migration messages	Number of synchronization messages
4 way join	CNU	4	34	24	4	27	24
	DNU	65	61	44	59	55	32
8 way join	CNU	8	131	112	8	115	112
	DNU	561	181	152	283	83	48

(The number of migration messages includes the number of synchronization messages. The number of messages reported in this table is measured each time the plan modification and plan migration algorithms are triggered. The measured numbers do not change through different repetitions, since the same steps in the algorithms are executed for each repetition.)

Table 3. Number of messages for plan modification and migration.

Table 3 summarizes the number of massages sent through the network for plan modification and plan migration, respectively. We now discuss a number of observations from this table.

In the plan modification part, first, the number of messages in CNU is the same as the number of nodes, *n*. This is because the messages consist of the message sent by a node to the central node to notify the change of its stream rate and *n-1* messages sent by the central node to all the other nodes (which are all affected nodes with the experiment setting) to update their NOSs. Second, the messages in DNU consist of the messages sent through the nodes in a join sequence to determine the nodes to be swapped, the messages sent to update the NOSs of the other nodes to realize node swapping, and the messages sent to update the NOSs to realize join placement changes. The complexity of the DNU approach in term of the number of messages sent is $O(n^3)$.

In the plan migration part, the table shows the total number of migration messages and the number of synchronization messages. (Note the former includes the latter.) In CNU, the migration messages consist of the messages for requesting window movements and the messages for synchronization. With the two-phase synchronization in place, the number of synchronization messages is $2n(n-1)$ since each node sends messages to all the other nodes in each of the two phases. In DNU, the migration messages consist of the messages for requesting the window movement each time NOSs swap or join placement changes and the messages for synchronization. The number of synchronization messages varies depending on the number of synchronized nodes and the number of synchronizations performed.

6 Related Work

Major research on adaptive query processing has been for relational databases (Markl, Raman, Simmen, Lohman, and Pirahesh 2004; Kabra and DeWitt 1998; Cole and Graefe 1994; Ioannidis, Ng, Shim, and Sellis 1997; Antoshenkov and Ziauddin 1996; Avnur and Hellerstein 2000; Deshpande and Hellerstein 2004; Babu, Bizarro, and DeWitt 2005), and recently for data streams (Zhu, Rundensteiner, and Heineman 2004; Tian and DeWitt 2003; Babu, Motwani, Munagala, Nishizawa, and Widom 2004; Babu, Munagala, Widom, and Motwani 2005). In this section we give an overview of the related research in these two areas.

Adaptive database query processing

We find two approaches based on *re-optimization*, which is the base of our proposed approach. Kabra et al. (Kabra and DeWitt 1998) introduce a *dynamic* re-optimization approach which detects the sub-optimality of a plan while executing the query and re-optimizes the plan in the midst of query execution if there is a significant difference between estimated and actual values. Similar to Kabra el al.'s approach, Markl et al. (Markl, Raman, Simmen, Lohman, and Pirahesh 2004) propose a technique called *progressive* query optimization which adds one or more checkpoint operators to a plan to compare the optimizer's estimates with the actual values and triggers reoptimization if a pre-determined threshold on the estimation error is exceeded. There have been other approaches to adaptive query processing as well, such as the *competition* model (Antoshenkov and Ziauddin 1996), *parametric* optimization (Cole and Graefe 1994; Ioannidis, Ng, Shim, and Sellis 1997), *tuple-routing* (Avnur and Hellerstein 2000; Deshpande and Hellerstein 2004), and *proactive* optimization (Babu, Bizarro, and DeWitt 2005). In the *competition* model approach, multiple plans are executed together until one of the plans becomes better than the others and then the executions of the sub-optimal plans are stopped. The *parametric* optimization approach is to prepare separate plans optimal for different partitions of the parameter domain and choose a plan when the actual parameter values are known at run-time. The *tuple-routing* approach handles the query processing by routing tuples through a pool of operators. This mechanism is handled by an operator called *eddy* which continuously reorders operators in a query as it is running. In the *proactive* optimization approach, query plans are selected with possible re-optimization in mind, that is, to minimize it. The authors introduce bounding boxes to determine the uncertainty in the estimates of statistics and use these bounding boxes during optimization to generate robust and switchable plans that minimize the need for re-optimization as well as the cost of switching plans.

There has been some work done in distributed databases as well. Scheuermann and Chong (Scheuermann and Chong 1997) present an adaptive algorithm for finding a query execution plan. This work, however, is not about adaptive query processing and does not address the problem of modifying a plan when the statistics or computing resources change. Zhou (Zhou 2003) proposes an adaptive distributed query processing framework. It, however, only focuses on the scheme for learning the selectivity and workload of distributed servers. Besides, the adaptive mechanism is based on the eddy approach to reorder the operations at runtime.

All these adaptive approaches are for databases and are not geared for handling data stream query processing.

Adaptive data stream query processing

Adaptive query processing of data streams pose unique challenges due to the requirement of continuous and unbounded processing of arriving data. There have been some research done to address these challenges using the re-optimization approach (Babu, Motwani, Munagala, Nishizawa, and Widom 2004; Babu, Munagala, Widom, and Motwani 2005; Zhu, Rundensteiner, and Heineman 2004; Yang, Kr¨amer, Papadias, and Seeger 2007). Babu et al. (Babu, Motwani, Munagala, Nishizawa, and Widom 2004; Babu, Munagala, Widom, and Motwani 2005) study the problem of quickly detecting the change in stream statistics and efficiently switching to an equivalent yet more efficient plan. Specifically, the focus of (Babu, Motwani, Munagala, Nishizawa, and Widom 2004) is on adaptively ordering pipelined filters, where they introduce the *A-Greedy* algorithm which uses a greedy strategy to reorder the filters when the greedy property is violated. Then, the focus of (Babu, Munagala,Widom, and Motwani 2005) is on handling the migration between join execution plans of a multi-way join query, where they introduce the *A-Caching* algorithm which stores the intermediate results of join subsequences to support the plan migration. These algorithms assume a centralized environment, and they do not work in a distributed environment because it requires a different model and techniques.

Zhu et al. (Zhu, Rundensteiner, and Heineman 2004) focus on the problem of plan migration and propose two solutions for the plan migration, a *moving state* strategy and a *parallel track* strategy. The moving state strategy temporarily suspends the query execution in order to produce an intermediate result for the new plan and then switches over to the new plan. The parallel track strategy runs both the new plan and the old plan in parallel and drops the old plan when it is not needed anymore. Yang et al. (Yang, Kramer, Papadias, and Seeger 2007) proposes an improvement over Zhu et al.'s approach by combining the two strategies. Besides the fact that these approaches are for centralized processing environment, the proposed techniques assume the tree ordering when moving an intermediate result during plan migration, which is inapplicable in our problem as we use the linear join ordering.

Another line of research on adaptive stream query processing is based on the tuple-routing mentioned above. This approach is different from the re-optimization approach. Tian and DeWitt (Tian and DeWitt 2003) consider the adaptive processing of stateful operators in a distributed environment. They use the *tuple-routing* approach (i.e,, eddies (Avnur and Hellerstein 2000)) as the basis and extend it to work in a distributed environment with *multiple eddies*. Each eddy at a local node takes the input tuples and, based on pre-defined policies, determines the next operator (in another node) to forward the output tuples to. This technique requires metadata for each tuple to keep track of its processing progress, and it makes the system prohibitively expensive in terms of the communication overhead. Claypool et al. (Claypool and Claypool 2008) present an improvement over the eddies, called *trained eddies (TEddies)*. TEddies processes tuples in a batch instead of single tuples and has an adaptive scheduler module to adapt to the changes of the stream statistics and the number of tuples in a batch. TEddies is meant for a centralized environment.

Another work in distributed databases that has a similar distributed data model to our work is OGSA-DAI (OGSA-DAI 2002). In their model, multiple databases are placed in different nodes distributed over the network and can be queried through a server which makes the queried databases appear as one logical database. That is, given a query, the distributed query processor generates an initial query plan and then breaks it into operators that need to be executed at each node. Their work is fundamentally different from ours in two aspects. First, their query optimizer is rule-based, that is, the plan is optimized based on a set of predefined rules, whereas our query optimization algorithm is cost-based. Second, our work focuses on the adaptive query processing which modifies the execution plan when the stream statistics change, while their work does not.

7 Conclusion

In this paper, we have addressed the problem of adaptively processing multi-way windowed stream joins over distributed data streams. The key idea is to use a node operator set (NOS) to support the adaptivity locally while ensuring the correctness globally over the network. Based on the notion of NOS, we have presented two distributed plan modification techniques depending on whether the NOSs in all nodes are updated centrally or distributed. In the centralized update, one node maintains and updates the NOSs and sends updated NOSs to affected nodes. In the distributed update, individual nodes communicate with their neighboring nodes to update their own NOSs by themselves. Further, we have presented distributed plan migration techniques for centralized and distributed NOS updates. Both techniques guarantee a correct switch-over from an old plan to a new plan. Finally, we have conducted two sets of experiments to test the adaptivity of the developed techniques as a whole and to study the time overheads of the plan modification part and plan migration part in one view.

There are some directions we are considering for future work. First, the proposed adaptive framework assumes the *entire* window contents are migrated from one node to another as a result of plan modification. A more efficient mechanism that requires only a part of the window content would be more desirable. Second, the framework considers only the stream statistics as something to adapt to. The change of available system resources (e.g., memory, CPU time) is another factor commonly considered for adaptivity. If the system resources become short, then approximating the join results may be necessary. In this case, finding an adequate quality metric of the approximate result in a distributed environment and computing it efficiently will be an interesting problem. Third, the join processing model assumes the linear ordering in which intermediate join results are not buffered. Extending the model to maintain intermediate join results may improve the efficiency of join processing. The solution requires a technique to maintain and migrate intermediate results efficiently in our adaptive framework. Fourth, a variation of the distributed NOS update algorithm can be developed by limiting the number of streams to be searched in determining the stream to be swapped; in this way, the quality of the query plan (i.e., the execution time) might not be as good as that of our proposed algorithm, but the number of messages exchanged can be reduced.

References

- Aho, A. V., Y. Sagiv, and J. D. Ullman (1979). Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems* 4(4), 435-454.
- Amini, L., N. Jain, A. Sehgal, J. Silber, and O. Verscheure (2006). Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, Number 71 (CD).
- Antoshenkov, G. and M. Ziauddin (1996). Query processing and optimization in Oracle RDB. *The Very Large Databases Journal 5*(4), 229–237.
- Avnur, R. and J. M. Hellerstein (2000). Eddies: Continuously adaptive query processing. In *Proceedings of the 19th International Conference on Management of Data*, pp. 261–272.
- Babu, S., P. Bizarro, and D. J. DeWitt (2005). Proactive re-optimization. In *Proceedings of the 24th International Conference on Management of Data*, pp. 107–118.
- Babu, S., R. Motwani, K. Munagala, I. Nishizawa, and J.Widom (2004). Adaptive ordering of pipelined stream filters. In *Proceedings of the 23rd International Conference on Management of Data*, pp. 407–418.
- Babu, S., K. Munagala, J. Widom, and R. Motwani (2005). Adaptive caching for continuous queries. In *Proceedings of the 21st International Conference on Data Engineering*, pp. 118–129.
- Ceri, S. and G. Pelagatti (1984). *Distributed Databases: Principles and Systems*.
- Claypool, K. T. and M. Claypool (2008). Teddies: Trained eddies for reactive stream processing. In *Proceedings of the 11st International Conference on Database Systems for Advanced Applications*, pp. 220–234.
- Cole, R. L. and G. Graefe (1994). Optimization of dynamic query evaluation plans. In *Proceedings of the 13rd International Conference on Management of Data*, pp. 150–160.

- Cormode, G., S. Muthukrishnan, and W. Zhuang (2006). What's different: Distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In *Proceedings of the 22nd International Conference on Data Engineering*, pp. 57.
- Das, A., S. Ganguly, M. N. Garofalakis, and R. Rastogi (2004). Distributed set expression cardinality estimation. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pp. 312.
- Deshpande, A. and J. M. Hellerstein (2004). Lifting the burden of history from adaptive query processing. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pp. 948–959.
- Gedik, B., K.-L. Wu, P. S. Yu, and L. Liu (2007). 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.
- Golab, L. and M. T. Ozsu (2003). 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 Press.
- Ioannidis, Y. E., R. T. Ng, K. Shim, and T. K. Sellis (1997). Parametric query optimization. *The Very Large Databases Journal 6*(2), 132–151.
- Kabra, N. and D. J. DeWitt (1998). Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 17th International Conference on Management of Data*, pp. 106–117.
- Kang, J., J. F. Naughton, and S. D. Viglas (2003). Evaluating window joins over unbounded streams. In *Proceedings of the 19th International Conference on Data Engineering*, pp. 341–352. IEEE Computer Society.
- Kumar, V., B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan (2005). Resource-aware distributed stream management using dynamic overlays. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pp. 783.
- Kumar, V., B. F. Cooper, and K. Schwan (2005). Distributed stream management using utility-driven self adaptive middleware. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pp. 3.
- Markl, V., V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh (2004). Robust query processing through progressive optimization. In *Proceedings of the 23rd International Conference on Management of Data*, pp. 659–670.
- OGSA-DAI (2002). http://www.ogsadai.org.uk/
- Olston, C., J. Jiang, and J. Widom (2003). Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 22nd International Conference on Management of Data*, pp. 563.
- Scheuermann, P. and E. I. Chong (1997). Adaptive algorithms for join processing in distributed database systems. *Distributed and Parallel Databases 5*(3), 233–269.
- Seshadri, S., V. Kumar, and B. F. Cooper (2006). Optimizing multiple queries in distributed data stream systems. In *Proceedings of Workshop of the 22nd International Conference on Data Engineering*, pp. 25.
- Sharfman, I., A. Schuster, and D. Keren (2006). A geometric approach to monitoring threshold functions over distributed data streams. In *Proceedings of the 25th International Conference on Management of Data*, pp. 301.

- Srivastava, U., K. Munagala, and J.Widom (2005). Operator placement for in-network stream query processing. In *Proceedings of the 24th Symposium on Principles of Database Systems*, pp. 250–258.
- Tian, F. and D. J. DeWitt (2003). Tuple routing strategies for distributed eddies. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pp. 333–344.
- Tran, T. M. and B. S. Lee (2010). Distributed stream join query processing with semijoins. *Distributed Parallel Databases 27*(3), 211–254.
- Viglas, S., J. F. Naughton, and J. Burger (2003). 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.
- VMWARE (1998). VMWare Workstation 6.0: http://www.vmware.com/.
- Yang, Y., J. Kramer, D. Papadias, and B. Seeger (2007). Hybmig: A hybrid approach to dynamic plan migration for continuous queries. *IEEE Transactions on Knowledge and Data Engineering 19*(3), 398–411.
- Zhou, Y. (2003). Adaptive distributed query processing. In *Proceedings of PhD Workshop of the 29th International Conference on Very Large Databases*.
- Zhou, Y., Y. Yan, F. Yu, and A. Zhou (2006). 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.
- Zhu, Y., E. A. Rundensteiner, and G. T. Heineman (2004). Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 23rd International Conference on Management of Data*, pp. 431–442.