# Adaptive Load Diffusion for Stream Joins

Xiaohui Gu and Philip S. Yu

IBM T. J. Watson Research Center,
Hawthorne, NY 10532
{xiaohui, psyu}@ us.ibm.com

**Abstract.** Data stream processing has become increasingly important as many emerging applications call for sophisticated realtime processing over data streams, such as stock trading surveillance, network traffic monitoring, and sensor data analysis. Stream joins are among the most important stream processing operations, which can be used to detect linkages and correlations between different data streams. One major challenge in processing stream joins is to handle continuous, high-volume, and time-varying data streams under resource constraints. In this paper, we present a novel load diffusion system to enable scalable execution of resource-intensive stream joins using an ensemble of server hosts. The load diffusion is achieved by a simple correlation-aware stream partition algorithm. Different from previous work, the load diffusion system can (1) achieve *fine-grained* load sharing in the distributed stream processing system; and (2) produce *exact* query answers without missing any join results or generate duplicate join results. Our experimental results show that the load diffusion scheme can greatly improve the system throughput and achieve more balanced load distribution.

## 1 Introduction

Many emerging applications call for sophisticated realtime processing over data streams, such as stock trading surveillance, network traffic monitoring, and sensor data analysis. In these applications, data streams from external sources flow into a stream processing system (e.g., [5,11,12]) where they are processed by different continuous query processing elements called operators. One of the most important continuous query operators is sliding-window join between two streams $S_1$ and $S_2$, called *stream join*. The output of the stream join contains every pair of tuples (i.e., data records) $(s_1, s_2), s_1 \in S_1, s_2 \in S_2$ that satisfy a join predicate. To handle infinite streams, each stream is associated with a sliding window to limit the scope of stream joins. Indeed, for many applications, we only need to correlate each newly arrived tuple with recently arrived tuples on the other stream. The stream join can be used to detect linkages and correlations between different data streams, which has many interesting applications. For example, let us consider two data streams consisting of phone call records and stock trading records, respectively. A sliding-window join between the suspicious phone calls and anomalous trade records over the common attribute "trade identifier" can be used to generate insider trading alerts. Other application examples of stream joins include (1) correlate similar images between two news video for hot topic detection; and (2) associate measurements (e.g., temperature, chemical concentration) from different sensors for environment monitoring and problem diagnosis.

In many cases, stream applications require immediate on-line results, which implies that query processing should use in-memory processing as much as possible. However, given high stream rates and large window sizes, even a single sliding-window join operator can have large memory requirement [7]. Moreover, some query processing such as video analysis can also be CPU-intensive. Thus, a single server may not have enough resources to produce accurate join results while keeping up with high input rates. There are two basic solutions to address the challenge: shedding some workload by providing approximate query results [8,7], or offloading part of workload to other servers. Our research studies the latter approach, focusing on providing load diffusion scheme to efficiently execute stream joins using a cluster of servers connected by high-speed networks.

Distributed stream processing has recently received much research attention. In [10], Shah et al. studied intra-operator load distribution for processing a single windowed aggregation operator on multiple servers. However, their solution was not based on the sliding-window stream join model. In [9], Xing et al. proposed a dynamic load distribution framework that can provide coarse-grained load balancing at inter-operator level. However, the inter-operator load distribution alone is not sufficient since it does not allow a single operator to collectively use resources on multiple servers. In [4], we propose an optimal component composition scheme for distributed stream processing systems that can achieve both QoS support and load balancing. In [2], Balazinska et al. proposed a contract-based load management framework that can migrate workload among different stream processing sites based on pre-defined contracts. Different from the above work, this work focuses on supporting fine-grained load distribution, called *load diffusion* for sliding-window stream joins. For producing exact join results, the load diffusion system preserves a *correlation constraint* that correlated tuples must be sent to the same server.

In this paper, we present a novel load diffusion middleware system to dynamically distribute stream join workload among a cluster of servers. Our principle goal is to provide scalable stream joins by efficiently utilizing all available resources in the server cluster. To achieve the goal, we propose a simple correlation-aware stream partition algorithm called single stream partition (SSP). The SSP algorithm dynamically spreads the tuples of one stream called the master stream among all hosts for load diffusion, and replicates the other stream called the slave stream for preserving the correlation constraint. To adapt to dynamic stream environments, the SSP algorithm can adaptively switch the master stream and the slave stream according to the stream rate changes. We formally prove the correctness of the SSP algorithm and analyze its properties. The adaptation strategy is then derived based on the formal analysis.

We implement the load diffusion scheme as a middleware proxy service. The load diffusion proxy virtualizes a cluster of stream processing servers into a unified stream processing service. Analogous to previous middleware systems (e.g., [1]), the load diffusion middleware aims at providing a managed and transparent stream processing service, which hides complicated system management details from upper-level application developers. The major operation performed by the load diffusion proxy is to route tuples to proper servers according to the load diffusion algorithm and the load conditions of different servers. We have implemented the proposed load diffusion algorithms and conduct extensive experiments on a distributed stream processing simulation testbed. The experimental results show that the load diffusion scheme can greatly improve the system throughput and achieve more balanced load distribution compared to previous approaches.

The rest of the paper is organized as follows. Section 2 introduces the system model. Section 3 presents the correlation-aware stream partition algorithms. Section 4 presents an experimental evaluation. Finally, the paper concludes in Section 5.

## 2 System Model

### 2.1 Stream Processing Model

We now briefly describe the basic model of sliding-window stream joins illustrated by Figure 1 (a). A data stream, denoted by $S_i$, consists of a sequence of tuples denoted by $s_i \in S_i$. In a stream $S_i$, a variable number of tuples arrive in each time unit. We use $r_i$ to denote the average arrival rate of the stream $S_i$. In a dynamic stream environment, the stream rate $r_i$ can change over time. We assume that each tuple $s_i \in S_i$ carries a time-stamp $s_i.t$ to denote the time when the tuple arrives on 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 on $S_i$ in the time interval $[t - W_i, t]$. The basic stream join operator considered in this paper is sliding-window symmetric join between two streams $S_1$ and $S_2$ over a common attribute A, denoted by $J_i = S_1[W_1] \bowtie_A S_2[W_2]$. The output of the join consists of all

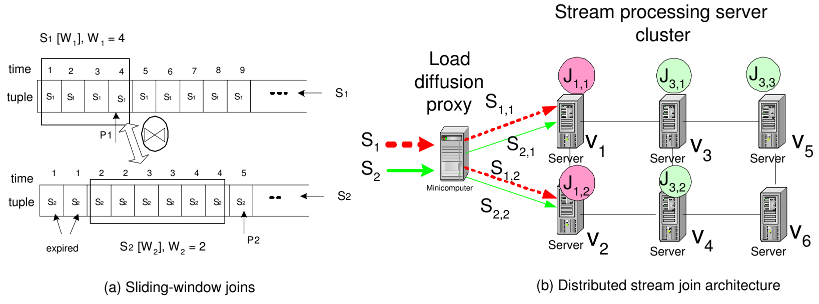

**Fig. 1.** The load diffusion system model

pairs of tuples $(s_1, s_2)$ such that $s_1.A = s_2.A$ and $s_2 \in S_2[W_2]$ at time $s_1.t$ (i.e., $s_2.t \in [s_1.t - W_2, s_1.t]$) or $s_1 \in S_1[W_1]$ at time $s_2.t$ (i.e., $s_1.t \in [s_2.t - W_1, s_2.t]$). Each processing between the two tuples $s_1$ and $s_2$ is called one join operation. Each join operator maintains two queues $Q_1$ and $Q_2$ for buffering incoming tuples from the streams $S_1$ and $S_2$, respectively. When a new tuple $s_i \in S_i, 1 \leq i \leq 2$ arrives, it is inserted into the corresponding queue $Q_i$ if $Q_i$ is not full. Otherwise, the system either drops the newly arrived tuple or replace an old tuple in the buffer with the newly arrived tuple. The tuples in both queues $Q_1$ and $Q_2$ are processed according to the temporal order, i.e., if $s_1.t \in Q_1 < s_2.t \in Q_2$, $s_1$ is processed first. Each queue $Q_i, i = 1, 2$ maintains a pointer $p_i$ to refer to the tuple currently processed by the join operator.

The sliding-window join algorithm processes a tuple $s_1 \in Q_1$ with the following steps: (1) update $Q_2$ by removing expired tuples. A tuple $s_2$ is expired if (a) it arrives earlier than $s_1.t - W_2$ and (b) it has been processed by the join operator (i.e., $p_2$ points to a tuple arrived later than $s_2$); (2) produce join results between $s_1$ and $S_2[W_2]$, denoted by $s_1 \bowtie_A S_2[W_2]$ by comparing $s_1.A$ and $s_2.A$, $\forall s_2 \in S_2[W_2]$; (3) update the pointer $p_1$ to refer to the next tuple in $Q_1$; (4) decide which tuple to process next by comparing $s_1.t$ and $s_2.t$, where $s_1$ and $s_2$ are the tuples pointed by $p_1$ and $p_2$, respectively. A symmetric procedure is followed for processing a tuple $s_2$ in the queue $Q_2$ of the stream $S_2$.

## 2.2 System Architecture

The distributed stream processing system consists of a cluster of servers connected by high-speed networks. Each server node, denoted by $v_i$, has a limited memory capacity $M_i$ for buffering tuples, and a certain CPU processing speed that can process on average $N_i$ join operations per time unit. Data streams are pushed into the distributed stream processing system from various external sources such as temperature sensors, stock tickers, and video cameras. The distributed stream processing system appears to a client as a unified stream processing service to serve a large number of continuous query processing over high volume data streams. The push-based stream environment has two unique features: (1) the tuples of a single stream can arrive in a bursty fashion (i.e., a large number of tuples can arrive in a short period of time); and (2) tuples are pushed into the system where data arrivals cannot be controlled by the system. The distributed stream processing system needs to efficiently utilize all available resources to achieve the best possible throughput for keeping up with the high arrival rates.

The architecture of the distributed stream processing system, illustrated by Figure 1 (b), consists of a load diffusion proxy and an ensemble of servers. The load diffusion proxy serves as a gateway of the distributed stream processing system to distribute stream processing workload across all servers. For each stream join request, the load diffusion proxy selects a number of servers to instantiate the join operator. The load diffusion proxy intercepts input streams and re-directs them to proper servers responsible for handling the stream joins. Due to the memory and CPU speed limits, a single server can only accommodate a certain data arrival rate in order to keep the unprocessed data in the memory. When tuples arrive too fast, the server has to drop tuples using some load shedding technique (e.g., [8]). However, dropping data can affect the accuracy of stream join results. Thus, the goal of our load diffusion scheme is to avoid dropping data as much as possible by spreading stream join workload across multiple servers.

The load diffusion proxy realizes fine-grained and balanced workload distribution using stream partitions. The stream partition algorithm can continuously split a high-volume stream into multiple substreams, each of which are sent to different servers for concurrent processing. Conceptually, the load diffusion proxy decomposes a resource-intensive join operator into multiple sub-operators executed on different servers. Each sub-operator only processes a subset of tuples on the original input streams. For example, in Figure 1 (b), the load diffusion proxy splits the stream $S_1$ into two substreams $S_{1,1}$ and $S_{1,2}$ that are sent to the server $v_1$, and $v_2$, respectively. Each substream has lower stream rate than the original stream. Different from load distribution for traditional distributed computing environments, our load diffusion scheme needs to send correlated data to the same server, which is called the correlation constraint. By observing the correlation constraint, the load diffusion proxy can maintain the full accuracy of stream joins. For example, let us consider a windowed stream join $S_1[W_1] \bowtie_A S_2[W_2]$. If the load diffusion proxy sends a tuple $s_1$ to a server node $v_i$, the correlated data include those tuples $s_2 \in S_2$ such that $s_2 \in S_2[s_1.t - W_2, t]$.

## 3 Replication-Assisted Single Stream Partition

The basic idea of the single stream partition (SSP) algorithm is to split one stream for load distribution and replicate the other stream for preserving the correlation constraint, which is illustrated by Figure 2. The partitioned stream is called the *master stream* and the replicated stream is called the *slave stream*. Each tuple of the slave stream is replicated on all the server hosts that are allocated to the join operator. Thus, we can freely partition the master stream since all the correlated tuples required by the partitioned stream are on the replicated stream, which have replicas on all server hosts. Figure 3 shows the pseudo-code of the SSP algorithm, which is described using an example as follows. Let us consider a join operator $J_i = S_1[W_1] \bowtie_A S_2[W_2]$ between the two streams $S_1$ and $S_2$ whose average arrival rates are $r_1$ and $r_2$, respectively. Suppose the sys-

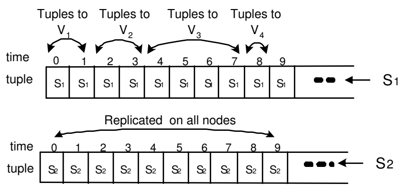

**Fig. 2.** The SSP example

```
Procedure $SSP(S_1, S_2, \{v_1, ..., v_k\})$
1. receive tuples for $S_1$ and $S_2$
3. $\forall s_1 \in S_1$, $S_1$: master stream
4.     send $s_1$ to the least-loaded host $v_b$
5. $\forall s_2 \in S_2$, $S_2$: slave stream
6.     send $s_2$ to all server hosts
```

**Fig. 3.** The SSP algorithm

tem allocates the host set $\{v_1, ..., v_k\}$ for executing the join operator $J_i$. Let us assume that $S_1$ is the master stream and $S_2$ is the slave stream. For each tuple $s_1$ arriving at the stream $S_1$, the SSP algorithm sends $s_1$ to one of the server hosts based on a certain selection policy (e.g., round-robin or least-loaded-first). For each tuple $s_2$ arrived at the stream $S_2$, the SSP algorithm replicates $s_2$ into $k$ copies, each of which is sent to the $k$ servers, respectively. By spreading the tuples of stream $S_1$ among all $k$ servers, the workload of the join operator $J_i = S_1[W_1] \bowtie_A S_2[W_2]$ is diffused among all $k$ servers since each server only processes a subset of all required join operations.

We now formally prove the correctness of the SSP algorithm. We define that a stream partition algorithm is correct if it executes the same set of join operations as the original join operator. We use $C(J_i)$ and $C'(J_i)$ to denote the sets of join operations performed by the original join operator and the join operations performed by the diffused join operator, respectively. We prove the correctness of the SSP algorithm by showing that $C(J_i) = C'(J_i)$.

**Theorem 1.** *Let $C(J_i)$ and $C'(J_i)$ denote the sets of join operations performed by the original join operator and the new join operator diffused by the SSP algorithm, respectively. We have $C(J_i) = C'(J_i)$.*

*Proof.* We first prove (1) $C(J_i) \subseteq C'(J_i)$ by showing that $\forall s_1$, if $s_1 \bowtie_A S_2[W_2] \in C(J_i)$, then $s_1 \bowtie_A S_2[W_2] \in C'(J_i)$, and $\forall s_2$, if $s_2 \bowtie_A S_1[W_1] \in C(J_i)$, then $s_2 \bowtie_A S_1[W_1] \in C'(J_i)$. Suppose the SSP algorithm sends $s_1$ to the server $v_i$. Because SSP replicates the stream $S_2$ on all servers, $S_2[W_2]$ must be present on the server $v_i$, too. Thus, $s_1 \bowtie_A S_2[W_2] \in C'(J_i)$. We now prove $\forall s_2$, if $s_2 \bowtie_A S_1[W_1] \in C(J_i)$, then $s_2 \bowtie_A S_1[W_1] \in C'(J_i)$. For any $s_2 \in S_2$, $s_2$ needs to join every tuple in $S_1[W_1]$. Suppose SSP sends $s_1 \in S_1[W_1]$ to the server $v_i$. Because $s_2$ is also present at $v_i$, we have $s_2 \bowtie_A s_1 \in C'(J_i)$. By aggregating all the results of $s_2 \bowtie_A s_1, \forall s_1 \in S_1[W_1]$, we have $s_2 \bowtie_A S_1[W_1] \in C'(J_i)$. Thus, we have $C(J_i) \subseteq C'(J_i)$. We then prove (2) $C'(J_i) \subseteq C(J_i)$ by showing that $\forall s_1$, if $s_1 \bowtie_A S_2[W_2] \in C'(J_i)$, then $s_1 \bowtie_A S_2[W_2] \in C(J_i)$, and $\forall s_2$, if $s_2 \bowtie_A S_1[W_1] \in C'(J_i)$, then $s_2 \bowtie_A S_1[W_1] \in C(J_i)$. The proof is straightforward since any join operation in $C'(J_i)$ follows the windowed join definition, which thus should appear in $C(J_i)$, too. Because $\forall s_1 \in S_1$, $s_1$ is only sent to one server, two different servers do not perform duplicated join operations. Thus, we have $C'(J_i) \subseteq C(J_i)$. Combining (1) and (2), we have $C(J_i) = C'(J_i)$. □

We now analyze the properties of the SSP algorithm. Since the number of total join operations is not changed by the SSP algorithm, each server in $\{v_1, ..., v_k\}$ only processes on average one $k'th$ of the original join operations. One advantage of the SSP algorithm is that it can achieve the finest-grained spreading for the master stream at a per-tuple basis. By splitting the master stream $S_1$ into $k$ substreams, the SSP algorithm can reduce the resource requirements for individual servers. Let $r_1$ denote the arrival rate of the master stream $S_1$. Each sub-stream of $S_1$ has an average arrival rate of $r_1/k$. Thus, in addition to reduce the processing workload, the SSP algorithm can reduce (1) memory requirement for buffering tuples in the sliding windows and (2) bandwidth requirement for receiving tuples.

**Theorem 2.** *Let $r_1$ and $r_2$ denote the rates of the two joined streams $S_1$ and $S_2$. Let $W_1$ and $W_2$ denote the sliding-window sizes of $S_1$ and $S_2$. Let $m$ denote the average tuple size. Let $k$ denote the server number. Let $\varDelta M$ and $\varDelta B$ denote the average memory reduction, average bandwidth reduction, and average processing load reduction at each server node compared to the original join operator. We have*

$$\varDelta M = \frac{k-1}{k} \cdot m \cdot r_1 \cdot W_1 \tag{1}$$

$$\varDelta B = \frac{k-1}{k} \cdot m \cdot r_1 \tag{2}$$

*Proof.* Without load diffusion, the original join operator is executed on a single server $v_i$. The server needs a memory space for buffering the tuples in the two sliding windows $S_1[W_1]$ and $S_2[W_2]$, which can be calculated as $m \cdot (r_1 \cdot W_1 + r_2 \cdot W_2)$. The server needs $m \cdot (r_1 + r_2)$ bandwidth for receiving the tuples. With load diffusion, the tuple arrival rate of the stream $S_1$ at each server is reduced to $\frac{r_1}{k}$. The memory space for buffering the tuples in the sliding windows at a single server is reduced to $m \cdot (\frac{r_1}{k} \cdot W_1 + r_2 \cdot W_2)$. The bandwidth requirement at each server is reduced to $m \cdot (\frac{r_1}{k} + r_2)$. Thus, the average memory reduction at each server is $\varDelta M = m \cdot (r_1 \cdot W_1 + r_2 \cdot W_2) - m \cdot (\frac{r_1}{k} \cdot W_1 + r_2 \cdot W_2) = \frac{k-1}{k} \cdot m \cdot r_1 \cdot W_1$. The average bandwidth reduction at each server is $\varDelta B = \frac{k-1}{k} \cdot m \cdot r_1$. □

We now analyze the overhead of the SSP algorithm. Since the SSP algorithm replicates the tuples of the slave stream $S_2$ on all allocated servers, the load diffusion proxy pushes more tuples into the server cluster than the original input streams. The system needs to spend part of CPU cycles on processing these extra tuples such as receiving the tuple from the network, extracting the time stamp and sequence number, dropping the tuple if not needed, inserting the tuple into the queue if it is useful and memory is not full, and replacing an old tuple if memory is full. We define the overhead of the SSP algorithm as the number of these extra tuples. We can easily derive that the overhead of the SSP algorithm is $(k-1) \cdot r_2$ since only $S_2$ is replicated on $(k-1)$ extra hosts. In order to minimize the algorithm overhead, the SSP algorithm adaptively selects the stream with lower rate as the master stream, and the other stream as the slave stream. The load diffusion proxy estimates the arrival rate of each stream by counting the number of arrived tuples on each stream within a sampling period. The average arrival rate of the input stream can be estimated by dividing the tuple number over the sampling period.

## 4 Experimental Evaluation

### 4.1 Experiment Setup

We have implemented the load diffusion middleware proxy that executes the proposed stream partition algorithm. We conduct experiments to evaluate the performance of the load diffusion proxy using a simulated stream processing cluster and a variety of stream join workloads. The source streams first arrive the load diffusion proxy and then directed to different server hosts for join processing. Each server executes the sliding-window join algorithm described in Section 2.1. The memory space of each server is randomly set in the range of [1000, 2000] tuples. The CPU speed of each server is distributed in the range of [1000, 5000] MIPS. Different values reflect the heterogeneity among different servers. The average CPU cost to process a join operation is set as 50 MIPS. The average CPU cost for processing each tuple upon receiving (i.e., insert the new tuple, drop the new tuple, or replace an old tuple with the new tuple) is set as 10 MIPS. The tuples on the input streams $S_i, i = 1, 2$ are generated at an average rate of $r_i$ tuples per second. We use the same tuple arrival model as [7] where the inter-arrival time is uniformly selected at random between $1/2r_i$ and $2/r_i$ time units. Our experiments use different stream rates $r_i$ to represent dynamic workloads. For comparison, we use the following metrics: (1) *throughput* that is defined as the number of join operations finished by all servers over a period of time, and (2) *effective CPU utilization* that is defined by the ratio between the CPU cycles spent on the join processing at a server per second over the server's CPU capacity. We use *LLF-Distribution* to denote the traditional least-loaded-first load distribution algorithm that instantiates a join operator on a single least-loaded server. In all experiments, we use $W_1 = W_2 = 10$ seconds. Each simulation run lasts 5000 seconds.

## 4.2 Results and Analysis

We first evaluate the scaling property of the SSP algorithm. The experiment executes 10 join operators using a 100 node heterogeneous cluster. The stream rates $r_1$ and $r_2$ of each join operator are randomly selected from the range of [5,20] tuples/second. Figure 4 shows the throughput of different algorithms as we gradually increase the number of servers allocated to each join operator. Each throughput value represents the total number of join operations performed by the system during the whole 5000-second simulation duration. The system randomly selects k servers for each join operator given the number of servers allocated to it. We observe that the SSP algorithm achieves best performance when each join operator is allocated with about 15 servers. The reason is that the overhead of the SSP algorithm increases proportionally to the number of allocated servers. In contrast, the throughput of the LLF-Distribution algorithm is unchanged during the above experiment since each join operator can only use one server. Figure 5 shows the average effective CPU utilization results as we gradually increase the number of servers allocated to each stream join. We observe that in the SSP algorithm, the average CPU utilization first increases as the algorithm spread the workload among all servers, and then decreases when more than 15 servers are used for each join operator since all servers are overwhelmed by excessive overhead tuples.

We then evaluate the load balancing property of the SSP algorithm. This experiment executes two stream join operators on a ten-node heterogeneous

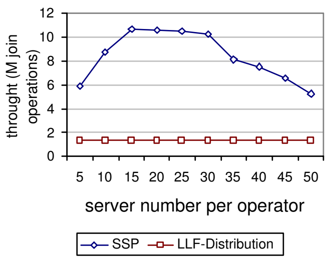

**Fig. 4.** Throughput results

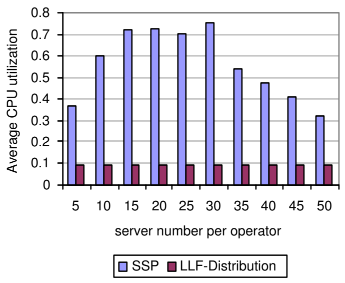

**Fig. 5.** CPU utilization results

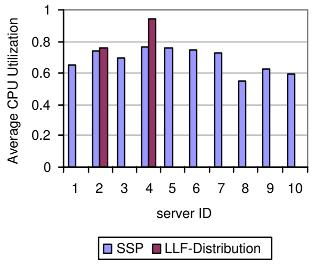

**Fig. 6.** Load balancing results

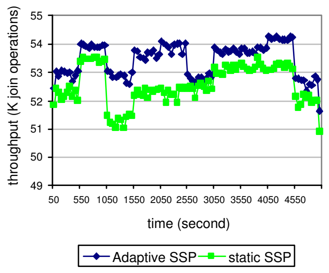

**Fig. 7.** Adaptation results

cluster. Figure 6 shows the average effective CPU utilization of the ten server nodes after the 5000 second simulation period. We observe that the SSP algorithm can achieve more balanced load distribution than the LLF-Distribution algorithm that can only perform load balancing at inter-operator level not at the intra-operator level. Our last experiment evaluates the adaptation strategy. The experiment runs ten stream join operators on a 100-node cluster. In the SSP algorithm, each join is allowed to use 5 servers. The initial average stream rates $r_1$ and $r_2$ are randomly selected from the range of [5,20] tuples/second. We then dynamically change the average stream rates every 500 seconds. The throughput value is sampled every 50 seconds. The throughput value at time $t$ records the total number of join operations performed by the whole server cluster between time $[t-50, t]$. Figure 7 shows the adaptation results of the SSP algorithm that dynamically switches the master stream and the slave stream based on the rate changes. We observe that the adaptive SSP consistently achieves better performance than the static SSP algorithm that always uses the same master and slave streams.

## 5 Conclusion

In this paper, we presented a novel load diffusion middleware proxy to enable distributed execution of resource-intensive stream joins using a cluster of servers. To the best of our knowledge, this is the first work that studied fine-grained load management problem for sliding-window stream joins. We proposed a simple correlation-aware stream partition algorithm that is proved to preserve the stream join accuracy while spreading the workload among distributed servers. Our experimental results show that the load diffusion scheme can greatly improve the system throughput and achieve balanced load distribution. For future work, we will develop more efficient and sophisticated stream partition algorithms and extend the system to support multi-way stream joins.

## References

1. C. Amza, A. Cox, W. Zwaenepoel. Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites, Proc. of the ACM/IFIP/Usenix Middleware Conference, June, 2003.
2. M. Balazinska, H. Balakrishnan, M. Stonebraker: Contract-based Load Management in Federated Distributed Systems, Proc. of 1st Symposium on Networked Systems Design and Implementation (NSDI), March, 2004.
3. G. Cybenko: Dynamic load balancing for distributed memory multiprocessors. Journal of Parallel and Distributed Computing, 7(2):279-301, 1989.
4. X. Gu, P. S. Yu, K. Nahrstedt, Optimal Component Composition for Scalable Stream Processing, Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS), June, 2005.
5. S. Krishnamurthy et al. TelegraphCQ: An Architectural Status Report. IEEE Data Engineering Bulletin, 26(1):11-18, March, 2003.
6. Arvind Krishna, Douglas C. Schmidt, and Raymond Klefstad, Enhancing Real-Time CORBA via Real-Time Java, Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS), May 23-26, 2004.
7. U. Srivastava, J. Widom: Memory Limited Execution of Windowed Stream Joins, Proc. of the 30th International Conference on Very Large Databases (VLDB), August, 2004.
8. N. Tatbul and U. etintemel and S. Zdonik and M. Cherniack and M. Stonebraker: Load Shedding in a Data Stream Manager, Proc. of the 29th International Conference on Very Large Data Bases (VLDB), September, 2003.
9. Y. Xing, S. B. Zdonik, J.-H. Hwang, Dynamic Load Distribution in the Borealis Stream Processor, Proc. of International Conference on Data Engineering (ICDE), April, 2005.
10. M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin, Flux: An Adaptive Partitioning Operator for Continuous Query Systems, Proc. of the 19th International Conference on Data Engineering (ICDE), March, 2003.
11. The STREAM Group. STREAM: The Stanford Stream Data Manager. IEEE Data Engineering Bulletin, 26(1):19-26, March 2003.
12. S. Zdonik et al. The Aurora and Medusa Projects. IEEE Data Engineering Bulletin, 26(1), March 2003.