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

In this paper we study the problem of providing ordered execution of time-based sliding window queries over input streams of sensor data with inherent delays. We present three approaches to achieve the ordered execution. 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 while using various system configurations. Our performance study is based on an extensive set of experiments using a realization of the proposed approaches in Nile, 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, time-based sliding windows are commonly used by several stream data systems [1, 2, 6, 7]. Figure 1(A) gives the pipelined evaluation of an example continuous query Q that computes the online total count of the items sold in common by two different department stores. Q uses a window w time units. 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.

*This research was supported in part by the National Science Foundation under Grants: IIS-0093116, IIS-0209120, and 0010044-CCR.

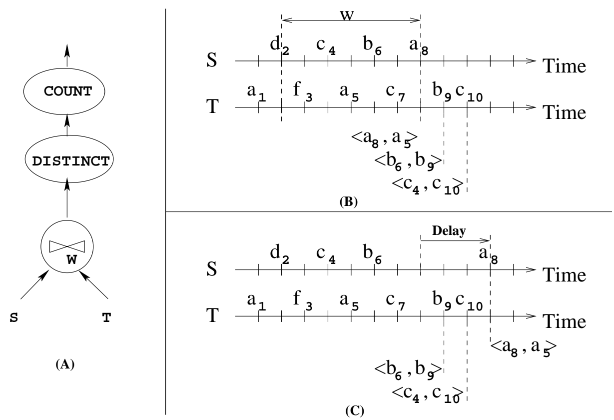
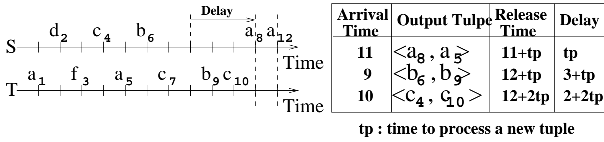


Figure 1. Motivating example.

The operation of the join over a sliding window (W-join) is described as follows [3, 4, 6]: 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 maximum timestamp (i.e., $TimeStamp(a_k) = k$ and $TimeStamp(< a_i, a_j >) = max(i, j)$). 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-



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 2. The Sync-Filter approach of W-join.

of-order release of the output tuples (i.e., tuples $< b_6, b_9 >$ 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, e.g., as in feedback control, periodicity detection, and trend prediction, require processing the input of their queries in-order (and therefore, produce ordered output). One approach to provide *in-order* execution of input tuples is to synchronize the processing of W-join over the input streams [7]. 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 study the Sync-Filter approach in terms of the average response time. Then, we propose a new approach, termed the *Filter-Order* approach, and provide a closed form representation of the average response time. Based on the analytical study, we propose a third approach, termed the *Adaptive* approach, that has the advantages of the two previous approaches while avoiding their drawbacks. We study the three approaches experimentally using our prototype system, Nile, which is a centralized stream data system that executes time-based sliding window queries [6]. 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 the Sync-Filter approach of W-join. Sections 3 and 4 introduce our proposed approaches, namely the Filter-Order approach and the Adaptive approach of W-join. We present the performance study in Section 5. Section 6 contains concluding remarks.

2 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.

Figure 2 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 2.

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)).

3 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.

Figure 3 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 $< b_6, b_9 >$

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

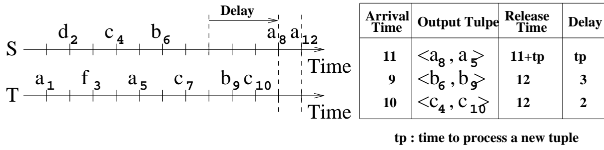


Figure 3. The Filter-Order approach of W-join.

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 the minimum timestamp of the last tuple seen from Streams S or T, $TS_{trigger}$, equals 6 (< 9). 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. 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.

By comparing the average response time of the Filter-Order approach with that of the Sync-Filter approach, it is clear that the processing time overlaps the waiting time. Therefore, the average output response time is expected to improve when using the Filter-Order approach. Let the time to perform a join operation between two tuples be 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 $|w|$ is the window size in seconds. The relative improvement in average response time when using the Filter-Order approach over the Sync-Filter approach[2], I_{Rel}, is:

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

4 The Adaptive Algorithm

Equation 1 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 especially as we consider the low memory overhead in the Sync-Filter approach. In this section we introduce the Adaptive approach that switches between the Sync-Filter and the Filter-Order approaches to achieve the best average response time. Initially the Adaptive W-join algorithm adopts the Sync-Filter approach, while performing two extra steps. Step 1: Monitor λ_1 and λ_2 (the arrival rates at the input data streams S and T, respectively.) Step 2: Verify the following condition: $c|w|\lambda_1\lambda_2 \geq \alpha$, where $0 \leq \alpha < 1$. α is a user-input parameter and indicates the required relative performance. 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. For example, when α equals 0.9 and the condition in Step 2 is TRUE, a relative improvement of at least $\frac{\alpha}{1+\alpha}$ [3] or 0.47 is achieved when using the Filter-Order approach.

5 Performance Study

The experiments are performed on a prototype stream query processor, Nile, and uses a hash-based implementation of the W-join [6]. The join buffers are structured as hash tables that have the join attribute as the hash key. We have implemented the proposed algorithms in Sections 2, 3, and 4. 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.

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 3) 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 rate 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 4 (a) gives the average response time when increasing c from 1 microsecond to 1 millisecond. Y-axis is the average response time per input tuple. 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

[2] The details of the equations's derivation is presented in [5].

[3] The term is obtained by substituting $c|w|\lambda_1\lambda_2$ in Equation 1 by α.

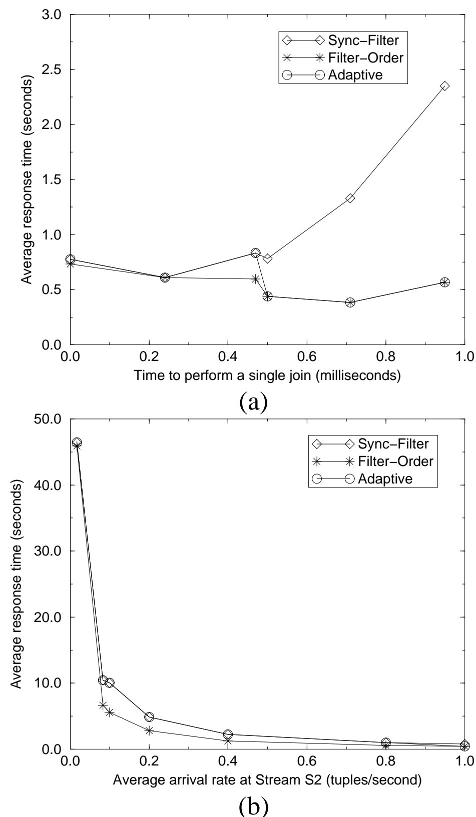


Figure 4. The average response time while (a) varying the number of concurrent queries, (b) varying the input rate of S_1 (the slow stream)

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 measurements since $c|w|\lambda_1\lambda_2 < \alpha$. At $c = 0.5$ milliseconds, $c|w|\lambda_1\lambda_2 = 0.3$ (i.e., $\geq \alpha$). Therefore, the Adaptive approach *switches* to the Filter-Order approach.

Changing Input Rate. In this experiment, we study the effect of the proposed approaches on the average response time 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 increase the input rate of the slow stream (S_1) from 0.01 to 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 4 (b) gives the average response time. In all the proposed approaches, the average response time increases significantly (more than one minute) at small arrival rates of the slow stream. However, the increase in Sync-Filter is larger than that of Filter-Order for the same reasons, as explained in the previous experiment. Similar to the behavior in the previous experiment, the Adaptive approach switches between Sync-Filter and Filter-Order when the rate of the slow stream is one tuple/second. Having smaller α will shift the switching point to a small arrival rate of the slow stream.

6. Conclusion

In this paper, we studied the problem of providing ordered execution of window joins over 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 overlaps the waiting time to receive delayed tuples from the other stream. We studied both Sync-Filter and Filter-Order analytically and based on this analysis, we proposed the Adaptive approach that switches between Sync-Filter and Filter-Order to achieve a given performance goal. We showed through real implementation of the approaches on Nile the superiority of our proposed approaches over the Sync-Filter approach.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, and et al. The Design of the Borealis Stream Processing Engine. In *Proc. of the CIDR Conf. Jan.*, 2005.

[2] S. Chandrasekaran, O. Cooper, A. Deshpande, and et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the CIDR Conf., Jan.*, 2003.

[3] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. of the VLDB Conf., Sep.*, 1991.

[4] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of the VLDB Conf., Sep.*, 2003.

[5] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Optimizing In-Order Execution of Continuous Queries over Streamed Sensor Data. In *University of Calgary, Department of Computer Science TR#2004-766-31, Apr.*, 2005.

[6] M. A. Hammad, M. F. Mokbel, M. H. Ali, and et al. Nile: A query processing engine for data streams. In *Proc. of the ICDE Conf., Mar.*, 2004.

[7] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of the PODS Conf., Jun.*, 2004.