Adaptive Load Diffusion for Multiway Windowed Stream Joins

Xiaohui Gu, Philip S. Yu, Haixun Wang
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{xiaohui, psyu, haixun}@ us.ibm.com

Abstract

In this paper, we present an adaptive load diffusion operator to enable scalable processing of Multiway Windowed Stream Joins (MWSJs) using a cluster system. The load diffusion is achieved by a set of novel semantics-preserving tuple routing algorithms. Different from previous work, the load diffusion operator can (1) preserve the MWSJ semantics while spreading tuples to different hosts for parallel join processing; (2) achieve fine-grained load balancing among distributed hosts; and (3) perform semantics-preserving online adaptations to maintain optimal performance in dynamic stream environments. We have implemented a prototype of the distributed MWSJ framework on top of the System S distributed stream processing system. Our experiment results based on both real data streams and synthetic workloads show that the load diffusion algorithms can efficiently scale-up the performance of MWSJ processing with low overhead.

1 Introduction

Data stream management systems (DSMS) represent a vibrant and exciting research area [19, 13, 30, 3, 10]. One fundamental problem in DSMS is to process continuous queries (CQ) over high-volume and time-varying data streams under resource constraints. Our research focuses on multi-way windowed stream join (MWSJ), a core operator in CQ systems. The MWSJ operator can be used to discover correlations across different streaming sources, which has many important applications in video surveillance, network intrusion detection, and sensor monitoring. Given high stream rates and large sliding-windows, stream joins often have large memory requirement [23]. Moreover, some query processing such as video image similarity analysis can also be CPU-intensive [14]. Previous work has proposed load shedding techniques (e.g., [25, 6, 23, 9]) to handle the overload problem. In contrast, we propose a new load diffusion approach, which can adaptively distribute excessive MWSJ workload among multiple hosts. Different from coarse-grained load distribution solutions (e.g., [29]), our scheme achieves *adaptive*, *fine-grained* load balancing at tuple level, which allows a single MWSJ operator to utilize aggregated resources of multiple hosts.

Different from other CQ operators, MWSJ presents several new challenges to distributed stream processing systems. First, MWSJ requires the distribution scheme to observe a *correlation constraint*, that is, each tuple needs to join with the tuples contained in the sliding-windows of all the other streams. Thus, the load diffusion scheme needs to consider not only load balancing requirements but also the correlation constraint. Second, correlations among unstructured streaming information (i.e., video, audio and text) often need to evaluate complex join predicates. For example, in our news video correlation application, the join predicate is whether two video images are close to each other in a multi-dimensional concept space. Thus, the load diffusion scheme is desired to be generic, which should be able to support both equijoins and non-equijoins. Third, in dynamic stream environments where stream rates and host load conditions can vary over time, the load diffusion scheme must perform continuous on-line adaptations in order to maintain efficient query processing.

In this paper, we present a novel *adaptive load diffusion* operator to achieve scalable processing of MWSJ queries. The diffusion operator performs *semantics-preserving tuple routing*, which considers not only the load balancing goal but also the correlation constraint for preserving join accuracy. Dynamic tuple routing has shown to be effective for adaptive CQ processing [2, 26, 7, 21, 8]. Previous tuple routing schemes have been focusing on optimizing CQ processing time. Our work provides a complementary tuple routing framework, which can provide fine-grained load diffusion for resource-intensive MWSJ queries. Existing tuple routing schemes do not consider the routing constraints (e.g., correlation constraint) imposed by the MWSJ semantics, which however can affect the accuracy of join results by routing two tuples that need to be joined to different hosts. In contrast, our tuple routing algorithms consider

both query semantics and host load conditions to achieve accurate scalable CQ processing. As the first step toward a semantics-preserving tuple routing framework, this paper focuses on the correlation constraint of MWSJ queries. By explicitly observing MWSJ semantics, the load diffusion operator achieves desired generality, which can (1) support both equijoins and non-equijoins; and (2) allow each host to execute different stream join algorithms (e.g., load-shedding-enabled stream joins [23], m-joins [28], and hash joins). In contrast, previous value-based partition schemes (e.g., Flux [22]) can only support equijoins. We summarize the major contributions of this paper as follows:

- We propose the first semantics-preserving tuple routing framework to provide adaptive load diffusion for processing MWSJ queries using cluster systems. We have proved that a diffusion overhead (i.e., tuple replications and intermediate join result transferring) is unavoidable in any semantics-preserving tuple routing solutions for MWSJ queries. This analysis introduces an interesting new optimization problem, that is, how to optimally route tuples to different hosts for join processing with best load balancing and minimum diffusion overhead.

- We present two semantics-preserving tuple routing algorithms, *aligned tuple routing* (ATR) and *coordinated tuple routing* (CTR), to accommodate different types of MWSJ queries in terms of sliding-window sizes, join selectivity and others. ATR performs one-hop routing while CTR performs multi-hop routing by allowing transmission of intermediate join results. Both ATR and CTR are (1) *scalable*, which allow a single MWSJ operator to utilize any number of hosts with the same diffusion overhead; (2) *semantics-preserving*, which do not miss any join results or generate duplicate join results; and (3) *adaptive*, which can dynamically adjust their algorithm behaviors to maintain best performance in dynamic stream environments.

- We have implemented a prototype of the semantics-preserving tuple routing framework on top of our distributed stream processing infrastructure System S [18] that consists of about 250 blade servers. We conduct extensive experiments using both real data streams and a range of synthetic workloads. Our experiments show several interesting results: (1) the diffusion algorithms can achieve much higher join throughput than existing distribution schemes; (2) the diffusion operator is light-weight and fast, which can route a tuple within tens of micro-seconds on a server host; and (3) on-line adaptations can effectively improve join performance in dynamic stream environments. We also evaluate our algorithms using different MWSJ queries to illustrate the tradeoffs between ATR and CTR.

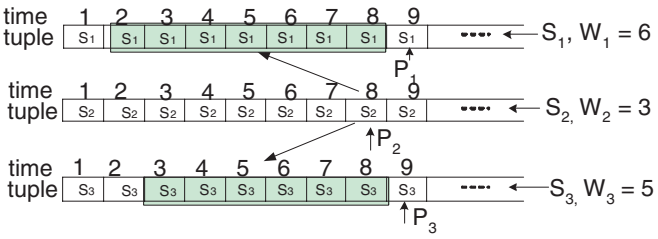

Figure 1. Multi-way windowed stream join.

The rest of the paper is organized as follows. Section 2 presents the system model, problem formulation and an overview of our approach. Section 3 and 4 present the ATR and CTR algorithms, respectively. Section 5 presents a thorough experimental evaluation. Section 6 briefly reviews related work. Finally, the paper concludes in Section 7.

2 Model, Problem and Our Approach

The MWSJ operator performs continuous join computations over sliding-windows of multiple streams, which is illustrated by Figure 1. A data stream S_i consists of a sequence of tuples $s_i \in S_i$. Let $s_i\langle t\rangle$ denote a tuple arrived on S_i at time t, and r_i denote the current mean arrival rate of the stream S_i. We assume that each tuple $s_i \in S_i$ carries a time-stamp $s_i.t$ in its header to denote the time when the tuple s_i arrives at the stream S_i. We use $S_i[W_i]$ to denote a sliding window on the stream S_i, where W_i denotes the length of the window in time units. At time t, we say s_i belongs to $S_i[W_i]$ if s_i arrives at S_i in the time interval $[t - W_i, t]$. We consider the multi-way stream join query among n input streams ($n \geq 2$), denoted by $J_i = S_1[W_1] \ldots \bowtie S_n[W_n]$. Figure 1 illustrates a three-way windowed stream join operator $S_1[6] \bowtie S_2[3] \bowtie S_3[5]$. The semantics of our sliding-window join is consistent with the semantics used in CQL [1]. Namely, the output of the MWSJ operator consists of all tuple groups $(s_1, s_2, ..., s_n)$, such that $\forall s_i \in S_i, \forall s_k \in S_k[W_k], 1 \leq k \leq n, k \neq i$ at time $s_i.t$, $s_1, ..., s_n$ satisfy a pre-defined join predicate $\theta(s_1, ..., s_n)$. We use $S_i[t, t + T]$ to denote a segment of S_i that includes all the tuples arrived at S_i within time $[t, t + T)$, where T is called the segment length[1]. In Figure 1, the tuple $s_2\langle 8\rangle$ needs to join with the tuples in $S_1[2, 9]$ and $S_3[3, 9]$.

The distributed MWSJ processing system consists of a diffusion operator D, a set of distributed join operators $J_{i,k}$, and a fusion operator F, illustrated by Figure 2. The diffusion operator dynamically routes input tuples to different

[1]This definition is to avoid the overlapping between two consecutive segments $S_i[t, t + T]$ and $S_i[t + T, t + 2T]$.

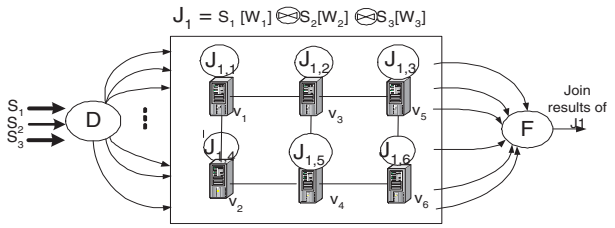

Figure 2. Distributed MWSJ execution.

server hosts for join processing, while the fusion operator aggregates dispersed join results into complete query answers. Different from the join operator, the diffusion operator performs fast tuple routing computations and requires little buffering of input streams. Thus, the diffusion operator is not the bottleneck in the system. The join operators $J_{i,k}$ at different hosts execute centralized sliding-window join algorithms such as load-shedding-enabled stream joins [23], m-joins [28] or hash-joins.

To preserve join accuracy, the diffusion operator need to consider not only the load balancing goal but also the correlation constraint: *given an MWSJ query* $J_i = S_1[W_1] \bowtie ... \bowtie S_n[W_n]$*, a group of tuples* $(s_1, s_2, ..., s_n)$ *that must be correlated according to the MWSJ semantics, are processed once and only once.* For example, in Figure 1, the tuple $s_2\langle 8\rangle$ needs to join with the tuples $s_1\langle 2\rangle$,..., $s_1\langle 8\rangle$, which can be routed to different hosts by the diffusion operator. Suppose the tuples $s_1\langle 4\rangle, s_1\langle 5\rangle, s_1\langle 6\rangle$ are routed to v_1 and $s_1\langle 5\rangle, s_1\langle 6\rangle, s_1\langle 7\rangle$ are routed to v_2. If we send $s_2\langle 8\rangle$ to either v_1 or v_2, we miss some join results. If we send $s_3\langle 8\rangle$ to both v_1 and v_2, we can generate duplicate join results. Thus, to preserve join semantics, the tuple routing scheme must be carefully designed to satisfy the correlation constraint.

We have proved that any semantics-preserving tuple routing algorithm for generic MWSJ queries (including both equijoins and non-equijoins) needs to either replicate some tuples on multiple hosts or route intermediate results between different hosts, which is called *diffusion overhead* [14, 15]. Thus, the *optimal load diffusion* problem can be defined as *routing each input stream tuple to one or more hosts via single or multiple hops such that (1) correlation constraint is satisfied, (2) workload of different hosts is balanced, and (3) diffusion overhead is minimized.*

The diffusion operator spreads join workload by adaptively routing input stream tuples to different hosts for parallel join processing. To achieve realtime stream processing, we strive to keep the diffusion operator fast, simple, and light-weight. We propose two different semantics-preserving tuple routing algorithms, *aligned tuple routing* (ATR) and *coordinated tuple routing* (CTR), to accommodate different types of MWSJ queries. ATR dynamically selects one stream as *master stream* whose tuples are distributed merely based on host load conditions. All the other streams align their tuple routing with the master stream to meet the correlation constraint, which are called *slave streams*. In contrast, CTR formulates the semantics-preserving tuple routing problem into a weighted minimum set cover problem, which routes each input tuple to a minimum set of least-loaded hosts that can cover all the correlated tuples. After a tuple is routed to a host, the join operator on that host processes the tuple using a pre-defined centralized stream join algorithm (e.g., load-shedding-enabled stream joins [23], m-joins [28], and hash joins). The join order can be dynamically decided based on the estimated join selectivity between different streams[28, 2, 26].

From the tuple routing point of view, ATR performs one-hop semantics-preserving tuple routing without transferring any intermediate join results between hosts. In contrast, CTR performs multi-hop semantics-preserving tuple routing where intermediate join results may be transferred among different hosts to generate final join results. From the workload partition point of view, ATR implements stream partitions where each host only processes a subset of input stream tuples. In contrast, CTR implements both stream partitions and operator partitions, which allows an m-way join query to be performed by a set of smaller k-way join operators ($k < m$). We did not explore in-depth the third alternative tuple routing solution that only implements operator partitions because (1) it cannot spread two-way stream joins that can be resource-intensive as well [23]; and (2) it has an inherent limitation that an n-way MWSJ query can only use at most (n-1) hosts. In contrast, both ATR and CTR allow an MWSJ query to utilize any number of hosts. To accommodate dynamic stream environments, both ATR and CTR perform continuous on-line adaptations to maintain optimal query processing performance .

3 Aligned Tuple Routing Algorithm

Given an MWSJ query $S_1[W_1] \bowtie S_2[W_2]... \bowtie S_n[W_n]$, ATR dynamically selects one input stream $S_M, 1 \leq M \leq n$ as the *master stream* and aligns the tuple routing of the other $n-1$ streams called *slave streams* with the master stream for preserving join semantics, illustrated by Figure 3.

Master stream routing. The master stream S_M is partitioned into *disjoint* segments that are routed to different hosts for join processing. In Figure 3, S_1 is the master stream before time $t = 11$ and S_2 becomes the master stream afterwards. When a master stream tuple s_M arrives, we first check whether s_M belongs to the current segment according to its time-stamp. If so, we send s_M to the least-loaded host selected at the beginning of the segment (i.e., time t for the segment $S_i[t, t+T]$. Otherwise, this tuple marks the start of a new segment and the cur-

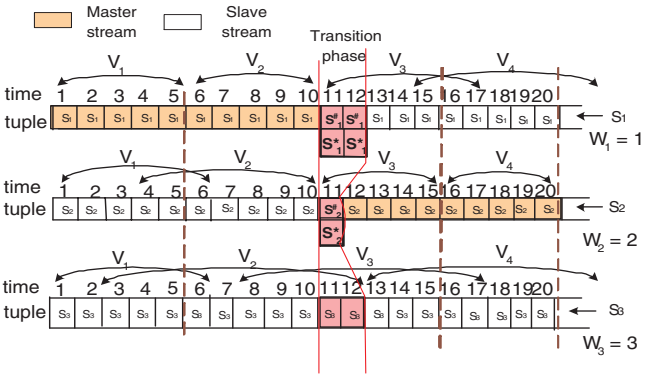

Figure 3. Aligned tuple routing scheme.

rent least-loaded host is selected as the routing destination of the new segment. Our scheme comprehensively considers all important resources (i.e., CPU, memory, bandwidth) in the distributed stream processing system by employing a combined metric w_i to represent the host load condition. For each resource type R_i, we define a load indicator $\phi_{R_i} = \frac{U_{R_i}}{C_{R_i}}$, where U_{R_i} and C_{R_i} denote the usage and capacity of the resource R_i on the host v_i, respectively. We then define the load value w_i as follows,

$$w_i = \omega_1 \phi_{cpu} + \omega_2 \cdot \phi_{memory} + \omega_3 \cdot \phi_{bandwidth} \quad (1)$$

where $\sum_{i=1}^{3} \omega_i = 1, 0 \leq \omega_i \leq 1$ denotes the importance of different resource types that can be dynamically configured by the system[2].

Slave stream routing. Each slave stream $S_i, 1 \leq i \leq n, i \neq M$ is partitioned into *overlapped* segments to preserve the MWSJ semantics, shown by Figure 3. For a master stream segment $S_M[t, t+T]$ whose tuples are routed to a host v_i, ATR routes the slave stream tuples in the segments $S_i[t - W_i, t + T + W_M], 1 \leq i \leq n, i \neq M$ to the same host v_i[3]. Similarly, if ATR sends the next master stream segment $S_M[t+T, t+2T]$ to a host v_j, ATR needs to send $S_i[t + T - W_i, t + 2T + W_M]$ to the same host v_j. Thus, the tuples arrived at S_i between the time period $[t+T-W_i, t+T+W_M]$ are replicated on both v_i and v_j. For example, in Figure 3, the tuples in $S_2[4, 7]$ are routed to both hosts V_1 and V_2.

The overhead of ATR is caused by the partial replication of the slave streams. For every segment $S_M[t, t+T]$, the total number of replicated tuples is $\sum_{i=1,i\neq M}^{n} (W_i + W_M) \cdot r_i$. For an MWSJ query $J = S_1[W_1] \bowtie ... \bowtie S_n[W_n]$, we define the overhead of ATR (O_{ATR}) as the average number of extra tuples generated per time unit[4], which can be calculated as follows,

$$O_{ATR} = \sum_{i=1,i\neq M}^{n} \frac{(W_i + W_M)}{T} \cdot r_i \quad (2)$$

The above equation indicates that the master stream selection S_M and segment length T can affect the ATR overhead. In dynamic stream environments where input stream rates can vary over time, ATR performs online adaptations to maintain optimal query processing, which are briefly described as follows due to the space limitation[5].

Master stream switching. When input stream rates experience changes or the query modifies the sizes of sliding-windows, ATR triggers a stream role switching algorithm to elect the stream that can minimize O_{ATR} as the new master stream. For example, in Figure 3, the master stream is changed from S_1 to S_2 at time $t_s = 11$. ATR employs a transition algorithm to preserve join semantics while performing dynamic stream role switching. Let t_s denote the switching time, S_M denote the old master stream, and S_N denote the new master stream. The transition durations for S_M and S_N are $[t_s, t_s + W_N)$ and $[t_s, t_s + W_M)$, respectively. For any tuple of S_M (or S_N) arrived during its transition phase, ATR sends a marked tuple s_M^* (or s_N^*) to the host selected for the old master stream segment $S_M[t_s - T, t_s]$ and a marked tuple $s_M^\#$ (or $s_N^\#$) to the host selected for the new master stream segment $S_N[t_s, t_s + T]$. Different from unmarked tuples, s_i^* only joins with tuples arrived before t_s and $s_i^\#$ only joins with tuples arrived after t_s. For any other streams $S_i, i \neq M, N$, they follow the ordinary ATR algorithm except sending $S_i[t_s - T - W_i, t_s + MAX(W_M, W_N)]$ instead of $S_i[t_s - T - W_i, t_s + W_M]$ to the hosts selected for the old master stream segment $S_M[t_s - T, t_s]$. We have proved that the above stream role adaptation algorithm can preserve join accuracy [14, 15].

Segment length adaptation. Equation 2 indicates that a larger segment incurs less diffusion overhead. However, larger segment also implies coarser load balancing granularity. Thus, the optimal segment length, denoted by T^*, should be dynamically decided to achieve best tradeoff between load balancing granularity and replication overhead. Figure 4 shows the performance of the ATR algorithm as

[2]We can assign higher weight to CPU resource if the join computation is CPU-bound or assign higher weight to network resource if network bandwidth is limited.

[3]The diffusion operator needs to buffer the tuples in $S_i[t - W_i, t]$ since its destination host is not selected until the time t (i.e., the beginning of the master stream segment $S_M[t, t+T]$). We can eliminate this buffering requirements for all slave streams by selecting the host for $S_M[t, t+T]$ at time $t - W_{max}$ instead of the time t, where W_{max} denotes the largest siding window among all slave streams.

[4]We can easily derive memory, CPU, and network bandwidth cost from the overhead tuple number. For example, let α denote the average tuple size. Then, ATR incurs on average $\alpha \cdot O_{ATR}$ memory overhead in the system.

[5]In [14], we presented detailed adaptation algorithms for two-way stream joins, which is similar to the ATR algorithms.

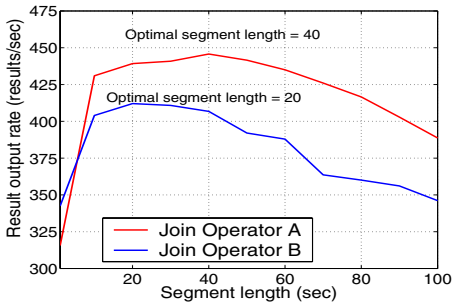

Figure 4. Segment length tuning.

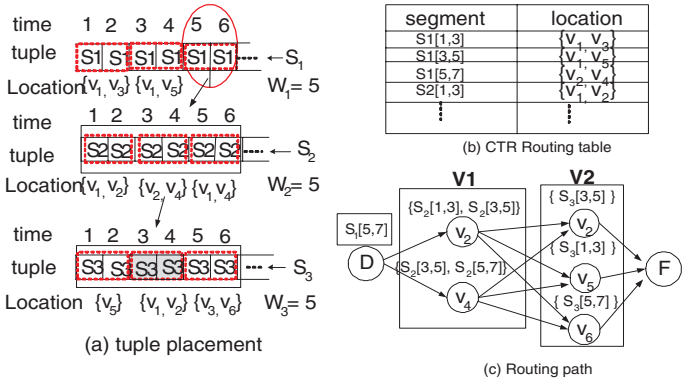

Figure 5. Coordinated tuple routing scheme.

a function of segment length for two different join operators. Currently, ATR employs a sampling-based profiling algorithm to find T^* as follows. Let T denote the current segment length and ΔT denote the sampling step value. ATR first changes the segment length to $T + \Delta T$ at the end of the current segment. If the system performance (e.g., throughput) improves, ATR gradually increases the segment length until the measured system performance reaches its peak value. Otherwise, if $T - \Delta T$ produces better performance, the system gradually decreases the segment length to search for T^*. We have proved that the online segment length adaptation can preserve the accuracy of MWSJ query results [14, 15].

4 Coordinated Tuple Routing Algorithm

CTR provides load diffusion for MWSJ queries by dynamically routing each input tuple to a minimum set of least-loaded hosts that can cover all the correlated tuples, illustrated by Figure 5. Different from ATR, CTR does not require strict alignment among different streams. Instead, each tuple s_i can independently choose its routing destinations based on the placement of other stream tuples that must be joined with s_i. Thus, CTR needs to maintain a routing table recording the placement of previously routed tuples. CTR groups tuples into segments and routes each segment as a whole to different hosts to control the routing table size and routing computation time. CTR calculates the routing path for each segment at the beginning of that segment. All the tuples in that segment follow the same routing path. A segment entry is deleted from the routing table if it does not needed by any other streams according to the MWSJ semantics. Figure 6 shows the pseudo-code of the CTR algorithm. When a tuple s_i arrives, CTR first checks whether s_i belongs to the current segment according to its time-stamp. If so, CTR retrieves the routing path P for the segment from the routing table, inserts P into the header of s_i, and routes s_i to a set of hosts according to the routing path. Otherwise, the tuple s_i marks the start of a new segment $S_i[t, t+T]$. CTR calculates the routing path

Procedure: $CTR(S_1[W_1] \bowtie ... \bowtie S_n[W_n], \{v_1, ..., v_m\})$
1. **while** receiving a tuple $s_i \in S_i$
2. **if** $s_i.t \geq (t+T)$ /*start a new segment*/
3. **for** a set of possible join orders $S_{i_1}[W_{i_1}] \rightarrow ...S_{i_{n-1}}[W_{i_{n-1}}]$
4. **for** $k = 1$ **to** $n-1$
5. calculate minimum set cover V_k for $S_{i_k}[W_{i_k}]$
6. derive one routing path $P_i = V_1 \rightarrow ...V_{n-1}$
7. select optimal P with minimum overhead (Equ. 3)
8. annotate P for duplication avoidance
9. set the routing path of $S_i[t, t+T]$ as P
10. $t \leftarrow t + T$ /*update the segment start time*/
11. $loc(S_i[t, t+T]) = V_1$ /*update routing table*/
12. **else** /* continue the current segment*/
13. annotate s_i with the routing path P
14. send a copy of s_i to each host in V_1

Figure 6. Coordinated tuple routing algorithm.

for the new segment as follows,

Step 1: Lookup correlated tuples. According to the MWSJ semantics, the tuples in $S_i[t, t+T]$ need to join with the sliding-windows of the other $n-1$ streams $S_{i_k}[W_{i_k}] = S_k[t - W_k, t + T], 1 \leq k \leq n, k \neq i$. To calculate the optimal routing path for $S_i[t, t+T]$, CTR first looks up the locations of the correlated tuples contained in $S_k[W_k], 1 \leq k \leq n, k \neq i$ in the routing able. For example, in Figure 5, CTR computes the route for the segment $S_1[5, 7]$. CTR first gets the locations of all correlated tuples in $S_2[1, 7]$ and $S_3[1, 7]$ (i.e., $S_2[1, 3]$ on two hosts $\{v_1, v_2\}$, $S_2[3, 5]$ on two hosts $\{v_2, v_4\}$, etc.).

Step 2: Calculate optimal routing path. The goal of the optimal routing path $P \triangleq V_1 \rightarrow ...V_{n-1}$ is to generate join results involving $s_i \in S_i[t, t+T]$ with minimum overhead. Given a join order (i.e., join probing sequence [28]) $s_i \rightarrow S_{i_1}[W_{i_1}]... \rightarrow S_{i_{n-1}}[W_{n-1}]$, the k-th routing hop $V_k, 1 \leq k \leq n-1$ consists of a set of hosts that can cover all the tuples contained in the sliding window

$S_{i_k}[W_{i_k}]$. For example, Figure 5 (c) shows a routing path for the segment $S_1[5,7]$. Interestingly, the optimal routing path calculation can be formulated into the *weighted minimum set cover* problem [5]. The detailed algorithm about this step will be presented in Section 4.1.

Step 3: Annotate the routing path to avoid duplicates. After deriving the complete routing path $P \triangleq V_1 \rightarrow ...V_{n-1}$, CTR needs to add annotations in P to avoid redundant join computations. Let us consider one routing hop V_k. CTR needs to send a copy of χ_{k-1} to each host in V_k, where $\chi_0 = s_i \in S_i[t, t+T]$ and $\chi_k = s_i \bowtie S_{i_1}[W_{i_1}]... \bowtie S_{i_{k-1}}[W_{i_{k-1}}], k > 1$. If any segment $S_{i_k}[t_k, t_k+T] \subseteq S_{i_k}[W_{i_k}]$ is replicated on a set of hosts U_z and there exist multiple common hosts between V_k and U_z (i.e., $|V_k \cap U_z| \geq 2$), the join computations between χ_{k-1} and $S_{i_k}[t_k, t_k+T]$ are redundantly computed since tuples in both χ_{k-1} and $S_{i_k}[t_k, t_k+T]$ are replicated on multiple hosts. For example, in Figure 5 (c), the tuples in $S_1[5,7]$ are routed to $\{v_2, v_4\}$ in the first routing hop and the segment $S_2[3,5]$ is replicated on both hosts v_2 and v_4. Thus, the join computation between $s_1 \bowtie S_2[3,5], \forall s_1 \in S_1[5,7]$ are calculated twice on both hosts. To address this problem, we annotate the routing path P to void some join computations. For any segment $S_{i_k}[t_k, t_k+T] \subseteq S_{i_k}[W_{i_k}]$, if $|V_k \cap U_z| \geq 2$, we select the least-loaded host $v_j \in V_k \cap U_z$ to execute the join computation between χ_i and $S_{i_k}[t_k, t_k+T]$. For any other hosts $v'_j \in V_k \cap U_z$, we annotate P with a flag $(v'_j / S_{i_k}[t_k, t_k+T])$ to void the join computation between χ_{k-1} and $S_{i_k}[t_k, t_k+T]$ on v'_j.

Step 4: Update routing table. CTR updates its routing table to record the placement information of the segment $S_i[t, t+T]$. Since the tuples in $S_i[t, t+T]$ are first routed to the hosts contained in V_1, the routing table records that the tuples in $S_i[t, t+T]$ are located on the hosts in V_1. Note that the routing table does not record the locations of intermediate join results since intermediate join results are not stored. Alternatively, we may also selectively store some intermediate results similar to the STAIR operator [8].

The overhead of CTR consists of two parts: (1) *replication overhead* by sending each tuple to multiple hosts in V_1; and (2) *intermediate join result overhead* by transferring intermediate join results χ_k from V_{k-1} to $V'_k = V_k - V_k \cap V_{k-1}$. Thus, for each segment $S_i[t, t+T]$ with a join order $S_i \rightarrow S_{i_1}[W_{i_1}]... \rightarrow S_{i_{n-1}}[W_{i_{n-1}}]$, a routing path $P \triangleq V_1 \rightarrow ...V_{n-1}$, and join selectivity $\sigma_{i_{k-1}, i_k}, 1 \leq k \leq n$, the number of extra data transferred over networks by CTR, denoted by $O_{CTR,P}$, can be calculated as follows,

$$O_{CTR,P} = r_i T \cdot (|V_1| - 1 + \sum_{k=2}^{n-1} \prod_{j=1}^{k-1} \sigma_{i_{j-1}, i_j} r_{i_j} W_{i_j} |V'_k|) \quad (3)$$

For an MWSJ query $J = S_1[W_1] \bowtie ... \bowtie S_n[W_n]$, we define the average overhead of CTR (O_{CTR}) as the average number of extra data transferred over networks by CTR per time unit, which can be calculated as follows,

$$O_{CTR} = \sum_{i=1}^{n} r_i (|V_1| - 1 + \sum_{k=2}^{n-1} \prod_{j=1}^{k-1} \sigma_{i_{j-1}, i_j} r_{i_j} W_{i_j} |V'_k|) \quad (4)$$

The goal of the optimal routing path calculation is to minimize the above load diffusion overhead, which will be described in the next section. At the beginning when a few tuples have been distributed, the host set V_k derived based on the correlation constraint can be empty or include very few hosts. During this warmup phase, the host set V_k is filled with a set of hosts selected based on the load condition only[6]. Different from ATR, the segment length of CTR does not affect its overhead (Equation 4). Thus, we should use small segment length to achieve fine-grained load balancing. However, smaller segment lengths can increase the routing table size and the minimum set cover computation time. Thus, we should select the minimum segment length according to the resource constraints of the diffusion operator. Similar to ATR, we can also dynamically adjust the segment length based on stream rate changes (e.g., use larger segment lengths for slower-rate streams). Similar to ATR, CTR also has provable correctness guarantee [15].

4.1 Optimal Routing Path Selection

The goal of the optimal routing path selection is to produce join results with minimum overhead denoted by O_{CTR}, which can be formulated into the *weighted minimum set cover* problem [5]. Suppose the join processing between s_i and $S_k[W_k], 1 \leq k \leq n, k \neq i$ can be performed in m different join orders. Let us consider one of the join orders $s_i \rightarrow S_{i_1}[W_{i_1}]... \rightarrow S_{i_{n-1}}[W_{i_{n-1}}]$. The *k-th* join probing is to produce all the join results between $\chi_{k-1} = s_i \bowtie S_{i_1}[W_{i_1}]... \bowtie S_{i_{k-1}}[W_{i_{k-1}}]$ and $S_{i_k}[W_{i_k}]$. Thus, the *k-th* routing hop is to select an optimal host set V_k as the routing destinations of χ_{k-1}, which needs to consider several factors. First, the optimal host set V_k should cover all the tuples in $S_{i_k}[W_{i_k}]$ to preserve the completeness of join results. Second, we want to find a *minimum* set of hosts V_k that can cover all the tuples in $S_{i_k}[W_{i_k}]$ to achieve minimum diffusion overhead. Third, we want to route input tuples to least-loaded hosts to achieve load balancing. Finally, we want to maximize the number of overlapping hosts between two consecutive routing hops $V_{k-1} \cap V_k$ to minimize the overhead of transferring intermediate join results between different hosts. Thus, when we calculate the *k-th* routing hop, we first remove those segments in $S_{i_k}[W_{i_k}]$ that is covered by the previous hop V_{k-1}. We then calculate the optimal host set cover for the remaining segments

[6]The number of hosts can affect the warmup speed. In our experiments, we observe that 5 hosts can be sufficient to quickly utilize all hosts in a 100-node cluster.

in $S_{i_k}[W_{i_k}]$. For example, in Figure 5 (c), CTR decides to route $S_1[5,7]$ to the first set of hosts $V_1 = \{v_2, v_4\}$. The intermediate join results $\chi_1 = S_1[5,7] \bowtie S_2[1,7]$ produced by the hosts $\{v_2, v_4\}$ will be routed to $V_2 = \{v_2, v_5, v_6\}$ to join with $S_3[3,5]$. Since χ_1 is already at v_2, it can join with $S_3[3,5]$ directly without inter-host transferring.

We now formally define the optimal host set selection problem. Let $E_{i_k} = \{e_{i_k,1} = S_{i_k}[t - W_{i_k}, t - W_{i_k} + T], e_{i_k,2} = S_{i_k}[t - W_{i_k} + T, t - W_{i_k} + 2T], ..., e_{i_k,J} = S_{i_k}[t, t + T]\}, J = [W_{i_k}/T]$, denote the segments contained in $S_{i_k}[W_{i_k}] = S_{i_k}[t - W_{i_k}, t + T]$. CTR first removes those segments that can be covered by the hosts included in the previous routing hop V_{k-1}. Next, CTR checks the routing table to retrieve the placement of all the remaining segments in E_{i_k}. CTR then transforms the segment placement information into host coverage information as follows: If a stream segment $e_{i_k,j}$ is placed on a set of hosts U_j, we say that any host in U_j covers the segment $e_{i_k,j}$. For example, in Figure 5 (a), the segment $S_2[1,3]$ is placed on $U_1 = \{v_1, v_2\}$. Thus, we say that the hosts v_1 and v_2 cover the segment $S_2[1,3]$. Let us denote $\mathbf{U} = \bigcup_{1 \le j \le J} U_j$. Each host $v_j \in \mathbf{U}$ covers a subset of all segments, denoted by $A_j \subseteq E_{i_k}$. For example, in Figure 5 (a), the host v_2 covers a segment subset $A_2 = \{S_2[1,3], S_2[3,5]\} \subset \{S_2[1,3], S_2[3,5], S_2[5,8]\}$. For load balancing, we associate a weight value w_i to each subset A_i that is defined as the load value of the host v_i calculated by Equation 1. Thus, we can formulate the optimal host set selection problem into a weighted minimum set cover problem: *Given a stream segment set E, segment subsets $A_1, ..., A_K \subseteq E$, and cost w_j for each subset A_j, we want to find a minimum set cover $I \subseteq \{1, ..., K\}$ such that $\cup_{j \in I} A_j = E$ and $\sum_{j \in I} w_j$ is minimum.*

The minimum set cover problem is a well-known NP-hard problem [5]. Thus, CTR uses a fast greedy heuristic algorithm to find the minimum set cover [5]. The basic idea is to select a subset A_j that has the minimum value of $\frac{w_j}{|A_j|}, A_j \neq \emptyset$, where $|A_j|$ denotes the cardinality of the set A_j. We then add A_j into the set cover I and update each remaining subsets by removing those elements included in A_j. The above process is repeated until the selected set cover I includes all the segments in E. The optimal host set V_k should include those hosts whose indices are included in I plus the reused hosts included in the previous hop V'_{k-1}. We repeat the optimal host set calculation for all $n-1$ routing hops to derive the routing path $P \triangleq V_1 \rightarrow ...V_{n-1}$ for the given join order $S_i[t, t+T] \rightarrow S_{i_1}[W_{i_1}] ... \rightarrow S_{i_{n-1}}[W_{i_{n-1}}]$. Using Equation 3, we can calculate the overhead of CTR using the routing path P. We repeat the above calculation for a set of possible join orders[7] and select the best routing path that incurs minimum overhead.

5 Experimental Evaluation

We have implemented a prototype of the semantics-preserving tuple routing framework on top of our distributed stream processing infrastructure System S [18]. The cluster system consists of about 250 blade servers connected by gigabips networks. Each host has an Intel Xeon 3.2GHZ CPU and 3G memory. We have implemented a video correlation application [14] on top of our system to search similar video shots across different news video streams for hot topic detection.

We first use six hosts to process an MWSJ query $J_1 = S_1[60] \bowtie S_2[60] \bowtie S_3[60]$, where S_i are real news video streams taken from NIST TRECVID-2005 data set. Each host executes a modified version of load-shedding-enabled windowed stream join algorithm [23]. The diffusion and fusion operators run on two other separate hosts. The join predicate is whether two video images are close to each other in a 40-dimensional concept space. Figure 7 shows the throughput of total join results generated by different algorithms during a 1200-second duration. The *semantics-unaware routing* algorithm always routes tuples to least-loaded hosts. The *centralized* algorithm executes the join query on one host. The throughput value is sampled every second where the *total throughput* at time t measures the total number of join results generated by the system from the beginning to the time t. We observe that both ATR and CTR can achieve higher throughput than the other two alternatives. ATR performs better than CTR for processing J_1. We then repeat the above experiment using a different join operator $J_2 = S_1[120] \bowtie S_2[120] \bowtie S_3[120]$. Figure 8 shows the throughput results. We observe that both ATR and CTR can still achieve much better performance than other alternatives. CTR performs better than ATR in this case since it has smaller overhead for stream joins with large sliding-windows.

Figure 9 and Figure 8 show the computation overhead of ATR and CTR during the two experiments. We observe that both ATR and CTR have low computation overhead (i.e., tens of micro-seconds), which is several orders of magnitudes less than the join computations (i.e., tens of or hundreds of milli-seconds). In terms of memory requirement. ATR has no extra memory requirement except buffering input tuples to correct out-of-order tuple arrivals due to network delay jitter. CTR needs additional memory to store the routing table, which is at most several Kilobytes in our experiments.

We conduct more extensive simulation experiments using various join workloads. The simulator consists of a

[7] The number of all possible join orders can be large given a large stream number n. In that case, we can select a limited number of good join orders based on estimated join selectivity between different streams.

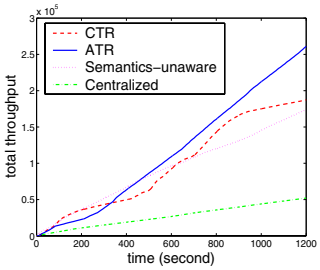

Figure 7. Throughput results for J_1.

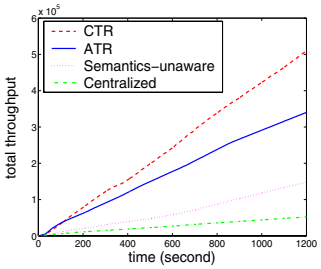

Figure 8. Throughput results for J_2.

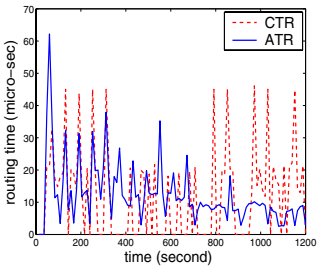

Figure 9. Routing time for J_1.

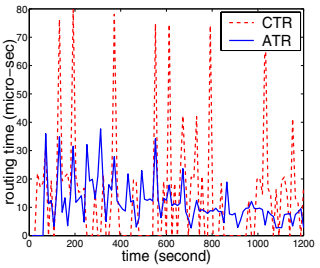

Figure 10. Routing Time for J_2.

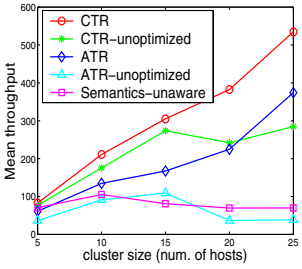

Figure 11. Scalability results.

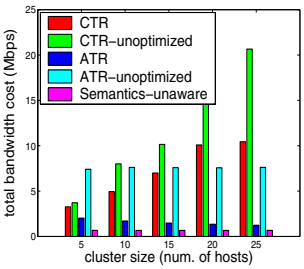

Figure 12. Network bandwidth results.

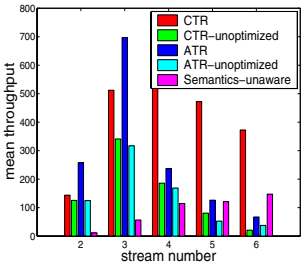

Figure 13. Stream number effect.

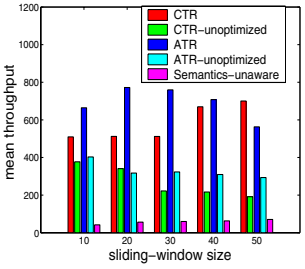

Figure 14. Window size effect.

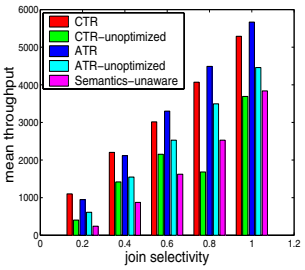

Figure 15. Join selectivity effect.

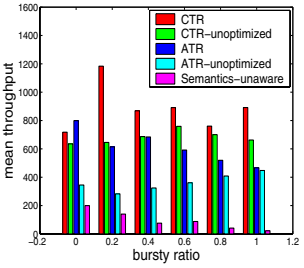

Figure 16. Stream burstiness effect.

workload generator, diffusion operators, fusion operators, and multiway stream join operators. All operators are fully implemented. Only the hardware, network, and underlying scheduler in the cluster system are simulated to enable easy control and large-scale experiments. We simulate a heterogeneous cluster system where the memory and CPU capacity of each host is uniformly distributed in the range of [500,1000] MB and [1000,5000] MIPS, respectively. The network bandwidth between cluster nodes is in the range of [100,1000] Mbps. The mean CPU time for receiving or sending a tuple from or to the network is 1 ms, and one join probing takes on average 10 ms. To quantify the effectiveness of our optimization algorithms, we also implement *ATR-unoptimized* that does not perform master stream switching and segment length optimization, and *CTR-unoptimized* that performs random host set selection for covering all correlated tuples. Unless otherwise specified, all simulation experiments are conducted on a machine with 1.6GHz Pentium CPU and 1GB physical memory. Each simulation run lasts 1000 seconds and has a certain warm-up period for the system to reach its stable performance. We repeat each experiment 5 times with different random seeds and report the average results. We use the *mean throughput* as the performance metric that denotes the average number of join results produced by the system per second.

Figure 11 shows the scalability results of different algorithms for processing $J_3 = S_1[30] \bowtie S_2[30] \bowtie S_3[30] \bowtie S_4[30]$ using different numbers of hosts. The tuple arrivals of each input stream follow a Poisson process with a stream rate r_i uniformly distributed in the range [5,20] tuples/second. The mean join selectivity is $\sigma = 0.1$ between every two streams. We observe that both ATR and CTR can achieve better scalability than the existing distribution solution. The optimizations are effective, especially under large clusters. Figure 12 shows the network bandwidth usage of different algorithms. The results show that both ATR and CTR can have much lower bandwidth cost than their un-optimized versions.

We now use different types of MWSJ queries to illustrate the trade-offs between ATR and CTR. We first evaluate the effect of stream number by running different join queries $J_k = S_1[20] \bowtie ... \bowtie S_k[20], 2 \leq k \leq 6$ on a 30-host cluster, shown by Figure 13. The results show that ATR has better performance for 2-way and 3-way join queries while CTR works better for larger stream numbers. We then evaluate the effect of sliding-window size, shown by Figure 14. We run a join query $J_i = S_1[W_i] \bowtie S_2[W_i] \bowtie S_3[W_i]$ with different sliding-windows $10 \leq W_i \leq 50$ on the 30-host cluster. We observe that ATR works better under small sliding windows while CTR works better under large sliding windows. The reason is that the overhead of ATR is proportional to the sliding window size while CTR is not. Moreover, CTR allows each stream to partition its sliding-window while ATR can only partition the window of the master stream. We now study the performance of ATR and CTR under different join selectivity, shown by Figure 15. We execute a three-way join query $J = S_1[30] \bowtie S_2[30] \bowtie S_3[30]$ on the same 30-node cluster with an increasing join selectivity $0.2 \leq \sigma \leq 1.0$. The results show that CTR works better under low join selectivity while ATR becomes better under high join selectivity. The reason is that CTR has larger intermediate result overhead under high join selectivity. Finally, we test our algorithms under bursty streams. The tuple arrival in a bursty stream follows a HIGH/LOW model while a burst of data are generated during the HIGH-period with a high stream rate and zero or a few data are generated during the LOW-period. We define a bursty ratio metric $\theta, 0 \leq \theta \leq 1$. The high rate and low rate are calculated by $\frac{1+\theta}{2} r_i$ and $\frac{1-\theta}{2} r_i$, respectively. When the bursty ratio is 0, the bursty stream becomes a normal dynamic stream while if the bursty ratio is 1, the bursty stream has r_i high rate and 0 low rate, which becomes conventional ON/OFF stream. We still use the three-way join query $J = S_1[30] \bowtie S_2[30] \bowtie S_3[30]$ executing on the 30-host cluster. We observe that CTR can achieve better performance under bursty stream environments. We also conduct the multi-operator experiments, which show that our scheme can still achieve better performance than existing distribution schemes. Due to the space limitation, we omit the results here. In summary, both ATR and CTR can consistently achieve better performance than existing load distribution solutions and their un-optimized versions for different join queries. The relative merit between ATR and CTR depends on a set of factors including the join size (i.e., the number of input streams), sliding-window size, join selectivity, and stream types.

6 Related Work

The original Eddies paper [2] proposes an aggressive operator reordering mechanism by monitoring the dataflow rates into and out of operators and routing tuples through operators based on those observations. Tian *et al.* extend the Eddies framework to the distributed environment. Madden *et al.* [21] considers computation sharing across different queries. A more recent Eddies paper [7] reduces runtime overhead by routing tuples in batches. The STAIRs operator [8] allows the query engine to manipulate the state stored inside the operators and undo the effects of past routing decisions. Our tuple routing solution is inspired by the above work but distinguishes itself by considering the MWSJ semantics and using tuple routing to achieve fine-grained load balancing in distributed stream processing systems.

The Flux operator [22] extends the Exchange operator [12] to support parallel CQ processing with dynamic value-based load balancing. In contrast, our scheme provides *value-independent* load balancing, which has several advantages: (1) avoid parsing stream tuple content to allow fast in-kernel implementation of a stream tuple router; (2) do not have to deal with the data skew problem; and (3) apply to both equijoins and non-equijoins. Ivanova and Risch proposed a customizable parallel execution platform for scientific stream queries [17]. Our work is similar to the above work in terms of considering query semantics. However, our work focuses on providing load balancing for MWSJ queries and provides the first semantics-preserving tuple routing framework. The Borealis project developed a dynamic inter-operator load distribution algorithm utilizing the operators' load variance coefficients [29]. In contrast, our work provides intra-operator load distribution for MWSJ queries.

Golab *et al* evaluated various join algorithms over multiple streams, and developed a join ordering heuristic based on a per-unit-time cost model [11]. The XJoin operator [27] addresses the problem of the streaming inputs by replacing blocking operators with streaming symmetric operators. XJoin addresses the memory overflow problem by spilling some inputs to disk. The MJoin operator [28] generalized the streaming binary join algorithms and demonstrated that multiway stream joins can be implemented in a more efficient way than using a tree of binary join operators. Hammad et al. proposed an efficient scheduling scheme to optimize multiple windowed joins over a common set of data streams [16]. Babu *et al.* proposed an adaptive ordering algorithm for multiway stream joins [4]. Tao *et al.* proposed a rate-based progressive join (RPJ) algorithm to maximize the stream join output rate according to the characteristics of the join relations [24]. Early hash join is another fast join algorithm that can achieve customizable tradeoff between join output rate and overall execution time [20]. Our work is different from the above work by providing adaptive load diffusion for MWSJ query processing.

7 Conclusion

In this paper, we have presented a novel adaptive load diffusion operator to achieve scalable processing of MWSJ queries. We propose the first semantics-preserving tuple routing framework that can dynamically distribute join workload at fine granularity without losing join accuracy. We have developed two different tuple routing algorithms, aligned tuple routing (ATR) and coordinated tuple routing (CTR), to accommodate different types of MWSJ queries. Both ATR and CTR can scale-up MWSJ processing using an arbitrary number of hosts, and perform on-line adaptations to maintain optimal performance in dynamic stream environments. Prototype experiments show that our algorithms can efficiently scale-up MWSJ processing with low overhead. We view our work as the first step towards employing semantics-preserving tuple routing to support scalable distributed CQ processing. Possible future work includes (1) generalizing the tuple routing framework to support other CQ semantics, and (2) extending the distributed MWSJ framework to other distributed computing environments such as P2P networks and wireless sensor networks.

References

[1] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *Standford University Technical Report 2003-67*, 2003.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *Proc. of SIGMOD*, 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motawani, and J.Widom. Models and Issues in Data Stream Systems. *Proc. of PODS*, 2002.

[4] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive Ordering of Pipelined Stream Filters. *Proc. of SIGMOD*, 2004.

[5] V. Chvatal. A Greedy-Heuristic for the Set Covering Problem. *Mathematics of Operations Research, 4:233-235*, 1979.

[6] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. *Proc. of SIGMOD*, 2003.

[7] A. Deshpande. An Initial Study of Overheads of Eddies. *Proc. of SIGMOD Record*, 2004.

[8] A. Deshpande and J. M. Hellerstein. Lifting the Burden of History from Adaptive Query Processing. *Proc. of VLDB*, 2004.

[9] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing Data-Stream Join Aggregates Using Skimmed Sketches. *Proc. of EDBT*, 2004.

[10] L. Golab and M. Tamer Özsu. Issues in Data Stream Management. *Proc. of VLDB*, 2003.

[11] L. Golab and M. Tamer Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. *Proc. of VLDB*, 2003.

[12] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. *Proc. of SIGMOD*, 1990.

[13] The STREAM Group. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin, 26(1)*, 2003.

[14] X. Gu, Z. Wen, C.-Y. Lin, and P. S. Yu. ViCo: An Adaptive Distributed Video Correlation System. *Proc. of ACM Multimedia*, 2006.

[15] X. Gu, P. S. Yu, and H. Wang. Adaptive Load Diffusion for Multiway Windowed Stream Joins. *http://www.research.ibm.com/people/x/xgu/LD.pdf, IBM Research Technical Report*, 2006.

[16] M. A. Hammad, M. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. *Proc. of VLDB*, 2003.

[17] M. Ivanova and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. *Proc. of VLDB*, 2005.

[18] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. *Proc. of SIGMOD*, 2006.

[19] S. Krishnamurthy and et al. TelegraphCQ: An Architectural Status Report. *IEEE Data Engineering Bulletin, 26(1)*, 2003.

[20] R. Lawrence. Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. *Proc. of VLDB*, 2005.

[21] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries Over Streams. *Proc. of SIGMOD*, 2002.

[22] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. *Proc. of ICDE*, 2003.

[23] U. Srivastava and J. Widom. Memory Limited Execution of Windowed Stream Joins. *Proc. of VLDB*, 2004.

[24] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: Producing Fast Join Results on Streams through Rate-based Optimization. *Proc. of SIGMOD*, 2005.

[25] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. *Proc. of VLDB*, 2003.

[26] F. Tian and D. J. DeWitt. Tuple Routing Strategies for Distributed Eddies. *Proc. of VLDB*, 2003.

[27] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 2000.

[28] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. *Proc. of VLDB*, 2003.

[29] Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. *Proc. of ICDE*, April 2005.

[30] S. Zdonik and et al. The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin, 26(1)*, 2003.