

# Window-Oblivious Join: A Data-Driven Memory Management Scheme for Stream Join

Ji Wu            Kian-Lee Tan            Yongluan Zhou  
National University of Singapore  
{wuji, tankl, zhouyong}@comp.nus.edu.sg

## Abstract

Memory management is a critical issue in stream processing involving stateful operators such as join. Traditionally, the memory requirement for a stream join is **query-driven**: a query has to explicitly define a window for each (potentially unbounded) input. The window essentially bounds the size of the buffer allocated for that stream. However, outputs produced by such approach may not be desirable (if the window size is not part of the intended query semantic) due to the volatile input characteristics. We discover that when streams are ordered or partially ordered, it is possible to use a **data-driven** memory management scheme for improved performance. In this work, we present a novel **data-driven** memory management scheme, called Window-Oblivious Join (WO-Join), which adaptively adjusts the state buffer size according to the input characteristics. Our performance study shows that, compared to traditional Window-Join (W-Join), WO-Join is more robust with respect to the dynamic inputs and therefore produces higher quality results with lower memory costs.

## 1. Introduction

The emerging stream applications (such as network intrusion detection, traffic monitoring, and online analysis of financial tickers) usually involve processing sheer volume of online data in a time responsive manner. As such, computations over streaming inputs are generally memory-intensive, especially for operators that need to maintain runtime states. For such operators (join, aggregation etc.), a query typically requires a window clause, which effectively instructs the executor the amount of buffer size needed by that query. We call this query-driven memory management. While such mechanism works well in many situations, there are scenarios where output quality can be severely impaired as the desired answers may be missing from the result set. For example, an input tuple may have already been purged from the memory before it completely joins with tuples from other streams due to the inflexible state buffer size

fixed by the query window<sup>1</sup>. More results may be obtained if the state buffer size can adapt according to the input characteristics. To make this more concrete, consider a location tracking application in a wireless sensor network environment. The location of an object (a transmitter) can be inferred by synthesizing the Signal Strength (SS) measured at the surrounding sensors. In such applications, each sensor produces a series of data tuples with uniform schema (*epoch, x, y, z, val*), (where *epoch* refers to the time when the signal is recorded, *x, y* and *z* correspond to the physical coordinates of the sensor, and *val* is the SS measured). A typical query for tracking an object looks like the following:

```
SELECT * FROM Sensor1 S1, Sensor2 S2,  
Sensor3 S3, Sensor4 S4  
WHERE S1.epoch = S2.epoch  
AND S2.epoch = S3.epoch  
AND S3.epoch = S4.epoch
```

In this query, data packets from four sensors are routed to the central location to be joined together before the target location can be predicted. Owing to the unreliable communication channel, which results in a highly dynamic network topology as well as the availability of multiple paths from the source to the centralized location, tuples may experience different transmission delays and therefore arrive at the central location in arbitrary order (i.e., tuples are not ordered according to their epoch values). Now, as the traditional Window-Join (W-Join) [8] only joins tuples that are within a pre-defined window boundary, it implicitly assumes that all latency and out-of-order effects are absorbed by the window specified by the user. However, this may not hold since users typically have no clue about the underlying input characteristics or the network topology. As such, query accuracy may drop significantly when packets encounter severe transmission delay or a high degree of order scrambling. The only way to obtain consistent quality results is to define

<sup>1</sup>We recognize that there are applications whereby the specified window is an important part of the query semantics, i.e. the user does not intend to obtain the entire set of the join results, but only certain fraction of them. For example, the user may be only interested in the results generated from the tuples received in the recent 5 minutes. In this paper, we do not focus on this type of query.

sufficiently large query windows, which inevitably leads to extravagant memory overheads that many systems may not afford. The dilemma of choosing the appropriate window size shows that the W-Join approach is too rigid and therefore not suitable for such applications.

To address the issue, we contend that, whenever possible, memory allocation for stream join should be data-driven instead of query-driven. We therefore propose a new memory management scheme, called *Window-Oblivious Join* (WO-Join), which dynamically determines the state buffer size according to the current data input. WO-Join characterizes a query’s memory requirements as a function of two types of delays: namely *intra-stream delay* and *inter-stream delay*. When these two delays are bounded, complete join results are computable using finite memory space. WO-Join guarantees complete join results when these two parameters are known apriori. If such information is not available beforehand, WO-Join can monitor the two parameters during runtime and allocate the buffer size accordingly to ensure high quality results, even under memory-constrained scenario. Experiment study has demonstrated that WO-Join significantly outperforms W-Join in terms of both output quality and memory-efficiency in many situations.

The rest of this paper is organized as follows. In Section 2, we formulate the problem and identify factors that impact memory consumptions for stream join. Section 3 presents the cost model for WO-Join queries. We extend our discussions to processing queries under memory-constrained scenarios in section 4. Section 5 reports the experiment results. Finally, we discuss the related work and conclude the paper in section 6 and section 7.

## 2. Preliminaries

### 2.1. Problem Statement

We consider WO-Join over a set of infinite streams  $S$  with equality join predicate. The WO-Join may include one or more MJoin [16] operators. Within an operator, one buffer is maintained for each input stream. The buffer serves as a sliding window for the stream so that input data are inserted and removed in a FIFO manner. The size of each buffer is adjusted dynamically for tuples to be joined in a memory-efficient way. In this paper, we focus on queries where all participating streams share the same join attribute, which is already a sufficiently complicated problem. Our technique can be extended to more complex scenarios (such as join predicates involving more than one attributes from each input) and we shall study them in our future work.

To summarize, the problem studied in this paper is as follows: Given a multi-way equijoin query without explicit window semantic, produce high quality results in a

$S$	The whole set of input streams
$S_i$	$i$ th input stream
$s_i$	A tuple in stream $S_i$
$s_i.A$	Join key where stream ordering is defined
$k_i$	Scrambling factor of stream $S_i$
$r_i$	Data rate of stream $S_i$
$PI(s_i)$	Physical index of tuple $s_i$
$VI(s_i)$	Virtual index of tuple $s_i$
$L_{S_i \leftrightarrow S_j}(t)$	Lag from $S_i$ to $S_j$ at time $t$
$MS_{S_i \leftrightarrow S_j}(t)$	Memory space to buffer $S_i$ tuples w.r.t. stream $S_j$ at time $t$
$Mul(S_i)$	Multiplicity of stream $S_i$

Table 1: Important notations used in the paper

memory-efficient manner. The quality of the results is measured by the output accuracy defined as follows:

$$\text{output accuracy} = \frac{\# \text{ of join tuples actually produced}}{\# \text{ of join tuples that can be produced}} \quad (1)$$

Note that when the streams are completely unordered, there is no way to produce high quality join results since the memory evaluation costs for that are unbounded. Here, we focus on applications where join results are bounded-memory computable. Back to our previous sensor network example, although data packets may experience time-varying delays, such delays are bounded as we know sooner or later packets will be delivered to the destination. Since the delays are bounded, tuple ordering on the join key (“epoch”) should be *in the long run* monotonically increasing. This opens the possibility to produce complete (or near-complete) results with limited memory space. In fact, the core issue we address is to relate the memory evaluation costs with the stream characteristics. As can be seen later, this issue is not as straightforward as it may look. We start the discussion by introducing two main sources of memory overheads for stream join, namely the *intra-stream delay* and the *inter-stream delay*.

### 2.2. Intra-stream Delay

Intra-stream delay causes tuples’ order to be scrambled within a stream (the tuple ordering issue). To facilitate our study, we first define what is a totally ordered stream, then quantify a partially ordered stream by a parameter called *Scrambling Factor* (SF). For ease of exposition, assume each tuple in the stream has an index, called *Physical Index* (or PI), which corresponds to the arrival position of that tuple in the stream. For example, the PI of the first arrived tuple from a stream is 1. The PI of the next arrived tuple is 2 and so on. A tuple  $s_i$ ’s PI value is given by the function  $PI(s_i)$ . Important notations used throughout this paper are listed in Table 1.

A totally ordered stream is defined as follows:

**Definition 2.1** A totally ordered stream  $S_i$  must fulfill the following condition for any pair of tuples  $s_i$  and  $s'_i$  from  $S_i$ :

$$\text{If } PI(s_i) < PI(s'_i), \text{ then } s_i.A < s'_i.A$$

A partially ordered stream with scrambling factor  $k$  is defined as follows:

**Definition 2.2** A partially ordered stream  $S_i$ , with **Scrambling Factor**  $k$ , must fulfill the following condition for any pair of tuples  $s_i$  and  $s'_i$  from  $S_i$ :

$$\text{If } s_i.A \leq s'_i.A, \text{ then } PI(s_i) - k \leq PI(s'_i)$$

where  $k$  is the minimum integer that satisfies the inequality.

Notably, our notion of totally ordered stream defines a strictly ordered sequence, i.e. no duplicates are allowed. If tuples with the same attribute value exist in the stream, such stream is only considered partially ordered even though it is in *non-descending* order<sup>2</sup>. The reason is that, as we shall see later, from memory management point of view tuples with duplicate values do affect memory requirements as if they are unordered. Therefore, we do not distinguish between tuples that are out of order and tuples with duplicate values in our definition.

Clearly, according to Definition 2.2, a lower  $k$  implies a stricter ordered sequence while a higher  $k$  implies a more scrambled sequence. For the rest of the paper, we use the value of SF (or  $k$ ) to measure the degree of out-of-order for a given partially ordered stream.

### 2.3. Inter-stream Delay

Inter-stream delay occurs when streams are not synchronized. Roughly speaking, the inter-stream delay is defined as the arrival time difference between the matching tuples from different inputs. Intuitively, such delay directly impacts the memory consumption: Longer inter-stream delay implies larger memory overheads. However, to judge whether streams are synchronized or to quantify the delay between unsynchronized streams is not trivial, especially when input streams are not totally ordered. In this section, we first discuss how to quantify the inter-stream delay between totally ordered streams, then extend the concept to partially ordered streams.

To ease the presentation, we make the following definition, which will be used throughout the paper.

**Definition 2.3** Given a tuple  $s_i \in S_i$ , a tuple  $s_j \in S_j$  whose join key is equal to  $\max\{s'_j | s'_j.A \leq s_i.A, s'_j \in S_j\}$  is called  $s_i$ 's **next-of-kin tuple** from  $S_j$ .

<sup>2</sup>Without loss of generality, we only consider ascending or non-descending order. A stream with descending or non-ascending order can be handled similarly.

**2.3.1. Totally Ordered Streams.** To begin with, let us consider delays between totally ordered streams. From the memory requirement perspective, we use *Lag* to quantify such delay. *Lag* measures the number of tuples from one stream that the system has to buffer as they may potentially join with tuples that have not arrived from the other stream.

**Definition 2.4** Let  $s_i$  and  $s_j$  be the latest arrived tuples from totally ordered streams  $S_i$  and  $S_j$  at time  $\tau$ , then the lag from  $S_i$  to  $S_j$ , denoted by  $L_{S_i \leftrightarrow S_j}(\tau)$ , is quantified as:

$$L_{S_i \leftrightarrow S_j}(\tau) = PI(s_i) - PI(s'_i) \quad (2)$$

where  $s'_i$  is  $s_j$ 's next-of-kin tuple from  $S_i$ .

The intuition is that the arrival of  $s_j$  can evict tuples in the  $S_i$  buffer whose join keys are less than  $s_j$ 's next-of-kin tuple. So only  $PI(s_i) - PI(s'_i)$  number of tuples need to be retained in the  $S_i$  buffer (remember there are no tuples with duplicate values here since streams are totally ordered). Note that  $L_{S_i \leftrightarrow S_j}(\tau)$  can be negative. This occurs when  $s'_i$  comes after  $s_i$  in  $S_i$ .

Another way of understanding this equation is we can treat  $s'_i$  as  $s_j$ 's "correspondence tuple" in stream  $S_i$ . Therefore the *Lag* between the streams can be measured by the distance between  $s_i$  and  $s'_i$ .

$S_i$  is said to be *synchronized* with  $S_j$  at time  $\tau$  if

$$0 \leq L_{S_i \leftrightarrow S_j}(\tau) \leq 1 \quad (3)$$

Furthermore, if  $L_{S_i \leftrightarrow S_j}(\tau) > 1$ , we say that, at time  $\tau$ ,  $S_i$  runs ahead of  $S_j$  or  $S_i$  leads  $S_j$ .  $S_i$  is called the "leading" stream and  $S_j$  is called the "lagging" stream.

It is evident that, if two totally ordered streams are always synchronized between each other, the buffer space required for both streams are nominal. The arrival of a new tuple from one stream can immediately evict the last arrived tuple from the other stream.

**2.3.2. Partially Ordered Streams.** Comparatively, synchronization between partially ordered streams is much harder to define. This is because when the next-of-kin tuples arrive at different time, it is difficult to tell whether the time difference is due to delay between the streams or due to scrambled tuple order within the stream. To better understand the synchronization issue in this case, we first introduce the notion of *Virtual Index* (VI):

**Definition 2.5** The virtual index of a tuple  $s_i \in S_i$ , denoted by  $VI(s_i)$ , is the set of  $PI(s_i)$  when stream  $S_i$  is sorted in non-descending order.

It is important to note that, different from  $PI(s_i)$  which returns a unique index value,  $VI(s_i)$  returns a set of consecutive indices. This is because attributes with duplicate

values take up multiple positions in the sequence. It is valid to map a tuple’s virtual index to any one of these positions. We denote  $|VI(s_i)|$  to be the size or cardinality of the set  $VI(s_i)$ . And let  $VI_{min}(s_i)$  and  $VI_{max}(s_i)$  be the minimum and maximum value from the set  $VI(s_i)$ . For a stream  $S_i$  whose join key is duplicate-free,  $VI_{min}(s_i) = VI_{max}(s_i)$  and  $|VI(s_i)| = 1$ . An important relationship between a tuple’s  $VI$  and its  $PI$  is summarized below:

**Theorem 2.1** For a partially ordered stream  $S_i$  with  $SF = k$ , we have the followings:

$$PI(s_i) - VI_{min}(s_i) \leq k \quad (4a)$$

and

$$VI_{max}(s_i) - PI(s_i) \leq k \quad (4b)$$

Due to space constraint, proof for the above theorem is omitted. Interested readers are referred to the extended version of this paper [17] for more details.

Now we are ready to derive the function to compute  $L_{S_i \leftrightarrow S_j}(\tau)$ . Here, the *Lag* measures the number of buffered tuples from  $S_i$  caused solely due to the delay between the streams, while the memory overheads owing to intra-stream delay are excluded.

**Definition 2.6** Let  $s_i$  and  $s_j$  be the latest arrived tuples from  $S_i$  and  $S_j$  respectively at time  $\tau$ , and  $s'_j$  be a tuple from  $S_j$  such that  $PI(s_j) \in VI(s'_j)$  and  $s'_i$  be  $s'_j$ ’s next-of-kin tuple from  $S_i$ . Then,

$$L_{S_i \leftrightarrow S_j}(\tau) = PI(s_i) - VI_{max}(s'_i) \quad (5)$$

The definition of synchronization for partially ordered streams is the same as that of totally ordered streams mentioned earlier (Equation 3).

The intuition here is we first find out the tuple  $s'_j$  that would have appeared at  $s_j$ ’s position if the stream is ordered, and then we “map”  $s'_j$  to its “correspondence tuple” in  $S_i$  as done in the discussion for totally ordered streams. As an example, Figure 1 shows a snapshot of two data streams arriving at the system. The timeline on the left indicates the tuple arrival time. The values inside  $\langle \rangle$ ,  $[\ ]$  and  $\{ \}$  correspond to the join key, PI and VI of the tuple respectively. For example, at  $t = t_5$ , tuple  $\langle 66 \rangle \in S_1$  arrives, with  $PI(\langle 66 \rangle) = p + 2$  and  $VI(\langle 66 \rangle) = p + 4$ , where  $p$  is the PI of the first arrived  $S_1$  tuple shown in the figure. Now consider at  $t = t_5$ , the last tuple arrived from  $S_1$  and  $S_2$  are  $s_1 = \langle 66 \rangle$  and  $s_2 = \langle 65 \rangle$  respectively. And the tuple  $s'_2 = \langle 63 \rangle$  is the one that satisfies the condition  $PI(s_2) \in VI(s'_2)$ . Then we find out  $s'_2$ ’s next-of-kin tuple from  $S_1$  to be  $s'_1 = \langle 63 \rangle$  (either the tuple arrived at  $t_1$  or at  $t_4$ ). Since  $PI(s_1) = p + 2$  and  $VI_{max}(s'_1) = p + 2$ , we get  $L_{S_1 \leftrightarrow S_2}(t_5) = 0$  (Definition 2.6). Hence the two streams

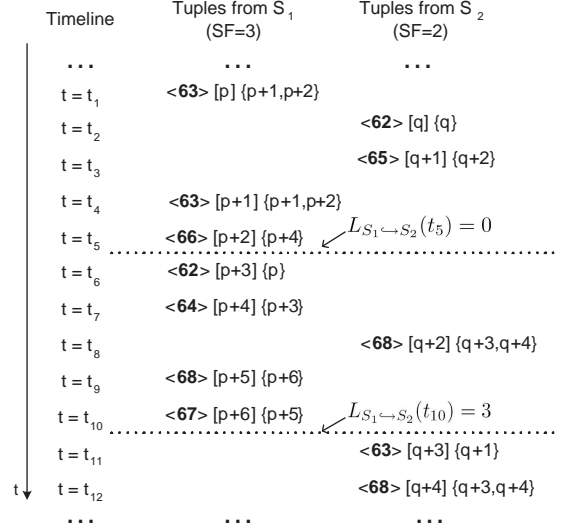


Figure 1: Example of synchronized streams

are synchronized at time  $t = t_5$ . Analogously, we can derive that  $L_{S_1 \leftrightarrow S_2}(t_{10}) = 3$ , with  $s_1 = \langle 67 \rangle$ ,  $s_2 = \langle 68 \rangle$ ,  $s'_2 = \langle 65 \rangle$  and  $s'_1 = \langle 64 \rangle$ .

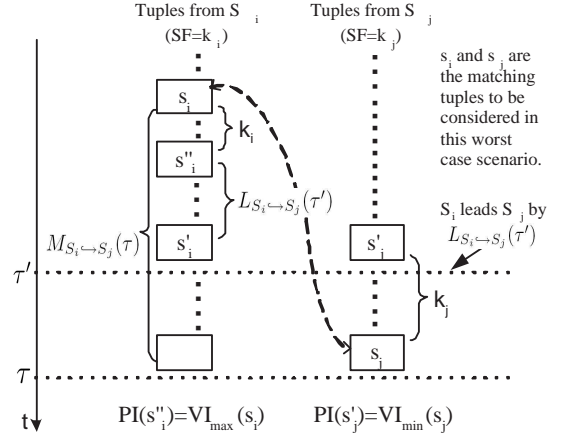
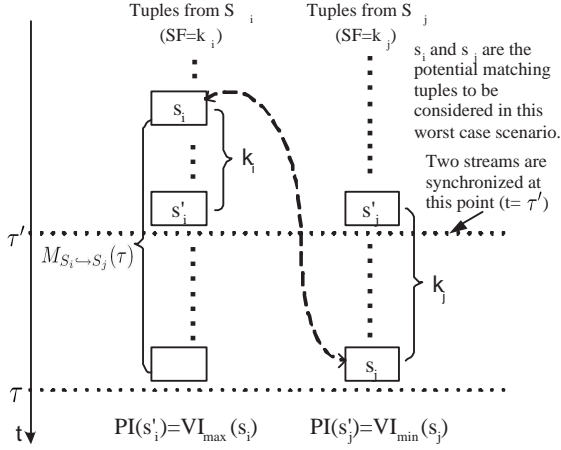
### 3. Memory Cost Model

An accurate memory cost model is crucial for WO-Join. It is not only important during query processing to ensure proper memory allocation, but also useful for other aspects such as performing the disk-buffer scheduling to handle severely unsynchronized streams, dynamically distributing memory among different queries, or implementing admission control to avoid memory congestion. As an example, we will show how to use the cost model for disk-buffer scheduling in Section 4. In this section, we start by analyzing the model for a single join operator, then extend it to queries involving multiple join operators.

#### 3.1. Cost Model Derivation

The emergence of MJoin [16] allows multiple inputs to be joined in one step. Compared to traditional binary join, MJoin leverages the memory efficiency by eliminating the intermediate join results. So we choose to base our memory cost model on MJoin. Binary join can be viewed as a special case of MJoin with only two inputs.

As mentioned before, the memory cost model characterizes the memory requirements based on two factors: scrambled tuple ordering ( $SF$ ) and delay among streams ( $Lag$ ). Given these two parameters, the model should provide the memory evaluation costs for generating the complete join results in the worst case scenario. In what follows, we con-



(a)  $S_i$  leads  $S_j$

Figure 2: Buffer requirements for  $S_i$  to join with synchronized streams (the worst case scenario)

consider the impact of the tuple ordering alone by first assuming streams are all synchronized. Then we extend the cost model to the unsynchronized stream scenario.

### 3.1.1. Joining Synchronized Streams.

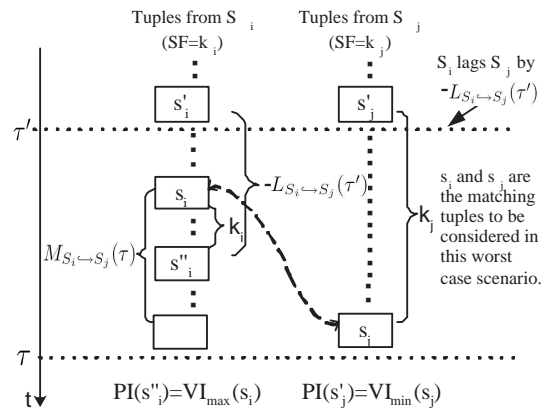
Given two streams  $S_i$  and  $S_j$ , let  $s_j$  be the latest tuple from  $S_j$  at time  $\tau$ , and  $s'_j$  be a tuple such that  $PI(s'_j) \in VI(s_j)$ . Suppose the two streams are synchronized at time  $\tau'$  when tuple  $s'_j$  arrived at the system, then the number of tuples from  $S_i$  that should be kept in the buffer at time  $\tau$ , denoted by  $M_{S_i \leftrightarrow S_j}(\tau)$ , in order to guarantee the complete join results with  $S_j$  is given as follows:

$$M_{S_i \leftrightarrow S_j}(\tau) = k_i + r_i \cdot \frac{k_j}{r_j} \quad (6)$$

We illustrate the intuition by the example given in Figure 2. The two streams  $S_i$  and  $S_j$  are synchronized at time  $\tau'$  when  $s'_j$  arrives. Let  $s'_i$  denote the latest tuple from  $S_i$  at time  $\tau'$  and  $s_i$  be  $s'_j$ 's next-of-kin tuple from  $S_i$ . We can guarantee that tuples arrived  $k_i$  tuples earlier than  $s'_i$  have join key values less than  $s_i.A$ . Therefore, we only need to buffer the last  $k_i$  tuples from  $S_i$  that arrived earlier than  $\tau'$ . Furthermore, we also have to buffer tuples from  $S_i$  that arrive during the interval  $[\tau', \tau]$ . The upper bound of this buffer is  $r_i \cdot \frac{k_j}{r_j}$  as the number of tuples that arrive from  $S_j$  during this period is at most  $k_j$  (Theorem 2.1).

### 3.1.2. Joining Unsynchronized Streams.

Now consider the memory overheads to join unsynchronized streams. For two streams  $S_i$  and  $S_j$ , let the latest arrived tuple from  $S_j$  at time  $\tau$  be  $s_j$ , and  $s'_j$  be a tuple such that  $PI(s'_j) \in VI(s_j)$ . Suppose the Lag from  $S_i$  to  $S_j$  at time  $\tau'$  when tuple  $s'_j$  arrives at the system is  $L_{S_i \leftrightarrow S_j}(\tau')$ , then the number of tuples from  $S_i$  that should be kept in the buffer at time  $\tau$ , denoted



(b)  $S_i$  lags  $S_j$

Figure 3: Buffer requirements for  $S_i$  to join with unsynchronized stream (the worst case scenarios)

by  $M_{S_i \leftrightarrow S_j}(\tau)$ , is:

$$M_{S_i \leftrightarrow S_j}(\tau) = \max\{1, k_i + r_i \cdot \frac{k_j}{r_j} + L_{S_i \leftrightarrow S_j}(\tau')\} \quad (7)$$

The intuition of the above equation is illustrated in Figure 3(a) and Figure 3(b). Figure 3(a) shows the scenario where  $S_i$  leads  $S_j$  by  $L_{S_i \leftrightarrow S_j}(\tau')$  at time  $\tau'$ . Imagine that if we "remove" the last  $L_{S_i \leftrightarrow S_j}(\tau')$  tuples from  $S_i$  arrived before  $\tau'$ , the situation is identical to the synchronized stream scenario illustrated in section 3.1.1: The amount of  $S_i$  tuples that should be retained in the buffer is  $k_i + r_i \cdot \frac{k_j}{r_j}$ . However, since now we need to additionally buffer  $L_{S_i \leftrightarrow S_j}(\tau')$  tuples due to the stream lag, the actual buffer size becomes  $k_i + r_i \cdot \frac{k_j}{r_j} + L_{S_i \leftrightarrow S_j}(\tau')$ .

Next let us consider the scenario where  $S_i$  lags  $S_j$  shown in Figure 3(b). Same as before, the number of tuples from  $S_i$  that should be buffered for  $S_j$  is  $k_i + r_i \cdot \frac{k_j}{r_j} + L_{S_i \leftrightarrow S_j}(\tau')$ . Interestingly, since now  $L_{S_i \leftrightarrow S_j}(\tau') < 0$ , this implies the

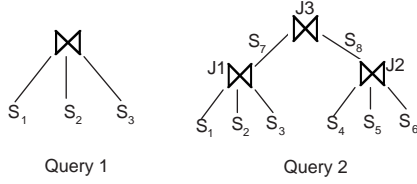


Figure 4: Example queries used in this paper

lag between  $S_i$  and  $S_j$  helps reduce the memory overheads incurred at  $S_i$ . Actually,  $|L_{S_i \leftrightarrow S_j}(\tau')|$  can be so large that the entire equation turns negative. Physically, it means stream  $S_i$  runs far behind  $S_j$  such that tuples from  $S_i$  can immediately join with all matching tuples from  $S_j$  (if such matching tuples exist) when they arrive. In this case, the size of buffer needed at  $S_i$  (for matching tuples from  $S_j$ ) is minimal. So we set it to 1.

The discussions so far focus on binary join. However, it is straightforward to extend it to multi-way join scenario. For example, consider the memory requirements to buffer  $S_i$  in a three way join  $S_i \bowtie S_j \bowtie S_k$ . We can calculate the memory requirements between  $S_i$  and each of the other input streams (i.e.  $M_{S_i \leftrightarrow S_j}(\tau)$  and  $M_{S_i \leftrightarrow S_k}(\tau)$ ). The maximum value will be the actual buffer required for  $S_i$ . Formally, the memory space needed for  $S_i$  in an MJoin operator with  $n$  input streams is:

$$M_{S_i}(\tau) = \max_{\forall j \in n, j \neq i} M_{S_i \leftrightarrow S_j}(\tau) \quad (8)$$

### 3.2. Issue at Query Level

In this section, we consider the memory requirements of a query plan tree that consists of multiple MJoin operators. In fact the central issue here is that we need to estimate the data characteristics of the intermediate join results so that we can proceed to predict the memory consumption for the downstream operators. Take Query 2 of Figure 4 as an example. In order to estimate the memory consumption for operator J3, the system needs to know the data characteristics of stream  $S_7$  and  $S_8$ . Particularly, we are interested in the upper bound of their SF and data rates. We show that these values can be derived from the inputs of the upstream MJoin operators (i.e. J1 and J2 in this example). The following theorem gives the equation to estimate the characteristics of output stream based on the characteristics of the input streams.

**Theorem 3.1** *For an MJoin operator with  $n$  input streams, the data rate and SF of its output stream, denoted by  $r_o$  and  $k_o$ , respectively, are bounded as follows:*

$$r_o \leq \sum_{i=1}^n r_i \prod_{\substack{j=1 \\ j \neq i}}^n Mul(S_j) \quad (9)$$

$$k_o \leq \sum_{i=1}^n k_i \prod_{\substack{j=1 \\ j \neq i}}^n Mul(S_j) \quad (10)$$

where  $k_i$  is the SF of stream  $S_i$  and  $Mul(S_j)$  is the multiplicity of stream  $S_j$ .

Proof of the above theorem can be found in [17]. Here, a stream's *Multiplicity* refers to the maximum size of the tuples with duplicate values. If a stream's join attribute value is unique, the *Multiplicity* is 1. It is important to note the stream's SF is always greater than or at least equal to its *Multiplicity*, since tuples with duplicate values affect stream's SF according to Definition 2.2.

Compared to SF or data rate, inter-stream delays among the intermediate result streams are much easier to quantify since they are simply determined by the most "lagging" input of each upstream operator. For example, if in Query 2,  $S_1$  runs behind both  $S_2$  and  $S_3$ ;  $S_4$  runs behind both  $S_5$  and  $S_6$ . Then the *Lag* between  $S_7$  and  $S_8$  is in fact only determined by the *Lag* between  $S_1$  and  $S_4$  (if the processing speed of the operator J1 and J2 are the same).

## 4. Memory-Constrained WO-Join

With the proposed cost model, a query processor is able to accurately predict the memory requirements for each join operation based on the input characteristics and dynamically allocate the memory space accordingly. However, because stream inputs are highly volatile, the required buffer size may vary drastically over time. For example, a data burst of one stream can cause a huge amount of delay between that stream and other input streams, leading to extravagant memory overheads in a short term. When this occurs, the required buffer space may easily exceed the physical memory available. Motivated by the observations that memory overheads mainly attribute to two factors: scrambled tuple ordering and lag among streams, we propose two techniques (plus one hybrid approach) to prevent the system from memory overrun while still maintaining high quality results. Each technique targets one of the contributing factors to minimize its impact on memory overhead.

### 4.1. Memory-Sort First Strategy

The Memory-Sort First (MSF) strategy aims to tackle the problem of scrambled tuple ordering. The idea is straightforward: the system performs an in-memory sort on all input streams first before the join. After the sorting, the SF value is reduced to the stream's multiplicity, which is the minimal possible SF value for a stream with duplicate join keys. Hence, for an MJoin operator, the space needed to buffer tuples from  $S_i$  is reduced from

$$k_i + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j / r_j + L_{S_i \leftrightarrow S_j}(t)\}$$

$$\text{to } Mul(S_i) + \max_{\forall j \in n, j \neq i} \{r_i \cdot Mul(S_j)/r_j + L_{S_i \leftrightarrow S_j}(t)\}$$

However, sorting input tuples costs extra buffer as well: an in-memory sort on a stream with  $SF = k$  requires extra space to buffer the  $k$  most recent tuples. As we shall see later in the experiments, such memory cost may be substantial especially when the query plan only involves a single join operator. Nevertheless, the benefit of MSF becomes apparent when the query plan consists of pipelined join operations. Another good side-effect is that output produced using MSF is also ordered. This may be crucial for applications that are sensitive to the data order. Although MSF reduces memory overheads, it introduces significant latency in producing the results because an input tuple has to wait until  $k$  number of its succeeding tuples have arrived before participating in the join operation.

## 4.2. Disk-Buffer First Strategy

The Disk-Buffer First (DBF) strategy deals with streams that are not synchronized. The intuition is when a stream  $S_i$  runs *far* ahead of stream  $S_j$ , recently arrived  $S_i$  tuples will not immediately contribute to any join results since the matching tuples from  $S_j$  have not arrived yet. To save memory, new tuples from  $S_i$  can be flushed onto the disk first and then retrieved later when their join counterparts from  $S_j$  are about to arrive. Essentially, the DBF strategy reduces the memory overheads by minimizing the *Lag* among input streams. The strategy works as follows: Every time a new tuple from the “lagging” stream (say  $S_j$ ) arrives, it triggers the query engine to read tuples of the “leading” stream (say  $S_i$ ) from the disk such that both  $L_{S_i \leftrightarrow S_j}(t)$  and  $L_{S_j \leftrightarrow S_i}(t)$  are 0 or close to 0. In actual implementation, the system reads slightly more tuples of the “leading” stream from the disk in advance such that  $L_{S_i \leftrightarrow S_j}(t) > C$ , where  $C$  is some predefined runtime parameter. The purpose of doing this is to minimize the possibility that the join to be blocked due to the I/O operations on  $S_i$  when  $S_j$ ’s data rate is higher than the I/O speed. It is important to note that DBF is only beneficial when inter-stream delays are significant. If the *Lag* is so small that the arrival time difference among matching tuples is less than the time for 1 disk read plus 1 disk write, then the “leading” stream will eventually run behind the “lagging” stream after the I/O operation. In our implementation, DBF strategy is only triggered when the *Lag* is no less than 3 times of the disk I/O time. According to the cost model, with the DBF strategy, the memory needed to buffer a stream  $S_i$  can be decreased from

$$k_i + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j/r_j + L_{S_i \leftrightarrow S_j}(t)\}$$

$$\text{to } k_i + C + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j/r_j\}$$

However, similar to MSF, DBF also introduces extra output latency because of the disk I/O operations.

## 4.3. Hybrid Approach

The MSF and DBF strategies are two complementary approaches. They can be combined to achieve even better memory reduction. The hybrid approach first sorts the recently arrived tuples in the main memory, then flushes the “leading” stream tuples onto the disk. According to the cost model, the buffer space required by using the hybrid strategy can be decreased from

$$k_i + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j/r_j + L_{S_i \leftrightarrow S_j}(t)\}$$

$$\text{to } Mul(S_i) + C + \max_{\forall j \in n, j \neq i} \{r_i \cdot Mul(S_j)/r_j\}$$

But the output latency using Hybrid also suffers the most. This approach is suitable for the situation where memory space is extremely limited while a relatively larger output latency is tolerable.

## 5. Experimental Study

We developed a WO-Join prototype system using Sun JDK 5.0. The system consists of two components: the query executor, and the memory manager. The query executor is similar to the main memory version of the MJoin operator [16], except that the buffer size for each stream is dynamically determined by the memory manager. The memory manager updates the stream characteristics such as data rate,  $SF$  and *Lag* upon the arrival of a tuple (which could be a batch of individual tuples in a batch processing mode) if these parameters are not known apriori, and determines the buffer size for the input streams.

We ran the experiments for both Query 1 and Query 2 as shown in Figure 4. Query 1 is used to validate our cost model and the memory efficiency of WO-Join, while Query 2 is for evaluating the WO-Join performance for pipelined joins. A data generator was implemented to produce streams with customizable  $SF$ , multiplicity, *Lag* and data bursts. By default, streams are generated according to Poisson process with mean inter-arrival time equal to 20ms. The  $SF$  of each stream ranges from 0 to 500 and the multiplicity is set to 2 unless otherwise specified. The size of each tuple is 24 bytes. The experiments were conducted on an IBM x255 server running Linux with four Intel Xeon MP 3.00GHz/400MHz processors and 18G DDR main memory. All experiments were repeated thirty times and the average values were reported. We also varied the above parameters and found that our conclusions are not sensitive to their values. Due to the space limit, we omit these results.

As a side note, other than the query executor, the memory manager also introduces memory overheads to the system since estimating the stream characteristics, such as the  $SF$ , requires maintaining stream states. Here we adopt an

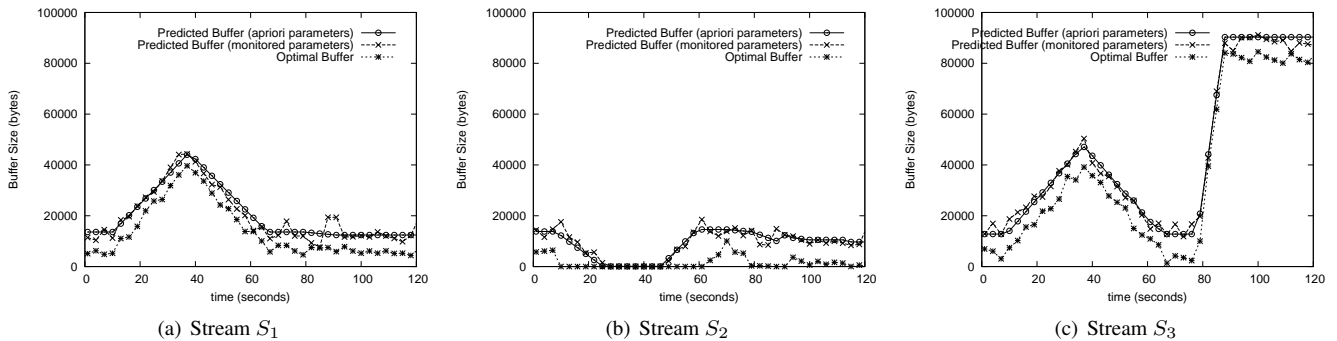


Figure 5: Buffer space for streams in Query 1.

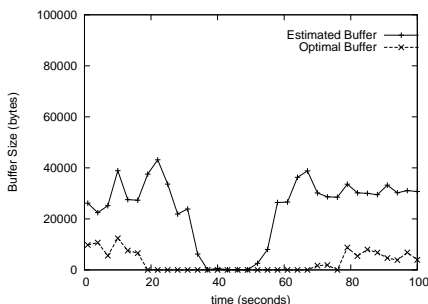


Figure 6: Buffer for  $S_7$  (Query 2)

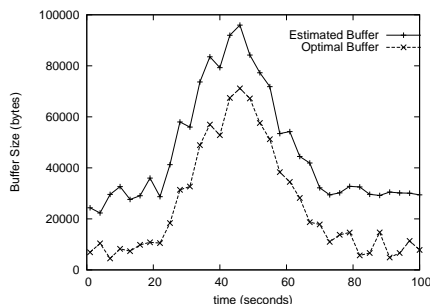


Figure 7: Buffer for  $S_8$  (Query 2)

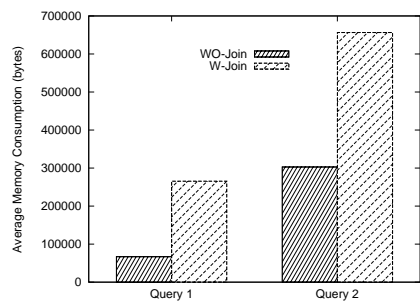


Figure 8: WO-Join vs W-Join

idea similar to *k-Mon* [4], which integrates the monitoring mechanism into the query executor to avoid duplicating stream states. Therefore, its overhead is almost negligible.

## 5.1. Experimental Evaluation

**Memory Cost Model Validation** The first experiment aims to validate our memory cost model. We start by running Query 1 and report the maximum buffer size for each input stream predicted by the cost model with and without apriori knowledge about the data characteristics. To compare, for each output tuple produced, we *backtrack* and find out the minimum buffer size required to produce that tuple, and report the maximal values recorded in each second. We call this value the optimal buffer space required to generate the complete join results. At  $t = 10s$ , we suppress  $S_2$ 's data rate to  $1/8$  of its initial value to emulate congestion at its source's side. The problem lasts for 30 seconds and then is restored. Similarly, to simulate data bursts, we increase  $S_3$ 's data rate by 8 times at  $t = 80s$  and hold it for 10 seconds before restoring its initial rate. Figure 5 depicts the buffer consumption for the three streams over a period of time. It is clear that the curves for the predicted buffer consumption closely follow the one for the optimal buffer consumption. As we can see, during the period when  $S_2$ 's source is congested, the buffer spaces for  $S_1$  and  $S_3$  in-

crease significantly. This is due to the sharp increase of the *Lag* from  $S_1$  to  $S_2$  and from  $S_3$  to  $S_2$ . For the same reason, when there is a data burst on  $S_3$  at  $t = 80$ , the buffer space for  $S_3$  increases substantially. In either case, the predicted buffer consumption (either based on apriori knowledge or based on parameters monitored during runtime) adapts well with respect to the input variations.

In terms of output quality, when the input parameters (data rate,  $SF$ , multiplicity and *Lag*) are known apriori, the system generates 100% join results. This validates our memory cost model which guarantees complete results given the accurate parameter settings. On the other hand, when the input characteristics are not known beforehand, they will be monitored during runtime by the memory manager for predicting the buffer size. In this case, we obtain an output accuracy of 99.6%. Minor answer tuples are missed when there is a drastic change in some of the input parameters which the memory manager does not catch up with instantly. Nevertheless, we can see from Figure 5 that the curve for the predicted buffer size based on monitored parameters almost coincides with the one based on apriori input knowledge except for some minor variations. For clarity, only the curve based on monitored parameters will be plotted for the subsequent experiments since it reflects a more realistic scenario. The output accuracy achieved based on monitored parameters is between 99.2% and 99.6% in all

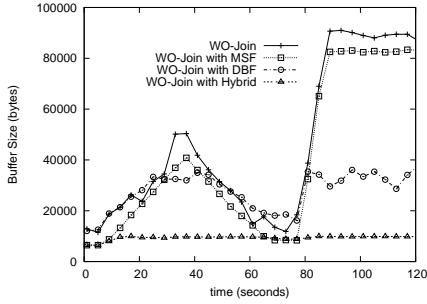


Figure 9: Memory reduction strategies

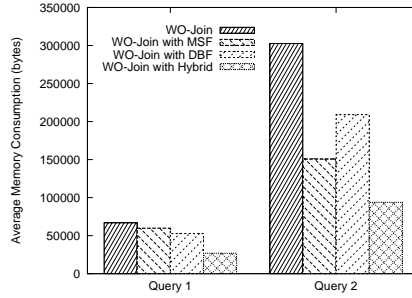


Figure 10: Avg memory consumption

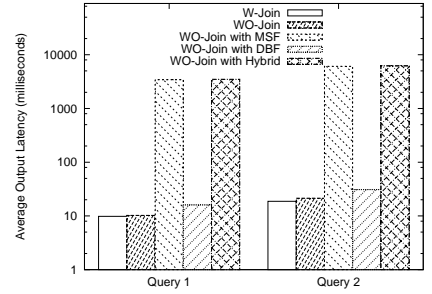


Figure 11: Output latencies

these experiments.

Next, we perform the similar experiment on Query 2. We would like to see whether the non-leaf operator ( $J_3$  in this case) adapts well using WO-Join. Similar to the above experiment, we suppress  $J_1$ 's output rate which causes  $S_7$  to run behind  $S_8$ . Changes in buffer consumption for  $S_7$  and  $S_8$  are shown in Figure 6 and Figure 7 respectively. However, this time the margin between the estimated value and the optimal value is much wider compared to the case for Query 1. This is because, for a non-leaf operator, the  $SF$  and data rate of the input streams are predicted based on the upper bounds of the statistics of the raw input streams (recall Theorem 3.1). However, this does not mean the system over-estimates the buffer consumption because the actual buffer required may still hit the predicted value in the worst-case scenario.

**Memory Overhead Comparison** Our next experiment compares the average memory consumption between WO-Join and traditional W-Join. The experiments on W-Join were conducted as follows: we first set the window size (or the buffer size) on par with the amount of buffer consumed by WO-Join. We get an output accuracy less than 40% for both Query 1 and Query 2. Then we slowly increase the window size until it is just large enough to produce the equivalent output quality as WO-Join (between 99.2% and 99.6% accuracy). The required memory consumptions are shown in Figure 8. As expected, since W-Join fixes the window size throughout the query execution, the required buffer size has to be much larger than WO-Join in order to achieve the same level of the output quality.

**Memory Constrained WO-Join** Now we evaluate the memory reduction strategies proposed in Section 4 using Query 1. Figure 9 compares the buffer consumption for  $S_3$  using these strategies (the observations of the buffer spaces for other streams are similar). As shown, the memory reduction achieved by MSF is insignificant. This is mainly because MSF requires additional buffer space for pre-sorting tuples, which almost offsets the savings brought by the ordered sequence. On the other hand, the DBF approach is more effective in this case. This can be attributed to DBF's

ability to “re-synchronize” the streams without much memory overhead. We can see from the figure the buffer size under DBF is always below a certain value regardless of the amount of  $Lag$  among streams, which is determined by other factors such as  $SF$  and data rate. It is not surprising that the Hybrid approach yields the best memory reduction. Since tuples are ordered and synchronized when they are joined, the buffer required at each stream is minimized. Actually, the major memory cost for the hybrid approach comes from the buffer for tuple pre-sorting required in MSF.

Figure 10 depicts the average memory consumptions of Query 1 and Query 2 under various strategies. Obviously, Hybrid achieves the lowest memory overheads. The interesting observation here is in Query 1, DBF consumes less memory than MSF, while in Query 2, the reverse is true. There are two reasons. First, in Query 2, tuple pre-sorting of MSF is only required at the leaf join operator. Hence, with pipelined join operators, the memory overhead for sorting input streams becomes less significant. Second, the DBF approach suffers from the rough estimation of the upper bound of  $SF$  and data rates for intermediate result streams. Therefore, the memory cost is still quite high even though the  $Lag$  among streams have been minimized.

**Output Latency** Lastly, Figure 11 compares the average output latencies among W-Join, the plain WO-Join and various WO-Join memory reduction strategies. The latency is measured by the difference between the output time of a result tuple and the arrival time of its last contributing input tuple. We can see that the latency of WO-Join is almost as low as the W-Join approach. This means the computation overhead incurred by WO-Join is negligible. The latency of DBF is about 50% higher than the plain WO-Join due to the I/O operations involved. The most stunning fact is that the latency of MSF and Hybrid is two orders of magnitude higher than that of WO-Join. The reason is, in MSF or Hybrid, tuples cannot be used to join with other streams upon arrival. Instead, they are stored in the sorting buffer first. The delay introduced by the sorting is much higher than the one caused by the join itself. Therefore, MSF and Hybrid suffer from severe output latencies in this case.

## 6. Related Work

While plethora of cost-efficient algorithms have been proposed for stream query, such as [3, 7, 15], works on memory-efficient processing are relatively fewer. Arasu et al. [2] classifies a broad range of queries into two classes: those can be evaluated with bounded memory and those cannot. Cammert et al. [5] proposes an adaptive memory management approach for W-Join. Also, various stream properties such as the ordered constraint [4] and *slack* [1] are identified in the literature. The idea is similar to our notion of *SF*. However, they are applied in the context of W-Join solely for the purpose of minimizing the memory overheads. We view these stream properties from a different angle: Since they (intra/inter-stream delay etc.) are the crucial factors that contribute to the memory consumption, we may build a memory cost model based on these factors so that stream join can be evaluated in a window-oblivious way and high quality results are attainable even in a memory-constrained scenario. Punctuation [13] or heartbeat [11] offers an alternative solution to join stream data without an explicit window semantic. However, their work makes different assumption from us which relies on the data sources to generate punctuation messages at an appropriate rate.

Works on memory-constrained stream join usually focus on certain metric such as “Max-Subset” [6, 9, 12, 18] to obtain the statistically optimal solution. Different from these approaches, our proposed memory-constrained join strategies (MSF and DBF) are based on the understanding of the memory consumption patterns to produce the complete or near-complete query answers given the input statistics. The disk-based approach for joining stream data is used in [10, 14, 16]. Unlike these techniques, the DBF strategy aims to minimize the *Lag* among streams and therefore tuples stored in the disk would not match tuples from other streams that are currently buffered in the memory. This means in DBF, the join is less likely to be blocked due the disk I/Os involved.

## 7. Conclusions

In this work, we have proposed a data-driven memory management scheme where users are oblivious to the window semantic and rely on the system to dynamically estimate the state buffer size to generate quality join results. We studied the memory consumption pattern and identified intra-stream delay and inter-stream delay are the two main causes for excessive memory utilizations. Based on these observations, we derived the memory cost model and proposed several memory overhead reduction strategies. Our experimental study has demonstrated the effectiveness of our proposed techniques.

## References

- [1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
- [2] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Trans. Database Syst.*, 29:162–194, 2004.
- [3] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418, 2004.
- [4] S. Babu, U. Srivastava, and J. Widom. Exploiting *k*-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.
- [5] M. Cammert, J. Kramer, B. Seeger, and S. Vaupel. An approach to adaptive memory management in data stream systems. University of Marburg Technical Report No. 49, 2005.
- [6] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD Conference*, pages 40–51, 2003.
- [7] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.
- [8] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.
- [9] F. Li, C. Chang, G. Kollios, and A. Bestavros. Characterizing and exploiting reference locality in data stream applications. In *ICDE*, page 81, 2006.
- [10] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD Conference*, pages 347–358, 2006.
- [11] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.
- [12] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.
- [13] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003.
- [14] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
- [15] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*, pages 37–48, 2002.
- [16] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.
- [17] J. Wu, K.-L. Tan, and Y. Zhou. Window-oblivious join: A data-driven memory management scheme for stream join. Unpublished Manuscript. <http://www.comp.nus.edu.sg/~wuji/TR/>.
- [18] J. Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams. In *SIGMOD Conference*, pages 359–370, 2005.