Optimizing In-Order Execution of Continuous Queries over Streamed Sensor Data

Moustafa A. Hammad
University of Calgary
Calgary, Alberta, Canada T2N 1N4
hammad@cpsc.ucalgary.ca

Walid G. Aref Ahmed K. Elmagarmid
Purdue University
West Lafayette, IN 47907, USA
{aref,ake}@cs.purdue.edu

Abstract

The widespread use of sensor networks in scientific and engineering applications leads to increased demand on the efficient computation of the collected sensor data. Recent research in sensor and stream data systems adopts the notion of sliding windows to process continuous queries over infinite sensor readings. Ordered processing of input data is essential during query execution for many application scenarios. In this paper we present three approaches for ordered execution of continuous sliding window queries over sensor data. The first approach enforces ordered processing at the input side of the query execution plan. In the second approach we utilize the advantage of out-of-order execution to optimize query operators and enforce an ordered release of the output results. The third approach is adaptive and switches between the first and second approaches to achieve the best overall performance with current input arrival rates and level of multiprogramming. We study the performance of the proposed approaches both analytically and experimentally and under a variety of conditions such as the asynchronous arrival of input data, and various levels of multiprogramming. Our performance study is based on an extensive set of experiments using a realization of the proposed approaches in a prototype stream query processing system.

1. Introduction

Continuous queries on streaming applications depend on windows to limit the scope of interest over the infinite input streams. Several forms of windowed execution are currently proposed in the literature, of which, sliding time windows are commonly used by several stream data systems [1, 3, 10]. The following example gives a continuous query Q that computes the on-line total sales of the items sold in common by two different department stores. Sold items are identified by their RFID (Radio Frequency Identifier) tags and sensors at the checkout terminals read the RFID of each sold item. SensorReadings1 and SensorReadings2 represent the stream of sales transactions detected by each sensor, respectively. The window clause indicates that Q is interested only in the last 10 minutes of the sales from each store.

```
SELECT COUNT(DISTINCT S.ItemType)
FROM SensorReadings1 S, SensorReadings2 T
WHERE S.ItemType = T.ItemType
WINDOW 10 minutes
```

Figure 1(A) gives the pipelined evaluation of Q. In the figure, the output from joining S and T is streamed as input to the DISTINCT and then to the COUNT operators at the top of the pipeline.

The operation of the join over a sliding window (W-join) is described as follows [9]: Tuple t_k in Stream S joins with tuple t_j in Stream T iff (1) t_k and t_j satisfy the join predicate (i.e., the WHERE clause in the SQL query), (2) the timestamp of tuple t_k is within window size from the timestamp of t_j. Old tuples, say t_o, from one input stream is expired (dropped from the window) iff t_o is far by more than window size from any new tuples in the other stream. Figure 1(B) gives an example of W-join between streams S and T. The ticks on the time line of S or T are equally spaced at one time unit between two consecutive ticks. We assume that each tuple is indexed by its timestamp. As a_8 arrives, W-join drops a_1 and produces the output tuple $< a_8, a_5 >$. Similarly, as b_9 and c_{10} arrive, W-join drops d_2 and produces the output tuples $< b_6, b_9 >$ and $< c_4, c_{10} >$, respectively.

W-join as described in the previous paragraph can potentially produce an *unordered* output stream. For example, in Figure 1(C), tuple a_8 in Stream S is delayed 3 time units while tuples b_9 and c_{10} in stream T arrive without delays. In this case, W-join will process tuples b_9 and c_{10} before processing the earlier tuple a_8. This will result in an out-of-order release of the output tuples (i.e., tuples $< b_6, b_9 >$

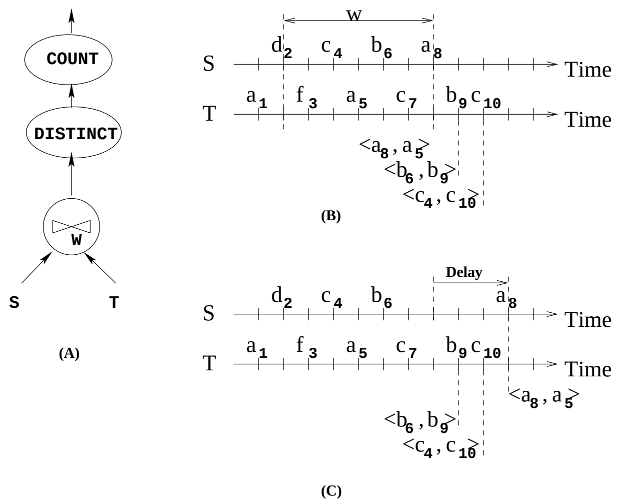


Figure 1. Motivating Example.

and $< c_4, c_{10} >$ will be released before tuple $< a_8, a_5 >$).

The notion of ordered output is crucial in the pipelined evaluation, mainly for two reasons: (1) The decision of expiring an old tuple from a stored state (e.g., a stored window of tuples in an online sliding-window $COUNT$ operation) depends on receiving an ordered arrival of the input tuples. Otherwise, we may expire an old tuple early (e.g., potentially report an erroneous sequence of count values). (2) Some important applications over data streams require processing the input of their queries in-order (and therefore, produce ordered output). This is especially true if the output from the queries is used as an input stream for further analysis, e.g., as in feedback control, periodicity detection, and trend prediction (patterns of continuous increase or decrease) in data streams.

One approach to provide *in-order* execution of input tuples is to synchronize the processing of W-join over the input streams [15]. We call this approach the *Sync-Filter* approach (for synchronize then filter). In this approach, and using the example in Figure 1(C), W-join will delay the processing of b_9 and c_{10} from stream T until verifying that a new tuple from Stream S arrives and has a larger timestamp. The obvious drawback of the *Sync-Filter* approach is that W-join will *block* waiting for new tuples at both streams before every join step. This will result in increased response times of output tuples.

In this paper, we make the following contributions:

1. We study the problem of in-order execution of W-joins in stream data systems and present the Sync-Filter approach as an alternative to provide in-order execution of W-join.

2. We study the Sync-Filter approach analytically and introduce a closed-form representation of the average response time.

3. We propose a new approach, termed the *Filter-Order* approach, and provide a closed form representation of the average response time. In addition, we study analytically the relative improvement of the Filter-Order approach over the Sync-Filter approach.

4. Based on our analytical study, we propose a third approach, termed the *Adaptive* approach, that has the advantages of the two previous approaches while avoiding their drawbacks.

5. We study the three approaches experimentally using our prototype stream data system, Nile [10]. The experimental study validates our analytical results and shows that the Adaptive approach can always achieve the targeted improvement in response time by switching between the Sync-Filter and the Filter-Order approaches.

The rest of the paper is organized as follows. Section 2 presents related work on window join over data streams. Section 3 introduces the system architecture and our basic assumptions. Section 4 presents the Sync-Filter approach of W-join. Sections 5 and 6 introduce our proposed approaches, namely the Filter-Order approach and the Adaptive approach of W-join. We present the performance study in Section 7. Section 8 contains concluding remarks.

2 Related Work

Stream query processing has been addressed by many evolving systems such as Borealis [1], Telegraph [4] and STREAM [3]. These systems suggest the support for window join processing as a practical implementation of join queries over data streams. Adaptive query processing [2] and the execution of continuous queries [7, 6] address the reordering of operators during the execution of continuous queries. The non-blocking execution and the support of continuous queries are essential for stream processing. The algorithms studied in this paper support continuous queries and are non-blocking.

The work on band join [8] addresses a binary join among stored relations where tuples are joined whenever their values are within a band of each other. This approach addresses a problem that is similar to joining two streams where the band represents a time interval between the two streams. However, the notion of ordered processing over infinite data streams is not addressed by the band join approach.

Window join processing has been addressed in [5, 11]. Psoup [5] handles streamed queries over streaming data and provides a similar definition for sliding time windows but with approximate answers. However, Psoup assumes no delays among input streams. The work in [11] addresses the window join over two streams where the two arriving

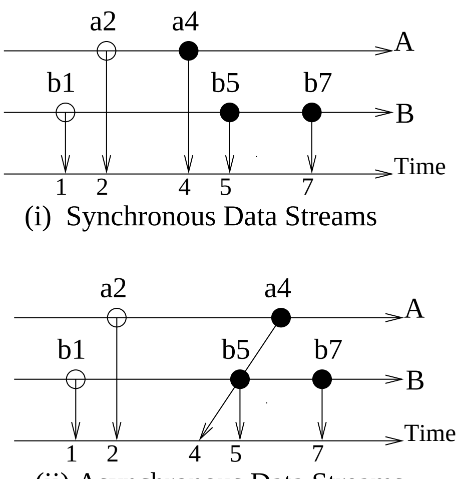


Figure 2. Synchronous vs. Asynchronous Data Streams.

streams have different arrival rates. The window is defined in terms of a tuple count. The authors suggest using asymmetric join, e.g., applying a full scan on one stream and probing a hash table on the other stream, to reduce execution cost. In contrast, we consider a time window that represents a constraint over all input streams.

3 Context and Environment

We consider a centralized stream data system, where data is collected from remote sensors for further processing and analysis. In each stream, data is ordered locally (tuples arrive with increasing timestamps). However, tuples from two different data streams are not necessarily ordered. We term this type of data streams as *asynchronous data streams*. Figures 2(i) and 2(ii) give two examples of synchronous and asynchronous data streams, respectively. In the figure, the index of each tuple represents the tuple's timestamp. For synchronous data streams (e.g., Figure 2(a)), tuples always arrive in increasing value of the timestamps across all streams. For asynchronous data streams (e.g., Figure 2(b)), tuples in the same stream always arrive with increasing values of the timestamps, however, there is no implicit order among tuples that belong to different data streams.

In the following, we illustrate two stream processing scenarios that receive asynchronous data streams. In the first scenario, a centralized stream processing unit receives data over the network from remote sensors. Each sensor augments a timestamp to the streamed data item (e.g., valid timestamp [14]). At the stream processing unit, the received data streams from various sensors are likely to become asynchronous due to the communication delays in the network channel between the physical sensor and the central processing unit. The second scenario of asynchronous data streams is when data items are timestamped upon arrival at the centralized processing unit (e.g., transaction timestamp [14]). In this case, asynchronous processing could potentially happen during the query pipelined execution. For example, a join operation, say J, in a query evaluation plan may receive two tuples, say s and t, from two different data streams. While J could receive t earlier than s, the timestamp of t could be newer than that of s. This situation may occur if t and s are processed by different child operators (e.g., different filtering operations). As a result, t and/or s could experience different processing delays before arriving at J.

Our focus in this paper is on sliding window queries defined in terms of time units. In our stream query processor, we employ the *"stream-in stream-out"* philosophy. The main idea is that since the input stream is composed of tuples that are ordered by some timestamp, the output tuples also appear as a stream that is ordered by a timestamp. A single sliding window query consists of multiple operators. These operators execute in a pipelined fashion where the output from one operator is incrementally added to the input of the next operator in the pipeline. The operators are connected by First-In-First-Out (FIFO) queues and a scheduler schedules the execution of each operator. This execution model is typical in many stream processing systems such as Fjord [12], Borealis [1], and STREAM [13]. We assume that for any continuous query, the stream data system can keep up with the aggregate arrival rates of all input streams.

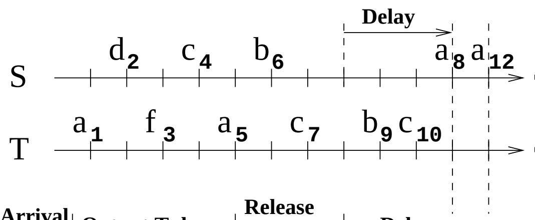
4 The Sync-Filter Approach

One straightforward approach to get ordered output from the W-join operator is by enforcing ordered processing of input tuples. In other words, for any two tuples t_i and t_{i+1} that are processed in sequence by W-join, $TimeStamp(t_i) \leq TimeStamp(t_{i+1})$. Note that t_i and t_{i+1} may not necessarily belong to the same stream. We describe the basic steps of W-join that uses the Sync-Filter Approach in the following: We consider a W-join over two input data streams, say S and T, where the *join Buffers* (i.e., the hash tables to store input tuples) over S and T are B_S and B_T, respectively. We consider a time window of size $|w|$. Output tuples are inserted in the *output queue* of the W-join operator.

1. Retrieve t_i and t_j such that t_i and t_j belong to two different input streams.

2. If t_i or t_j does not exist, return to Step 1.

3. Select $\{t_k : t_k \in \{t_i, t_j\} \wedge \text{TimeStamp}(t_k) = \min(\text{TimeStamp}(t_i), \text{TimeStamp}(t_j))\}$. /* t_k may belong to S or T, let us assume that t_k belongs to S */.



Arrival Time	Output Tulpe	Release Time	Delay
11	$<a_8, a_5>$	11+tp	tp
9	$<b_6, b_9>$	12+tp	3+tp
10	$<c_4, c_{10}>$	12+2tp	2+2tp

tp : time to process a new tuple

Figure 3. W-join using the Sync-Filter Approach.

4. Remove from B_T every tuple t_o such that: TimeStamp(t_k) - TimeStamp(t_o) $> |w|$.

5. Retrieve from B_T the set of tuples, say $\mathcal{R}$, that satisfy the join condition with t_k and add $t_k \times \mathcal{R}$ to the output queue

6. Add t_k to B_S.

7. Return to Step 1

Figure 3 gives the execution of the Sync-Filter approach for the example of Figure 1(C). As b_9 in Stream T arrives, W-join blocks waiting for another tuple from Stream S. At time 11, a_8 arrives in Stream S. W-join processes a_8 and removes a_1 from Stream T since a_8 and a_1 are far by more than window (6 time units). Finally, W-join produces the output tuple $< a_8, a_5 >$. Notice that W-join processes b_9 and c_{10} only when tuple a_{12} arrives in Stream S. At time 12, W-join processes b_9 and produces the output tuple $< b_6, b_9 >$ at time $12 + t_p$, where t_p is the time to process an input tuple by the W-join. Then, W-join processes c_{10} and produces the output tuple $< c_4, c_{10} >$ at time $12 + 2t_p$. The delay in processing every tuple is given in the rightmost column of the table in Figure 3.

4.1 Analysis

In this section, we provide an analysis of the response time of input tuples (the time elapsed between the arrival and the complete processing by the W-join). We consider a binary W-join between two streams, say S and T, where the time to perform a join operation between two tuples is c. Let λ_1 tuples/second be the average arrival rate of Stream S and let λ_2 tuples/second be the average arrival rate of Stream T. Let $|w|$ be the window size in seconds. The time to process a new input tuple from Stream S is $C_S = c|w|\lambda_1$ and the time to process a new input tuple from Stream T is $C_T = c|w|\lambda_2$. For the window join to keep up with the arrival rate, $C_S < \frac{1}{\lambda_2}$ and $C_T < \frac{1}{\lambda_1}$. Using the equations of C_S and C_T, the condition of the window join to keep up with the arrival rate is:

$$c|w| < \frac{1}{\lambda_1 \lambda_2} \qquad (1)$$

Without loss of generality, we can assume that $\lambda_1 < \lambda_2$. During $\frac{1}{\lambda_1}$ seconds (the inter-arrival times of two tuples in Stream S), n tuples could be received by Stream T. On average, $n = \frac{\lambda_2}{\lambda_1}$ tuples. The n tuples will not join immediately and must wait until a new tuple, say t_j, arrives at Stream S. Let $t_{k_1}, t_{k_2}, \ldots, t_{k_n}$ be the n tuples ordered by their arrival times. The response time of $t_{k_1} = \frac{1}{\lambda_1} + C_S$. The response time of $t_{k_2} = \frac{1}{\lambda_1} - \frac{1}{\lambda_2} + 2C_S$. The response time of $t_{k_3} = \frac{1}{\lambda_1} - \frac{2}{\lambda_2} + 3C_S$, etc. The response time of $t_{k_n} = \frac{1}{\lambda_1} - \frac{n-1}{\lambda_2} + nC_S$. The total response time of the n tuples is:

$$\frac{n}{\lambda_1} - \frac{n^2 - n}{2\lambda_2} + \frac{n(n+1)}{2} C_S \qquad (2)$$

while the average response time is:

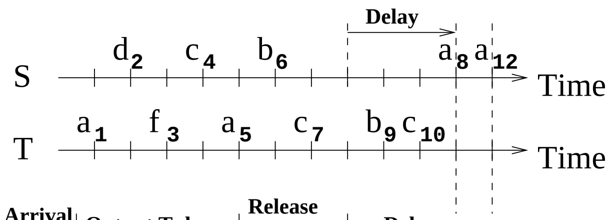
$$\frac{1}{\lambda_1} - \frac{n-1}{2\lambda_2} + \frac{n+1}{2} C_S \qquad (3)$$

4.2 Discussion

The advantage of the Sync-Filter approach, besides its simplicity and guaranteed provision of ordered output, is that W-join needs to store only those tuples that are within window from each other. Notice that tuples b_9 and c_{10} are not stored in the buffer of Stream T. Instead, b_9 and c_{10} are kept in the input queue [1]. In addition, W-join drops old tuples as new tuples are processed (e.g., dropping a_1 when W-join processes a_8). Therefore, the Sync-Filter approach eliminates the need to check the window condition (i.e., that tuples are within window from each other) while scanning the buffer of the joined stream.

One drawback of the previous approach is that W-join blocks while waiting for a delayed tuple from one stream (e.g., a_8) even though some tuples (e.g., b_9 and c_{10}) could be waiting to join in the other stream. A better approach is to *overlap* the time of processing the waiting tuples with the waiting time to receive the delayed tuple. Apparently, this new approach has to prevent the out-of-order release of output tuples (see the example in Figure 1(C)). In the following section, we describe a W-join algorithm that guarantees the ordered release of output tuples while avoiding the drawback of the Sync-Filter approach.

[1]Notice that the input queue of T will not increase indefinitely since we always assume that tuples from Stream S will eventually arrive.



Arrival Time	Output Tulpe	Release Time	Delay
11	$<a_8, a_5>$	11+tp	tp
9	$<b_6, b_9>$	12	3
10	$<c_4, c_{10}>$	12	2

tp : time to process a new tuple

Figure 4. W-join using the Filter-Order Approach.

5 The Filter-Order W-join Algorithm

In the Filter-Order W-join Algorithm (Filter-Order, for short), W-join processes input tuples independent of their global order. Furthermore, W-join buffers the output tuples before releasing them in-order. The following algorithm describes the steps of W-join using the Filter-Order approach. In this algorithm, the *hold buffer* refers to a min-heap data structure to store output tuples sorted by their timestamps. Let t_k and t_j be the last tuples received from Streams S and T, respectively. Let $TS_{trigger}$ = min$\{TimeStamp(t_k), TimeStamp(t_j)\}$, let $TimeStamp(< t_k, t_j >)$ = max$\{Timestamp(t_k), Timestamp(t_j)\}$ and let $TimeStamp(null)$ = largest defined timestamp.

1. Check for a new tuple in any of the input streams. Repeat this step until having at least a single input tuple in one of the input streams. /* Without loss of generality, assume that Stream S has a new tuple, t_k.*/

2. $TS_{trigger} = min\{TimeStamp(t_k), TimeStamp(t_j)\}$

3. Add t_k to its corresponding buffer (i.e., B_S).

4. Remove from B_T every tuple t_o such that: TimeStamp(t_k) - Timestamp$(t_o) > |w|$.

5. Retrieve from B_T the set of tuples, say $\mathcal{R}$, that satisfy the join condition with t_k and add $t_k \times \mathcal{R}$ to the hold buffer.

6. For every tuple, say t_h, at the top of the hold buffer such that $TimeStamp(t_h) \leq TS_{trigger}$, add t_h to the output queue.

7. Return to Step 1.

Figure 4 gives the execution of W-join using the Filter-Order approach. W-join processes b_9 once b_9 arrives (without blocking to wait for a_8). The output tuple $< b_6, b_9 >$ is stored in the hold buffer and is not released immediately. Similarly, W-join processes c_{10} and stores the output tuple $< c_4, c_{10} >$ in the hold buffer. W-join cannot release the two output tuples since $TS_{trigger}$ equals 6 (i.e., the timestamp of the last tuple seen from Stream S). As tuple a_8 arrives at time 11, W-join updates $TS_{trigger}$ to 8, produces $< a_8, a_5 >$ and releases this tuple immediately since $(TimeStamp(< a_8, a_5 >) = 8) \leq TS_{trigger}$. At time 12, tuple a_{12} arrives and $TS_{trigger}$ is set to 10 (the timestamp of the last tuple in Stream T). W-join can now release the output tuples $< b_6, b_9 >$ and $< c_4, c_{10} >$. Notice that the time to produce $< b_6, b_9 >$ and $< c_4, c_{10} >$ is overlapped with the waiting time to receive a_{12} and the total delay to receive the three output tuples is lower than that of the Sync-Filter approach by 3 tp.

5.1 Analysis

In this section, we study the Filter-Order approach analytically in terms of its response time. We use the same notation as that in Section 4.1. Recall that the Filter-Order approach overlaps the processing time of the new tuples from one stream with the waiting time for a new tuple from another stream. Consider the n tuples (see Section 4.1) t_{k_1}, $t_{k_2}, \ldots, t_{k_n}$. The response time of t_{k_1} is $\frac{1}{\lambda_1}$. The response time of t_{k_2} is $\frac{1}{\lambda_1} - \frac{1}{\lambda_2}$, etc. The response time of t_{k_n} is $\frac{1}{\lambda_1} - \frac{n-1}{\lambda_2}$. The total response time for all the n tuples is:

$$\frac{n}{\lambda_1} - \frac{n^2 - n}{2\lambda_2} \tag{4}$$

and the average response time is:

$$\frac{1}{\lambda_1} - \frac{n-1}{2\lambda_2} \tag{5}$$

5.2 Discussion

Memory Requirement. In addition to the join buffers of the input data streams, the Filter-Order approach needs extra memory to hold the output (the hold buffer). As long as W-join is selective, the memory consumption in the Filter-Order approach is expected to be lower than that of the Sync-Filter approach. However, for a W-join with low selectivity (a tuple from one stream joins with more than one tuple from the other stream), the hold buffer may need to store an increased amount of tuples. Since we assume that none of the input streams will block indefinitely, the memory consumption in both the Sync-Filter and Filter-Order approaches is always bounded and is not critical in this paper.

Average Response Time. By comparing the average response time of the Filter-Order approach with that of the Sync-Filter approach (i.e., Equation 3 and Equation 5), it is clear that the term $\frac{n+1}{2}C_S$ disappears in the case of Filter-Order since the processing time overlaps the waiting time. Therefore, the average output response time is expected to improve when using the Filter-Order approach. However, we still need to evaluate the relative significance of this improvement especially as the Filter-Order consumes more memory. In the following, we study analytically the relative improvement in average response time when using the Filter-Order approach over the Sync-Filter approach. Such measure will be referred to as I_{Rel}.

I_{Rel} is obtained by simply dividing the difference between the average response time of Sync-Filter (Equation 3) and Filter-Order(Equation 5) by the average response time of the Sync-Filter approach (Equation 3):

$$I_{Rel} = \frac{\frac{n+1}{2}C_S}{\frac{1}{\lambda_1} - \frac{n-1}{2\lambda_2} + \frac{n+1}{2}C_S} \qquad (6)$$

However, $C_S = c|w|\lambda_1$ and $n = \frac{\lambda_2}{\lambda_1}$ (from Section 4.1). By substituting these values in Equation 6 and reducing the equation, we get the following:

$$I_{Rel} = \frac{c|w|\lambda_2\lambda_1}{1 + c|w|\lambda_2\lambda_1} \qquad (7)$$

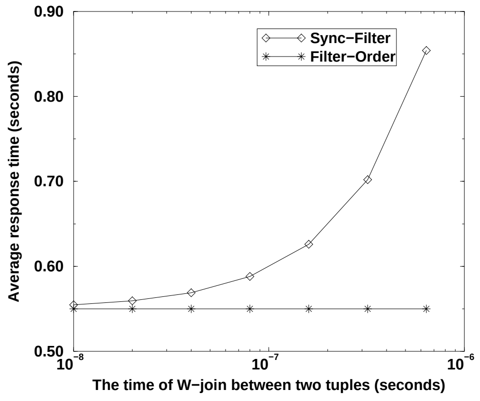
Therefore, the upper-bound of I_{Rel} is:

$$\lceil I_{Rel} \rceil = \lim_{c|w| \to \frac{1}{\lambda_2\lambda_1}} I_{Rel} = 0.5 \qquad (8)$$

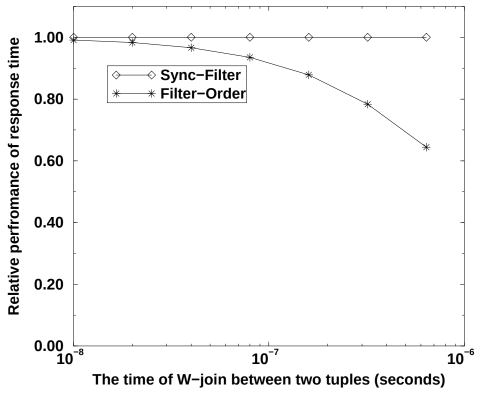
In the following, we provide two numerical examples that compare the performance of the Sync-Filter and the Filter-Order approaches.

Example 1: Figure 5 (a) gives the expected performance of W-join when using the Sync-Filter and the Filter-Order approaches. The example uses Equation 3 and Equation 5 while increasing the processing time (i.e., c in Section 4.1). In this experiment λ_1 is set to 1 tuple/second and λ_2 is set to 10 tuples/second. The window size is set to one day and c starts at 0.01 μ second with double increase at each new point. Figure 5 (b) gives the relative performance of the same experiment. As c increases, the average response time of the Sync-Filter approach increases while the average response time of the Filter-Order approach remains fixed (since the average response time of the Filter-Order approach does not depend on c). As c increases to the point that the system can hardly keep up with the arrival rate (i.e., $c|w| \approx \frac{1}{\lambda_1\lambda_2}$), the Filter-Order approach becomes closer to $\lceil I_{Rel} \rceil$ of 0.5.

Example 2: We repeat the previous experiment using low arrival rate in both streams ($\leq$ 1 tuple/second) and varying the arrival rate of stream S (e.g., λ_1) between 0.0001 tuples/second and 0.1 tuples/second. We set the window size to one day and set c to 0.0001 seconds. Figure 6(a) and Figure 6(b) give the absolute and relative performances of W-join using the Sync-Filter and the Filter-Order approaches, respectively. In this example, the Filter-Order achieves a significant absolute as well as relative reduction in average response time (i.e., from 10 seconds to 5 seconds) as the arrival rate increases.



(a)



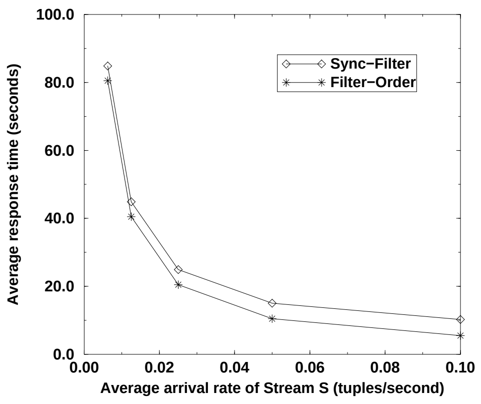
(b)

Figure 5. Example 1: Analytical comparison between the Sync-Filter and the Filter-Order approaches while varying the processing speed.

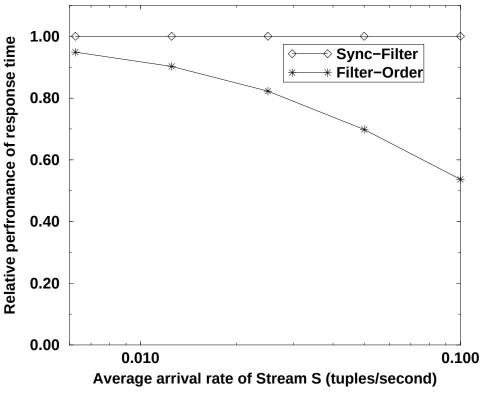
We validate the results obtained from the analytical study in the performance study section.

6 The Adaptive Algorithm

Equation 7 shows that the relative performance improvement when using the Filter-Order approach is significant at specific ranges of arrival rates and processing speeds. Otherwise, the Sync-Filter approach is a valuable option espe-



(a)



(b)

Figure 6. Example 2: Analytical comparison between the Sync-Filter and the Filter-Order approaches while varying the input arrival rate of Stream S.

cially as we consider the low memory overhead in the Sync-Filter approach. In this section we introduce the Adaptive approach that keeps the advantages of both the Sync-Filter and the Filter-Order approaches while avoiding their drawbacks.

Initially the Adaptive W-join algorithm adopts the Sync-Filter approach. However, the Adaptive approach performs two extra steps:

1. Monitor λ_1 and λ_2 (the arrival rates at the input data streams S and T, respectively.)
2. Verify the following condition that is based on Equation 1: $c|w|\lambda_1\lambda_2 \geq \alpha$, where $0 \leq \alpha < 1$.

When the condition in Step 2 is fulfilled, the Adaptive approach switches to the Filter-Order approach while continuing to perform the above two steps. The Adaptive approach switches back to Sync-Filter when the test condition in Step 2 is false. α is a user-input parameter and indicates the required relative performance. For example, when α equals 0.9 and the condition in Step 2 is TRUE, a relative improvement of at least $\frac{\alpha}{1+\alpha}$ [2] or 0.47 is achieved when using the Filter-Order approach.

The steps of the Adaptive approach are as follows:

(A) Steps A.1 to A.6 are Steps 1 to 6 of the Sync-Filter Algorithm (refer to Section 4)./* *Start with Sync-Filter */*

(B) If $c|w|\lambda_1\lambda_2 < \alpha$, Goto Step A.1 /**Continue using Sync-Filter*/.*

(C) Steps C.1 to C.6 are Steps 1 to 6 of the Filter-Order Algorithm (refer to Section 5.) /* *Switch to Filter-Order */*

(D) If $c|w|\lambda_1\lambda_2 \geq \alpha$, Goto Step C.1 /* *Continue using Filter-Order */.*

(E) Retrieve t_i and t_j such that t_i and t_j belong to two different input streams. If t_i or t_j does not exist, repeat Step E./* *Switch to Sync-Filter - Clean hold-buffer first*/*

(F) $TS_{trigger} = min\{TimeStamp(t_i), TimeStamp(t_j)\}$. /* Without loss of generality, assume that tuple $TimeStamp(t_i) = TS_{trigger}$ and $t_i \in$ Stream S */

(G) Add t_i to its corresponding buffer (i.e., B_S).

(H) Remove from B_T every tuple t_o such that: TimeStamp(t_i) - Timestamp(t_o) $> |w|$.

(I) Retrieve from B_T the set of tuples, say $\mathcal{R}$, that satisfy the join condition with t_i and add $t_i \times \mathcal{R}$ to the hold buffer.

(J) For every tuple, say t_h, at the top of the hold buffer such that $TimeStamp(t_h) \leq TS_{trigger}$, add t_h to the output queue.

(K) If the hold buffer is not empty, return to Step E.

(L) Return to Step A.1 /* *Start Sync-Filter*/.*

7 Performance Study

7.1 Experiments Setup

The experiments are performed on a prototype stream query processor, Nile [10]. A sensor is represented using a *stream-type* that provides the following interfaces, *InitStream, ReadStream, and CloseStream.* In order to collect data from the streams and supply them to the query execution engine, Nile has a *stream manager* that registers new

[2]The term is obtained by substituting $c|w|\lambda_1\lambda_2$ in Equation 7 by α

stream-access requests, retrieves data from the registered streams into its local buffers, and supplies data to be processed by the query execution engine. The stream manager runs as a separate thread and schedules the retrieval of tuples in a round robin fashion. To interface the query execution plan to the stream manager, Nile uses a *StreamScan operator* to communicate with the stream manager and receive new tuples as they are collected by the stream manager. We use a hash-based implementation of W-join as in [9]. The join buffers are structured as hash tables that have the join attribute as the hash key. We have implemented the steps of the proposed algorithms in Sections 4, 5, and 6. The hold buffer in the Filter-Order and the Adaptive approaches is implemented as a dynamic min-heap structure. The Adaptive approach maintains estimates of the average arrival rate per input stream and the average processing speed.

Our measure of performance is the *average response time per input tuple*, which is the average time to completely process an input tuple by W-join. This time includes the waiting time, the processing time, and the time to produce an output tuple (if any). We perform our experiments on synthetic data streams, where each stream consists of a sequence of integers. In the experiments, the inter-arrival time between two consecutive tuples of an input data stream follows the Exponential distribution with mean $\frac{1}{\lambda}$. All the experiments are run on an Intel Pentium 4 CPU 2.4 GHz with 512 MB RAM running Windows XP.

7.2 Experimental Results

To verify our analytical findings, we perform the experiments presented in Section 5.2.

7.2.1 Varying the Number of Concurrent Queries

In this experiment, we study the performance of the proposed approaches as we vary the number of concurrent queries. Our workload is a set of concurrent W-join queries over two data streams, S_1 and S_2. We measure the time to process a single W-join operation per query (parameter c in Section 4.1) as we increase the number of concurrent queries. Since c is directly proportional to the number of concurrent queries in our workload, we vary the value of c by varying the number of concurrent queries. We use a window of size one minute. The average stream arrival rates in S_1 (the slow stream) and S_2 (the fast stream) are 1 tuple/second and 10 tuples/second, respectively. We set α of the Adaptive approach to 0.3 (i.e., we would like to switch to Filter-Order if the relative improvement is greater or equal to $\frac{0.3}{1+0.3}$ or $\approx 25\%$). We collected the average response time of the input tuples during the lifetime of the experiment (20 minutes for each run).

Figure 7 gives the average response time when varying c between 1 microsecond and 1 millisecond. Y-axis is the

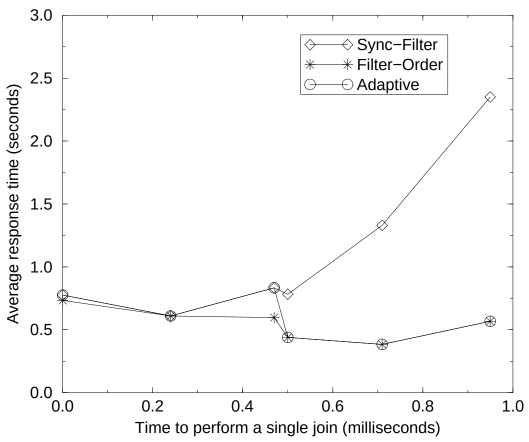


Figure 7. Average response time while varying the number of concurrent queries.

average response time per input tuple. Figure 8 gives the performance of the proposed approaches relative to Sync-Filter.

With all processing times, Sync-Filter has the worst average response time. At large processing times, the difference between Sync-Filter and Filter-Order is significant and the difference gets smaller at small processing times. This can be interpreted as follows: Using Sync-Filter while increasing the processing time per join tuple, leads to excessive delays of tuples in the fast stream (i.e., S_2). This is the case as new tuples from S_2 must wait for a new tuple from the slow stream (i.e., S_1) to proceed in W-join. On the other hand, Filter-Order shows small or no variations in the average response time as we increase the processing time. This is mainly a result of overlapping the processing of tuples from S_2 while waiting for new tuples from S_1. The Adaptive approach behaves similar to Sync-Filter in our first three measurement since the condition $c|w|\lambda_1\lambda_2 < \alpha$ is true. At $c = 0.5$ milliseconds, $c|w|\lambda_1\lambda_2 = 0.0005*60*10*1 = 0.3$ (i.e., $\geq \alpha$). Therefore, the Adaptive approach *switches* to the Filter-Order approach. One interesting observation is that the analytical measure of the relative performance is *conservative*. For example, at $\alpha = 0.3$, we expect Filter-Order to perform 25% better than Sync-Filter. Experimentally, the performance improvement was around 45%. This is the result of using an estimate of c to evaluate the condition $c|w|\lambda_1\lambda_2 < \alpha$.

In Figure 8, the relative improvement exceeds 50% (our upper bound) at the last two measurements because the processing speed of Sync-Filter starts to lag behind the input arrival rate.

The performance of Sync-Filter and Filter-Order in Figures 7 and 8 is similar to that in Figures 5(a) and 5(b), re-

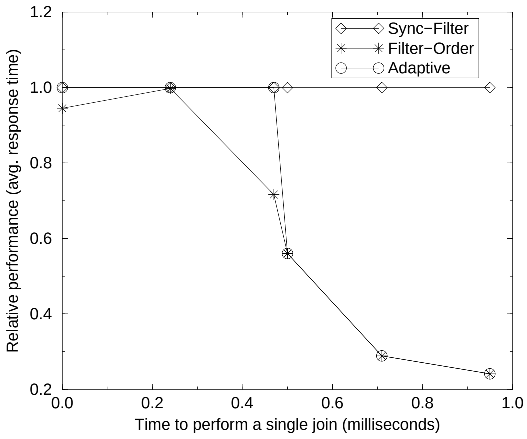


Figure 8. Relative performance among the proposed approaches in terms of average response time (normalized by that of Sync-Filter.)

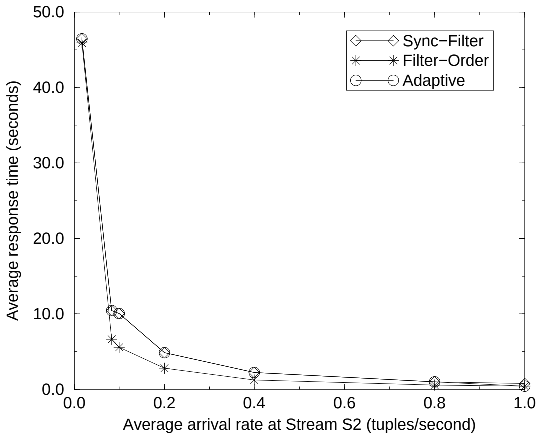


Figure 9. Average response time while varying the input rate of S_1 (the slow stream).

spectively. Recall that the curves in Figures 5(a) and 5(b) are based on the analytical derivation in Equations 3 and 5. This verifies our analytical derivation in Sections 4.1 and 5.1.

7.2.2 Changing Input Rate

In this experiment, we study the effect of the proposed approaches on the average response time of the input tuples while varying the arrival rate of the slow stream. We use a binary W-join with a window size of one minute as in the previous experiment. We fix the input rate of the fast stream (S_2) at 10 tuples/second and vary the input rate of the slow stream (S_1) between 0.01 and one tuple/second. As in the previous experiment, the Adaptive approach uses $\alpha = 0.3$. We fix the multiprogramming level such that $c \approx 0.5$ milliseconds.

Figure 9 gives the average response time while Figure 10 gives the performance of the proposed approaches relative to Sync-Filter. In all the proposed approaches, the average response time increases significantly (more than one minute) at small arrival rate of the slow stream. Obviously, this is a direct result of having large n ($n = \frac{\lambda_2}{\lambda_1}$) in Equations 3 and 5. However, the increase in Sync-Filter is larger than that of Filter-Order for the same reasons, as explained in the previous experiment.

¿From Figure 10, the relative improvement of Filter-Order over Sync-Filter is small at low arrival rates and gets larger as the rate of the slow stream increases. Recall that as we increase the rate of the input data streams, the term $c|w|\lambda_2\lambda_1$ in Equation 7 becomes more significant and increases the overall relative improvement between Filter-Order and Sync-Filter.

Similar to the behavior in the previous experiment, the Adaptive approach switches between Sync-Filter and Filter-Order when the slow stream rate is one tuple/second. Having smaller α will shift the switching point to a small arrival rate of the slow stream.

Finally, this experiment supports our analytical derivations in Section 5.2. This follows if we compare the performance trends of Sync-Filter and Filter-Order in Figures 9 and 10 with those in Figures 6(a) and 6(b), respectively.

7.2.3 Joining more than two data streams: Discussion

Although this paper focuses on binary W-joins, the proposed approaches are directly applicable when joining more than two data streams. Generally, two W-join implementations are possible for this case: (1) The multi-way W-join and (2) a pipeline of more than one binary W-join. For the first implementation, our proposed approaches can be extended in a straightforward fashion to consider more than two data streams. For example, Sync-Filter will need to synchronize new tuples based on receiving an input tuple from each of the input streams. For the pipelined implementation of W-join, our approaches apply directly to each binary W-join in the pipeline.

The cases of W-join where the window is different per input stream and where the window is based on tuple count are beyond the scope of this paper.

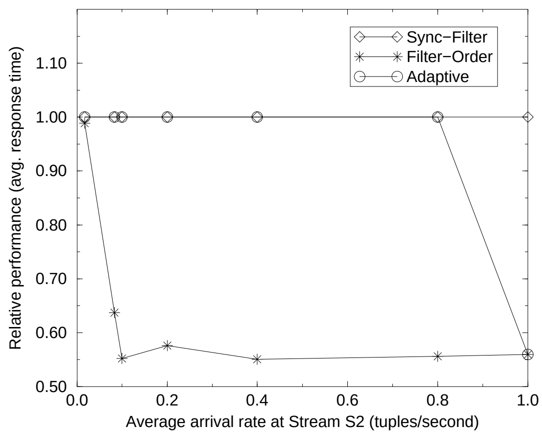


Figure 10. Relative performance among the proposed approaches in terms of the average response time (normalized by that of Sync-Filter).

8. Conclusion

Ordered evaluation of continuous queries over data streams is crucial in stream processing systems. In this paper, we studied the problem of providing ordered execution of window joins over asynchronous data streams. We showed that the Sync-Filter approach that enforces ordered processing of input tuples to guarantee ordered output can result in increased response time. We then proposed the Filter-Order approach that applied the filter step of the window join followed by the buffering and ordering steps. In this way, the processing time of input tuples from one stream will overlap the waiting time to receive synchronous tuples from the other stream. We illustrate that when both Sync-Filter and Filter-Order approaches provide comparable performance, Sync-Filter is recommended because of the low memory overhead. We studied both the Sync-Filter and the Filter-Order approaches analytically and verified the relative performance improvement. Based on the analysis, we proposed the Adaptive approach that switches between Sync-Filter and Filter-Order to achieve a given performance improvement goal. We showed both analytically and through real implementation of the approaches on a prototype stream data system the superiority of our proposed approach over the Sync-Filter approach.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, and et al. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the SIGMOD Conference, June*, 2000.

[3] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(1), 2001.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, and et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the CIDR Conference, January*, 2003.

[5] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proceedings of the VLDB Conference, August*, 2002.

[6] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proceedings of the ICDE Conference, February*, 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagracq: A scalable continuous query system for internet databases. In *Proceedings of the SIGMOD Conference, June*, 2000.

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the VLDB Conference, September*, 1991.

[9] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proceedings of the SSDBM Conference, July*, 2003.

[10] M. A. Hammad, M. F. Mokbel, M. H. Ali, and et al. Nile: A query processing engine for data streams. In *Proceedings of the ICDE Conference, March*, 2004.

[11] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the ICDE, February*, 2003.

[12] S. Madden and M.J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the ICDE Conference, February*, 2002.

[13] R. Motwani, J. Widom, A. Arasu, and et al. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the CIDR Conference, January*, 2003.

[14] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann, 2000.

[15] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proceedings of the PODS Conference, June*, 2004.