Efficient Pipelined Execution of Sliding Window Queries over Data Streams

M. A. Hammad T. M. Ghanem W. G. Aref A. K. Elmagarmid M. F. Mokbel
Department of Computer Sciences
Purdue University
West Lafayette, IN., USA
{mhammad,ghanemtm,aref,ake,mokbel}@cs.purdue.edu

Abstract

Emerging data stream processing systems rely on windowing to enable on-the-fly processing of continuous queries over unbounded streams. As a result, several recent efforts have developed window-aware implementations of query operators such as joins and aggregates. This focus on individual operators, however, ignores the larger issue of how to coordinate the pipelined execution of such operators when combined into a full windowed query plan. In this paper, we show how the straightforward application of traditional pipelined query processing techniques to sliding window queries can result in inefficient and incorrect behavior. Then, we present two execution techniques, namely the Time Probing Approach (TPA) and the Negative Tuple Approach (NTA), that guarantee correct behavior for pipelined sliding window queries. TPA provides the best performance for sliding window queries that include a single-window; while NTA performs the best for sliding window queries that include multiple windows. A detailed performance study has been conducted using a prototype stream database system and both real and synthetic data streams. In addition to correct execution, on average, our proposed approaches provide an order of magnitude reduction in delays of the query answers when compared to conventional pipelined execution.

1 Introduction

Data stream applications such as network monitoring, online transaction flow analysis, and sensor processing pose tremendous challenges for database systems. One major challenge is the development of techniques for providing continuously updating answers to standing queries over potentially unbounded streams. The basic approach for addressing this challenge is the introduction of *windows* for queries. Window clauses added to standing queries define a continuous segmenting of the input data streams. At any instant, the window defines the set of tuples that must be considered by the query in order to produce an output. The continuous application of window clauses as new data arrives at the query processor results in incremental processing of input data streams. Combined with various types of non-blocking query operators, this incremental processing results in a system that continuously can provide query answers on-the-fly, even when effectively the input streams are never-ending.

A number of recent research efforts have introduced algorithms for *windowed* versions of one or more relational operators (e.g., see [3, 6, 8, 20, 21]). Current techniques, however, are limited in the following aspects: (1) Window algorithms have been proposed for only a few query operators (e.g., joins [6, 12] and aggregates [9, 11, 22]). (2) The focus has been on the execution of individual operators. The interaction among multiple operators in a pipelined query plan has largely been ignored. (3) Window algorithms have the ability to add to the output result incrementally. However, they do not have the ability to undo parts of the previously reported result, which is an essential operation in some operators (e.g., MINUS). (4) Scheduling strategies such as Chain [1] and Train processing [4] have focused on memory optimization for processing streams with high input rates. As we discuss later in the paper, these techniques do not address the problems that arise when considering pipelined execution of window queries with highly selective operators or low arrival rates of input streams.

In this paper, we address limitations faced in stream query processing by introducing the *Time Probing* and

Negative Tuple approaches to handle pipelined query execution. The Time Probing Approach (TPA for short) utilizes the notion of time to expire stored tuples during query execution. TPA provides the best performance for sliding window queries with a single time window and can be easily integrated with many of the window operators proposed in the literature. The Negative Tuple Approach (NTA for short) builds on a different and more general model of execution. In this model, the processing of a sliding window query can be expressed as processing a sequence of positive and negative tuples [16] or as processing a sequence of insertions and deletions [2]. NTA provides the best performance of sliding window queries with multiple windows and can accommodate general notions of window expiration and predicate windows easily. We study the proposed approaches for sliding window queries with either single or multiple windows (i.e., a different window for each input stream [18]) while considering a wide span of stream arrival rates. The proposed approaches are implemented inside a prototype stream query processor, Nile [16], which executes optimizer-based query evaluation plans that consist of multiple pipelined operators. Pipelined operators are connected through First-In-First-Out (FIFO) queues. Similar execution models are adopted in other stream processing systems (e.g., Fjord [19], Aurora [3] and STREAM [21]). Many variants of window queries are proposed in the literature; for concreteness, in this paper we focus on one particular (and we believe, common) window query type, namely, *sliding window queries* that are defined in terms of *time units*.

The contributions of this paper can be summarized as follows:

1. We present a definition of correctness for sliding-window query plans and show how the straightforward application of existing pipelined query processing techniques can result in incorrect or inefficient behavior (Section 2).

2. We propose two approaches, namely the *Time Probing* and *Negative Tuple* approaches, for correct and efficient execution of pipelined query plans (Section 3).

3. We present a classification of window operators based on their input/output characteristics. We describe new algorithms for the windowed operators. (Section 4).

4. We implement the proposed approaches in Nile [16] and provide a detailed set of experiments to show the advantages of the proposed schemes. The experiments are performed using both real and synthetic data streams (Section 5 and Section 6).

The rest of the paper is organized as follows. Section 2 provides the definition of correctness. In Section 3, we propose the *Time Probing* and *Negative Tuple* approaches for correct and efficient pipelined query execution. Section 4 describes a classification of windowed operators, and presents new algorithms for the windowed operators. In Section 5, we present the realization of the proposed approaches inside a prototype stream database engine. Section 6 provides an extensive set of experiments that study the performance of the proposed approaches. Section 7 highlights related work in stream query processing. Finally, Section 8 concludes the paper.

2 Correctness and Motivation

In this section, we introduce a definition of correctness of continuous sliding window queries. Then, we show that the conventional pipelined execution fails to adapt to the correctness definition. The correctness measure assumes that both input and output tuples are timestamped and that the order of processing of input tuples is preserved. In other words, the query operators always produce their output tuples with timestamps that are monotonically increasing.

2.1 Correctness Measure

Let $WQ(I_1, I_2, \ldots, I_n, w_1, w_2, \ldots, w_n)$ be a sliding window query, where I_j is the j^{th} input data stream and w_j is I_j's sliding window.

Definition 1 *A correct execution of WQ must provide, for each time instance T, output that is equivalent to that of a snapshot query Q that has the following properties: (1) Q is a relational query consisting of WQ with its window clauses removed. (2) The input to Q from each stream I_j is the set of input tuples that arrived during the time interval $T - |w_j|$ and T.*

Similar notions of correctness have been proposed in other systems, e.g., [21, 24].

2.2 Problem I: Delayed Answers

Figure 1(a) gives the SQL representation and pipelined execution plan of the continuous sliding window query Q_1 *"Continuously, report the total sales of items with price greater than 4 in the last hour"*. The Select and SUM operators are scheduled as in the conventional query pipeline (Input Triggered Approach).

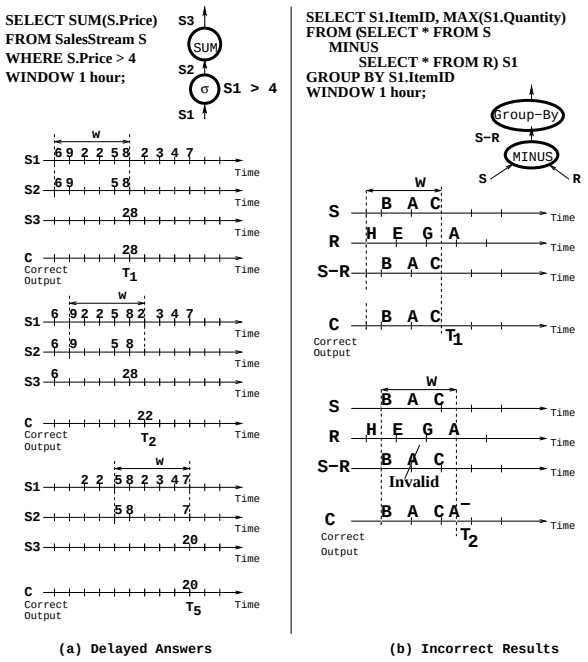


Figure 1. Motivating examples

In the *Input Triggered Approach*, an operator, say Op, is scheduled only when an input tuple arrives at Op's input. $S1, S2$, and $S3$ represent the input stream, the output stream after the Select operator, and the final output stream after applying the SUM operator, respectively. Stream C represents the expected correct output from Q_1 when the query reacts to the arrival of new input as well as the expiration of the tuples exiting from the sliding window. For simplicity, in the example, we assume that tuples arrive at equal intervals. At $S3$, the reported value for the sum is correct at times T_1 (28) and T_5 (20), but is incorrect in between. For example, the correct output at time T_2 is 22 (due to the expiration of the old tuple 6). Similarly, at time $T_2 + 1$ (not shown in the figure), the correct SUM is 13 due to the expiration of tuple 9). However, because of the Input Triggered scheduling, the SUM operator will not identify its expired tuples until receiving an input at time T_5. Note that the SUM operator could have reported the missing values (e.g., 22 and 13) at time T_5. In this case, the output in $S3$ at time T_5 will match the correct output. However, this is totally dependent on the pattern of input data and will include a significant delay. For example, in $S3$, if both 22 and 13 are released immediately before 20, the output delays for each is $T_5 - T_2$ and $T_5 - T_3$, respectively. Thus, at best, the Input Triggered Approach would result in an *increased delay* of the output.

2.3 Problem II: Incorrect Result

Figure 1(b) gives the SQL representation and pipelined execution plan for Q_2 *"For each sold item in SalesStream S and not in SalesStream R, continuously report the maximum sold quantity for the last hour"*. S and R represent the two input streams to the MINUS operator, while $S - R$ and C represent the output and the correct answer, respectively. Until time T_1, the MINUS operator provides a correct answer. At time T_2, A is added to R and therefore, A is no longer a valid output in $S - R$. Notice that A was still within the current window when A became *invalid*. In this case, the MINUS operator needs to invalidate a previously reported output tuple by generating an *invalid* output tuple. Let A^- be the invalid output tuple produced by MINUS in the correct output of Stream C at time T_2. A^- removes any effect of the previously output A in Stream C. Note that, in this scheme, parent operators of MINUS (e.g., Group-By in this case) must be able to react to the arrival of an *invalid* tuple. Thus Q_2 indicates that the incremental evaluation of window operators needs to incorporate a new type of output/input tuple, i.e., *invalid tuple*.

3 Proposed Scheduling Approaches

3.1 Approach I: Time Probing

Recall that in the example of Figure 1(a), the incorrect output was the result of scheduling higher operators in the query plan only when input tuples exist at their input queues (e.g., the SUM operator). The *Time Probing Approach* (TPA, for short) avoids this delayed processing by scheduling an operator when any of the following two events happen. The first event is when an input tuple arrives at the operator's input queue. The second event is when a stored tuple expires. The second event can only occur for *statefull* operators (i.e., operators that store a set of tuples during execution such as window join and window aggregate operators). The steps of the TPA-Scheduler are as follows:

1. Retrieve the next operator, say Op, from the queue of scheduled operators.

2. If an input tuple, say t, exists at the input queue of Op, then schedule Op to process t.

3. If Op is a *statefull* operator and a stored tuple, say t_o, in Op expires, then schedule Op to remove t_o and produce a new output (if any).

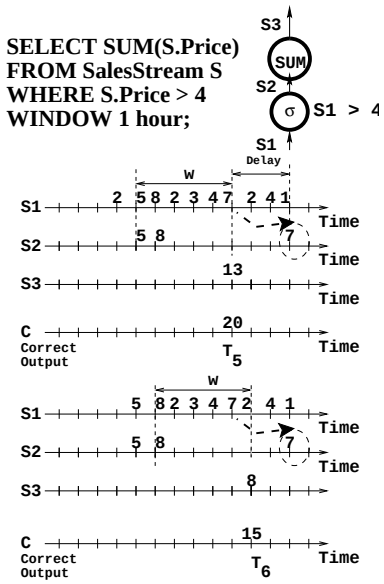


Figure 2. Voluntary expiration problem

The first two steps of the TPA-Scheduler are self-explained. Step 3 schedules a statefull operator to expire a stored tuple even without a new input tuple. Notice that the naïve implementation of this step, which voluntarily expires a stored tuple based solely on the operator's clock, could produce *incorrect* results as we illustrate by the example in Figure 2. In this example, we introduce a three clock-tick delay between the time that the tuple of value 7 is received at $S1$ and the time it is received at $S2$. Such delays are likely to occur as tuples incur different processing speeds with different operators. Stream C represents the correct results when receiving and processing the input value 7 with no delays (in this case the Stream C will be similar to the case in Figure 1(a) at time T_5). As a result of expiring stored tuples voluntarily, the SUM operator will expire tuple 5 at T_6 and produce an *incorrect* SUM 8 in $S3$. Notice that value 8 never occurs in Stream C. Moreover, the correct SUM value of 20 (in Stream C at time T_5) never appears in Stream $S3$. Thus, by voluntarily expiring old tuples without checking for new tuples, which could be delayed in the pipeline, the operator can produce a *nondeterministic* and incorrect output.

Therefore, when scheduling an operator in Step 3, the operator *probes* its descendants in the pipeline for the oldest tuple, say t_o, that is being processed. The *probe path* ends at another stateful operator or at the scan operator. Since tuples always arrive at an operator in increasing timestamp order, the operator can use the timestamp of t_o to determine whether or not a stored tuple, t, can be expired. Let $|w|$ be the window size and TS be the timestamp of tuple t. Then, t is expired during a time probe only if: $t_o.TS$ - $t.TS > |w|$. This condition is valid for a sliding window query WQ with a single window. For sliding window queries that contain multiple windows, each tuple's timestamp is compared against its source window size.

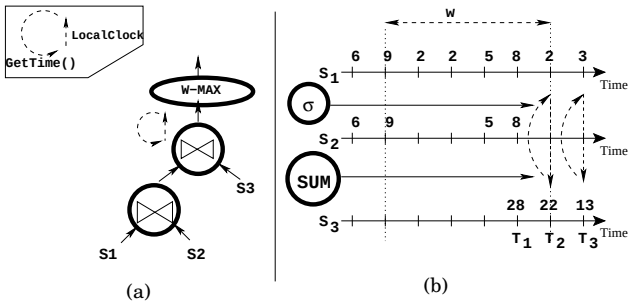


Figure 3. The Time Probing Approach

3.1.1 Implementation

In general, intermediate tuples in TPA (e.g., the output tuples from the window join) needs to maintain more than one timestamp, each from their constituent input tuples. This is important since the expiration condition is always evaluated against a single-window size per timestamp. We refer to the minimum and the maximum timestamps of the set of the tuple's timestamps as $minTS$ and $maxTS$, respectively, (or tuple-order for short). For the special type of sliding window queries that use a *single* window among the input data streams, the intermediate tuple needs to maintain only two timestamps, $minTS$ and $maxTS$, regardless of the number of joined data streams. Every statefull operator in the pipeline stores the value of $maxTS$ corresponding to the last processed (or probed) tuple. We refer to this value as the *LocalClock* of the operator. Furthermore, each statefull operator provides a mechanism to report its LocalClock, when probed by a parent operator in the pipeline. We extend the traditional operator iterator interface (i.e., *Open(), GetNext(), and Close()*), to include a new call-back interface, *GetTime()*, that returns the value of LocalClock. Figure 3(a) illustrates TPA when the windowed MAX operator (W-MAX) is scheduled to verify tuple expiration. W-MAX calls GetTime() on W-MAX's immediate child operators thereby updating W-MAX's LocalClock.

Example. Figure 3(b) gives the execution of TPA for the example of Figure 1(a). Recall that, in the exam-

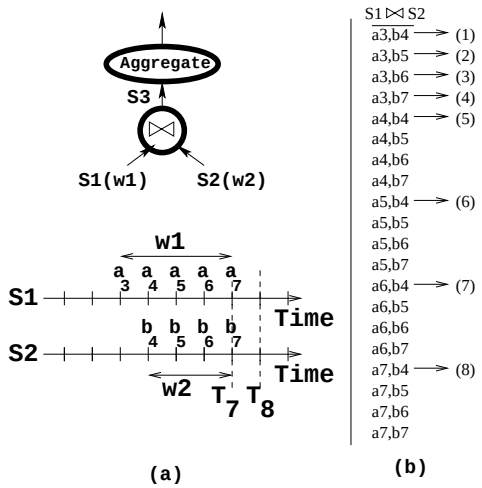


Figure 4. Expiring tuples in sliding window queries with multiple windows

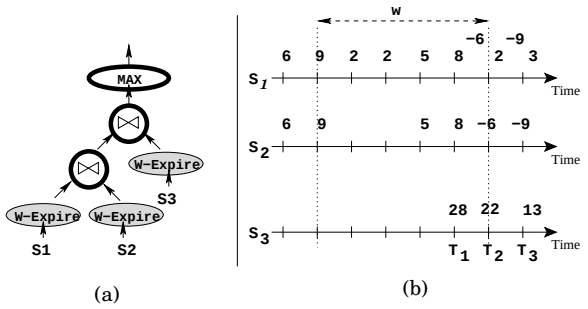


Figure 5. The Negative Tuple Approach

ple, input tuples exist at every clock tick. Then, the Select operator can always update its LocalClock without further probing. At time T_2, the SUM operator does not receive any new tuples and tuple 6 expires. Thus, the SUM operator is scheduled to probe the Select operator asking for the Select operator's LocalClock. The Select operator replies back with the timestamp of the tuple of value 2. Then, the SUM operator recognizes that it has to expire the old tuple with value 6, and hence, update the answer of Q_1 to be 22. Similarly, the answer at T_3 is updated to be 13.

Processing of Invalid Tuples. The processing of invalid tuples (e.g., that could be produced from the MINUS operator) depends on the type of the window operator. In Section 4, we present a classification of the window operators and algorithms for the window operators.

3.1.2 Discussion

One implicit assumption in the design of window-based statefull operators (e.g., window join [12] and window aggregate [9]) is that tuples stored in the operator's state (in-memory buffer) are ordered by their timestamps. In this case, detecting tuple expiration is simplified by examining a small number of tuples at the beginning of the sorted state. This ordering is feasible when considering a sliding window query with a single window. However, for a sliding window query with multiple windows, intermediate tuples may expire in any order. We illustrate this case by the example in Figure 4(a). Stream $S1$ has a window of size 5 time units and Stream $S2$ has a window of size 4 time units. Assume a Cartesian product between $S1$ and $S2$. Figure 4(b) gives the output tuples (similarly, the tuples stored in the Aggregate buffer) at time T_7. Assume that tuples are sorted on the timestamp of $S1$. At time T_8, tuple $a3$ expires from Stream $S1$ while tuple $b4$ expires from Stream $S2$. Since tuples in Figure 4(b) are sorted based on the timestamp of $S1$, few comparisons are sufficient to discover the expired tuples that correspond to $a3$ in Stream $S1$. These tuples are marked (1) to (4) in Figure 4(b). However, to expire tuples that correspond to $b4$ (in Stream $S2$) we need to scan the buffer sequentially since no sorting order is maintained on the timestamp of $S2$ [1]. These tuples are marked (5) to (8) in the figure. Therefore, in contrast to single-window queries, when using TPA for multiple-window queries, the CPU cost for expiring old tuples is proportional to the buffer size (sequential scan), and hence can be costly. In the following section, we propose a scheduling approach that overcomes this drawback.

3.2 Approach II: Negative Tuples

The *Negative Tuple Approach* (NTA) is inspired by the fact that, in general, window operators need to process *invalid tuples* (e.g., see Figure 1(b)). A tuple, say t, that is expired from a sliding window w can be viewed as a *negative tuple* t^- that goes through the pipeline following the footsteps of t. t^- cancels the effect of t in all query operators. *Negative tuples* are another form of *invalid tuples* that are produced by the MINUS operator. NTA unifies the handling of both *invalid* and *negative* tuples. With negative tuples, query operators in NTA can be scheduled using the *Input Triggered* scheduling, i.e., an operator is scheduled only when a tuple exists at its input queue. Notice that in the existence of negative tuples, Input Triggered scheduling

[1] We could have another sorted data structure to speed lookup at the second timestamp. However, this approach is not scalable with the number of joined streams and includes the additional cost of maintaining the sorted data structure

will always produce correct output with no delays.

3.2.1 Implementation

Every tuple in NTA stores a single timestamp, *TS*. Binary query operators (e.g., window join) use the TS of an input tuple to preserve ordered execution among the input data streams. The timestamp of the output tuple is a single timestamp that equals the *maximum* timestamp of the operator's input tuples. Notice that, in contrast to TPA, NTA does not expire a tuple based on the tuple's timestamp. Instead, tuple expiration in NTA depends on the *value*, rather than the timestamp, of the input negative tuple.

Since *negative tuples* are synthetic tuples, we need a new mechanism to generate *negative tuples* for each input stream. We illustrate this mechanism by a conceptual new operator, *W-Expire* (see Figure 5(a)). In Sections 5 and 6, we present efficient implementations of the conceptual *W-Expire* operator. For an incoming tuple t, *W-Expire* performs the following steps: (1) Store t in *W-Expire*'s window structure. (2) Forward t to the parent operator. (3) Produce the *negative tuple* t^- when t is expired due to the sliding window. t^- has the same attributes as those of t and is tagged with a special *flag* that indicates that this tuple is negative. Other query operators should be extended to process *negative tuples*. On the other hand, with NTA, query operators (that can process negative tuples) no longer need the window constraint to guide their execution, e.g., to expire an old tuple.

Example. Figure 5(b) gives the execution of *NTA* for the example in Figure 1(a). At time T_2, the tuple with value 6 expires. Thus, it appears in S_1 as a new tuple with value -6. The tuple -6 passes the selection filter as it follows the footsteps of tuple 6. At time T_2, the SUM operator receives a *negative* input with value 6. Thus, SUM updates its output value to 22. Similarly at time T_3, SUM receives a *negative* tuple with value 9. Thus, the result is updated to 13.

3.2.2 Discussion

A major advantage of *NTA* is that it is very simple to implement. The adaptation of other query operators to handle *negative tuples* properly is needed anyway, to support invalid tuples. Such adaptation is needed even in *TPA*. The simplicity of *NTA* makes it suitable for stream query processing engines. On the other side, an obvious disadvantage of *NTA* is that it doubles the number of tuples that go through the query pipeline [2]. This overhead is more evident when compared to TPA, especially, for single-window queries. For multiple-window queries, TPA incurs significant CPU processing while expiring stored tuples, (i.e., $O(\lambda|w|)$, where λ is the input rate and $|w|$ is the window size). This overhead is negligible in NTA since NTA depends on the value of the negative tuple, in contrast to the timestamp, to expire old tuples. By designing statefull operators that maintain an ordered state based on the tuples' values, the overhead of expiring tuples can always be bounded to a few comparisons. We compare the performance of the Time Probing and Negative Tuple approaches for queries with a single-window and multiple windows in Section 6.

[2]Optimizations that reduce the number of negative tuples floating in a query evaluation pipeline is possible and is a very interesting research area that the authors of this paper address in a separate setup and is beyond the scope of this paper.

4 Classification and Design of Window Operators

In this section, we provide execution models for pipelined window query operators, and present algorithms of the windowed operations.

4.1 Classes of Window Operators

Based on the type of input and output tuples, we distinguish among four cases of window query operators:

- *Case* 1: A *positive* tuple, t^+_{out}, is produced at the output stream as a result of a *positive tuple*, t^+_{in}, being added to the input stream.
- *Case* 2: A *negative* tuple, t^-_{out}, is produced at the output stream as a result of a *positive* tuple, t^+_{in}, being added to the input stream.
- *Case* 3: A *positive* tuple, t^+_{out}, is produced at the output stream as a result of a *negative* tuple, t^-_{in}, being added to the input stream.
- *Case* 4: A *negative* tuple, t^-_{out}, is produced at the output stream as a result of a *negative* tuple, t^-_{in}, being added to the input stream.

Cases 1 and 4 can arise in all window operators. For example, consider the windowed aggregate operator. For Case 1, when this operator receives an input tuple t^+_{in}, a new aggregate value t^+_{out} could be produced. For Case 4, when a negative tuple t^-_{in} becomes an input to the operator, and assuming that t^+_{in} results in an earlier output, then the aggregate tuple t^-_{out} should be produced. t^-_{out} indicates that the corresponding aggregate value is no longer part of the output stream.

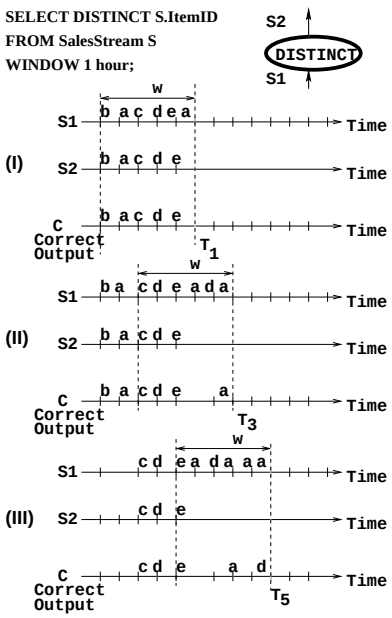


Figure 6. Unexpected answers from the DISTINCT operator

Algorithm 4.1 The W-DISTINCT Algorithm

```
1)  For all expired tuples, t_e, in H
2)      Remove t_e from H
3)      If t_e is found in DL /*t_e was reported as distinct*/
4)          Remove t_e from DL
5)          Probe H using the values of t_e
6)          If a matching tuple is found in H
            /* A duplicate of the expired tuple
               still exists in the current window*/
7)              Add t_e to DL and to the output stream.
8)          Else   If input tuple was t_e^-
9)                     Add t_e^- to the output stream
10)             EndIf
11)         EndIf
12)     EndIf
13)     Delete t_e
14) EndFor
15) If new tuple t_n exists at the input stream
16)     Probe H using the values of t_n
17)     If no matching tuple is found in H
            /* Tuple t_n is distinct */
18)         Add t_n to DL and to the output stream
19)     EndIf
20)     Add t_n to H
21) EndIf
```

Cases 2 and 3 are special to some window operators as will be explained in the following subsection.

4.2 Window DISTINCT (W-DISTINCT)

Problem Description

Conventional approaches for duplicate elimination can produce *incorrect* output when applied to sliding window queries as we illustrate by the following example. Figure 6 gives a sliding window query that contains a windowed DISTINCT operation. $S1$, $S2$, and C represent the input stream, the output stream after the DISTINCT operator, and the correct output from the DISTINCT operator, respectively. $S2$ reports correct answers until time T_1. However, at time T_3, tuple a of Stream $S2$ expires. Since tuple a was one of the distinct tuples of Stream $S2$, the window-output of Stream $S2$ at time T_3 does not reflect the correct distinct values (compared to Stream C). Similarly, at time T_5, tuple d expires from Stream $S2$ and the distinct tuple in $S2$ (a single tuple e) does not reflect the correct distinct tuples at T_5 (the distinct values at time T_5 are the tuples e, a, and d). This incorrect output of the DISTINCT operator results from ignoring the effect of tuple expiration.

The W-DISTINCT Algorithm

Similar to the traditional hash-based duplicate-elimination algorithm [13], the windowed DISTINCT algorithm (W-DISTINCT) compares an input tuple t_n with the set of previously received tuples (stored state). If t_n is distinct, W-DISTINCT inserts t_n in the output stream and adds t_n to the stored state. On the other hand, W-DISTINCT differs from the traditional duplicate-elimination algorithm on the following: (1) The stored state represents the set of tuples in the last window (old tuples are dropped from the window). (2) W-DISTINCT outputs a new tuple t to replace an expired tuple t_e, whenever t_e was produced before as a distinct tuple and t is a duplicate for t_e. Typically, this is Case 3 that is presented in Section 4.1.

W-DISTINCT uses the following two data structures: (1) A hash table, H, to store the distinct tuples in the current Window, and (2) a sorted list, Distinct List (or DL for short), to store all output distinct tuples sorted by their minTS. Given these data structures, Algorithm 4.1 illustrates the steps of the W-DISTINCT operator. Expired tuples in Step 1 of the algorithm are identified either when the timestamp of the tuples is far by more than window from the LocalClock or when an invalid (similarly negative) tuple is received.

Consider the example in Figure 6 while using the proposed W-DISTINCT Algorithm. At time T_3, tuple a of Stream $S2$ expires and the condition in Step 3 of W-DISTINCT is True. Therefore, Steps 4-7 of W-DISTINCT will produce a new output tuple, a, which represents the correct answer as in Stream C. Similarly, W-DISTINCT produces a correct output both at

time T_5 and when tuple d expires.

Analysis of the W-DISTINCT Algorithm

W-DISTINCT features a regulating property for its output rate as we illustrate by the following analysis and in the experimental section.

The window size is defined in time units with length $|w|$. The mean time between tuple arrivals in the input stream follows an exponential distribution with rate λ tuples/second. Therefore, the window size in terms of number of tuples is $\lambda|w|$ tuples, on average. Let n_{key} be the total number of distinct tuples in the input stream. For example, if the stream is a sequence of alphabetical letters, then $n_{key} = 26$. We assume a uniform distribution for the input data streams. As a result, each tuple has equal probability ($\frac{1}{n_{key}}$) to appear as the next input in any data stream. To measure the output rate, we consider a period of execution of time length $|w|$ and calculate the number of distinct tuples, N_d, in this period. The output rate equals the ratio $\frac{N_d}{|w|}$.

For a new tuple, t_d, the probability, $Prob_d$, that t_d is distinct in window size $\lambda|w|$ equals the probability that no tuple with the same value as t_d already exists in the same window. Therefore, $Prob_d$ equals $(1 - \frac{1}{n_{key}})^{\lambda|w|}$. For a set of size n_{key}, the number of tuples that does not belong to window $\lambda|w|$ is $n_{key}(1 - \frac{1}{n_{key}})^{\lambda|w|}$ and the number of distinct tuples that belong to window w is:

$$N_d = n_{key}(1 - (1 - \frac{1}{n_{key}})^{\lambda|w|}) \quad (1)$$

Therefore, the output rate is $\frac{N_d}{|w|}$. From the previous equation, we observe the following: (1) The output rate stabilizes (becomes almost constant) when the input rate increases (e.g., for large λ, $OutputRate = \frac{n_{key}}{|w|}$. Therefore, for a high input rate, W-DISTINCT *regulates* the output rate. This observation supports the traditional optimization of pushing W-DISTINCT down the query pipeline to limit the number of propagating tuples. (2) The output rate decreases as we increase the window size, and vice versa.

4.3 Window Set Operations

The window UNION (W-UNION) operator is straightforward and can be implemented with little modification using traditional UNION operator. However, W-UNION must process input tuples from different sources in-order (increasing maxTS) and expire its old tuples. On the other hand, the window MINUS (W-MINUS) and window INTERSECT (W-INTERSECT) operators are quite involved. W-INTERSECT has the similar Cases of Section 4 to those of W-Group-By (Appendix A), therefore, we choose in this section to present W-MINUS operator only.

Algorithm 4.2 W-MINUS Algorithm

```
1)  For all expired tuples, t_e =< A_e, TO_e >, from H_S or H_R
        /* Expired tuples from different streams are
             processed in their expiration order*/
2)      If t_e ∈ stream S
3)          Remove t_e from H_S and update f_s(t_e) in F_S
4)          Retrieve f_r(t_e) from F_R
5)          If f_s(t_e) > f_r(t_e)
6)              If t_e ∉ O_S
7)                  Remove from O_S tuple t_i with oldest minTS
8)                  Add to the output stream an invalid tuple
                    < A_i, [TO_i.minTS, TO_e.minTS + |w|] >
9)              Else Remove tuple t_e from O_S
10)             EndIf
11)         EndIf
12)     Else If t_e ∈ stream R
13)         Remove t_e from H_R and update f_r(t_e) in F_R
14)         Retrieve f_s(t_e) from F_S
15)         If f_s(t_e) > f_r(t_e)
16)             Retrieve from H_S tuple t_n with the newest maxTS
17)             Add to the O_S and to the output stream a tuple
                < A_n, [TO_n.minTS, TO_e.minTS + |w|] >
18)         EndIF
19)     EndIF
20) EndFor
21) If exists new tuple t_n =< A_n, TO_n > at the input stream of S
22)     Add t_n to H_S and update f_s(t_n) in F_S
23)     Retrieve f_r(t_n) from F_R
24)     If f_s(t_n) > f_r(t_n)
25)         Add t_n to the O_S and to the output stream
26)     EndIf
27) EndIf
28) If exists new tuple t_n =< A_n, TO_n > at the input stream of R
29)     Add t_n to H_R and update f_r(t_n) in F_R
30)     Retrieve f_s(t_n) from F_S
31)     If f_s(t_n) ≥ f_r(t_n)
32)         Remove from O_S tuple t_i with the oldest minTS
33)         Add to the output stream an invalid tuple
            < A_i, [TO_i.minTS, TO_n.maxTS] >
34)     EndIf
35) EndIf
```

The W-MINUS between streams S and R produces in the output stream tuples in S that are not included in R during the last window. Recalling that in the example in Figure 1(b), W-MINUS can produce an invalid output tuple as it receives a new input tuple (i.e., W-MINUS contains *Case* 2). Furthermore, W-MINUS can produce new output tuples when a previously input tuple expires (i.e., W-MINUS contains *Case* 3). To better understand the last case, consider an expired tuple t_e from Stream R that has no duplicates in R. In addition, t_e has a duplicate tuple in Stream S. When tuple t_e expires, the duplicate tuple of t_e in Stream S must be reported as the new output tuple. In addition to *Cases* 2 and 3, W-MINUS also contains *Cases* 1 and 4. Therefore, W-MINUS operator presents all the cases in Section 4.

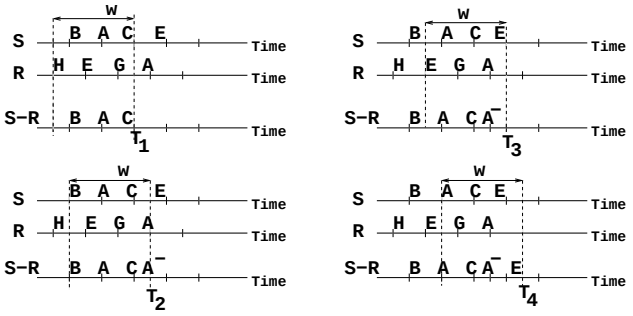


Figure 7. W-MINUS Example.

The proposed W-MINUS Algorithm is duplicate-preserving (i.e., MINUS ALL). The duplicate free version of the operator can be easily implemented by following the W-MINUS with a W-DISTINCT operator. We adopt the SQL definition of duplicate preserving MINUS operator, where duplicates are significant in each stream (e.g., if stream S has n duplicates of tuple a and R stream has m duplicates of tuple a, the output is $max(o, n-m)$ duplicates of tuples a). The Algorithm uses the following data structures:

- Hash tables (H_S and H_R): to store the input tuples from streams S and R, respectively.
- Frequency tables (F_S and F_R): to store the number of occurrences for each distinct tuple in S and R, respectively. We use $f_s(t)$ and $f_r(t)$ to represent the count of duplicates for tuple t in Streams S and R, respectively.
- Output table (O_S): to store the output tuples from Stream S that are not expired or invalidated.

Given these data structures and an input tuples of the form: $< \mathcal{A}, \mathcal{TO} >$, where $\mathcal{A}$ represents the values in the list of attributes and TO is the tuple-order, the Algorithm 4.2 presents the details of W-MINUS operator for sliding window queries with a single window. The extension to for sliding window queries with multiple windows is straightforward. We provide an analysis of the space and time complexity in Appendix C.

Example. The example in Figure 7 is the same as that in Figure 1(b), however, with more input tuples. Up to time T_1, Stream S contains the tuples B, A, and C ($f_s(B) = f_s(A) = f_s(C) = 1$). Since no corresponding tuples for B, A, and C exists in Stream R ($f_r(.) = 0$), the tuples are reported in the output stream $S - R$ (Steps 21-27 of the W-MINUS Algorithm). At time T_2, tuple A arrives in Stream R ($f_r(A) = 1$). Since A also appeared in Stream S and was reported in the output Stream (as part of the difference set), the W-MINUS produces A^- tuple to invalidate tuple A in the output stream (Steps 28-35). At time T_3, tuple E arrives in Stream S. Since tuple E exists in Stream R, the tuple is not reported in the Stream $S - R$. However, at time T_4, tuple E of Stream R expires ($f_r(E) = 0$). At this time, tuple E of Stream S should belong to the difference set. Therefore, at time T_4, tuple E is reported at the output stream (Steps 12-19 of the Algorithm).

Analysis of the W-MINUS Algorithm

The space complexity of the W-MINUS algorithm is determined by the sizes of the hash table and the frequency table for each input stream. In addition the Output table size is included in the space complexity of Stream S. Let λ be the arrival rate for both input streams and let w be the window size w in time units. Then, the size of the hash table per stream is λw tuples. The size of a stream's frequency table depends on the number of distinct tuples in the window and equals N_d, as in Equation 1. The worst case size of the Output table for stream S is the window size λw. Therefore, the total space requirement of the W-MINUS, S_{Minus}, is:

$$S_{Minus} = 2(\lambda w + N_d) + \lambda w$$

Similar to W-DISTINCT, the time complexity of the algorithm depends on the search complexity in the other stream. A tuple from one stream needs to access the frequency table of the other stream (Algorithm 4.2 steps 4,14,23,30) to search for a match. We assume that the frequency table has a hash structure. Therefore, the search complexity depends on the size of the hash bucket. Let the size of the stream's frequency table be N_d (Equation 1). Let H_b be the number of buckets in the hash table. Then, on average, the $BucketSize = \frac{N_d}{H_b}$.

4.4 Window Aggregate and Group-By

Similar to W-DISTINCT, the correct execution for window aggregate (W-Aggregate) and window Group-By (W-Group-By) may produce a new output when a tuple expires (Case 3 in Section 4). This is the case since the aggregate operation represents a function over a set of tuples (e.g., SUM), changing this set (either by expiration or addition) usually invalidates the value of the previous output and produces a new output. We focus in this Section on the W-Group-By operation as W-Aggregate is a special case of W-Group with a single group.

To comply with the measure of correctness in Section 2, the incremental evaluation of W-Group-By must have the following properties:

- W-Group-By must react for every change in the input window contents.
- A group G is no longer part of the current output when all tuples $\in G$ expire (the W-Group-By Algorithm produces a NULL tuple for group G in this case).
- Operators followed by W-Group-By must be able to distinguish from the output stream those tuples that belong to the current W-Group-By result.

To address the last property, W-Group-By assigns the tuple-order for the output tuples such that only the output tuples that belong to current window are part of the result. Furthermore, a basic assumption is that the latest output value for a group *overrides* any previous value for the same group.

We present the W-Group-By Algorithm while considering a general execution framework that can support any aggregate function (e.g., SUM, COUNT, MEDIAN ...etc). Input tuples has the form $< \mathcal{G}, \mathcal{A}, TO >$, where $\mathcal{G}$ represents the values of the group attributes, $\mathcal{A}$ represent the values of the aggregate attributes (attribute a_i belongs to $\mathcal{A}$ if a_i appears in the $AggrFn_1(a_1)\ldots AggrFn_n(a_n)$ list of the SQL SELECT clause), and TO is the tuple-order of the input tuple. For simplicity we use $\mathcal{F}$ to represent the aggregate functions $AggrFn_1(.),\ldots, AggrFn_n(.)$. We also use $\mathcal{V}$ to refer to set of results after applying function $\mathcal{F}$ on the group tuples. The W-Group-By algorithm uses the following data structures:

- GroupHandle, $\mathcal{GH}$ (one for each group): stores the state of the current group such as current aggregate values and the smallest minTS among all tuples in the group($\mathcal{GH}.minTS$).
- Hash table, H: stores all tuples in the current window hashed by values in their grouping attributes. An entry in H stores the tuple and the corresponding $\mathcal{GH}$ and has the form: $(< \mathcal{G}, \mathcal{A}, \mathcal{TO} >, \mathcal{GH})$.

Algorithm 4.3 presents the details of the W-Group-By operation for sliding window queries with a single window. The extension to for sliding window queries with multiple windows is straightforward. Steps 2 to 11 handle tuple expiration, Steps 5 and 6 produce a new output if the expired tuple belongs to a group that still contains non-expired tuples. Steps 8 and 9 produce a NULL value for the empty group. New tuple is handled by Steps 14 to 21 of the Algorithm. Step 17 creates new group for new tuples that does not belong to any of the current groups. Steps 19 to 21 compute a new aggregate value and produce anew group value. The tuple-order of the output tuples is assigned at either Step 8 or Step 21 of the Algorithm.

Algorithm 4.3 W-Group-By Algorithm

1) For all expired tuples, $(< \mathcal{G}_e, \mathcal{A}_e, TO_e >, \mathcal{GH}_e)$, in H
2) Remove $(< \mathcal{G}_e, \mathcal{A}_e, TO_e >, \mathcal{GH}_e)$ from H
3) Probe H with values in $\mathcal{G}_e$
4) If found a matching entry $(< \mathcal{G}_e, \mathcal{A}_h, TO_h >, \mathcal{GH}_e)$
`/* The group still has non-expired tuples and should report new aggregate values */`
5) Apply $\mathcal{F}_e$ for tuples in group $\mathcal{G}_e$ to get $\mathcal{V}$.
6) Add $< \mathcal{G}_e, \mathcal{V}, [\mathcal{GH}_e.minTS, TO_e.minTS + |w|] >$ to the output stream
7) Else `/* The Group expires */`
8) Add $< \mathcal{G}_e, NULL, TO_e.minTS + |w| >$ to the output stream
9) Delete $\mathcal{GH}_e$
10) EndIf
11) Delete $(< \mathcal{G}_e, \mathcal{A}_e, TO_e >, \mathcal{GH}_e)$
12) EndFor
13) If exists new tuple $< \mathcal{G}_n, \mathcal{A}_n, TO_n >$ at the input stream
14) Probe H with values in $\mathcal{G}_n$
15) If not found a matching entry, $(< \mathcal{G}_n, \mathcal{A}_h, TO_h >, \mathcal{GH}_n)$
`/* Tuple` $< \mathcal{G}_n, \mathcal{A}_n, TO_n >$ `forms a new group */`
16) Create $\mathcal{GH}_n$ for $\mathcal{G}_n$
17) EndIf
18) Add $(< \mathcal{G}_n, \mathcal{A}_n, TO_n >, \mathcal{GH}_n)$ to H
19) Apply $\mathcal{F}_n$ for tuples in group $\mathcal{G}_n$ to get $\mathcal{V}$.
20) Add $< \mathcal{G}, \mathcal{V}, [\mathcal{GH}_e.minTS, TO_n.max] >$ to the output stream
22) EndIfd

4.5 Window Join (W-Join)

Typically, the binary join iterates over all tuples in one input source (the outer data source) and for each outer tuple retrieves all matching tuples from the inner data source. For joining data streams, a symmetric evaluation is more appropriate than the fixed-outer evaluation since both sides of the join can act as outer to perform the join. The extension of the symmetric approach for W-Joins over data streams is presented in [14, 12, 18].

According to the cases in Section 4, W-Join needs to address *Cases* 1 and 4. W-Join never produces a new output as an input tuple expires or produces an invalid output. W-Join needs to process tuples in increasing maxTS and assigns tuple-order for its output tuples as follows: The minTS equals the minimum value of minTS for all joined tuples. The maxTS equals the maximum value of maxTS for all joined tuples.

We illustrate the symmetric evaluation of W-Join in

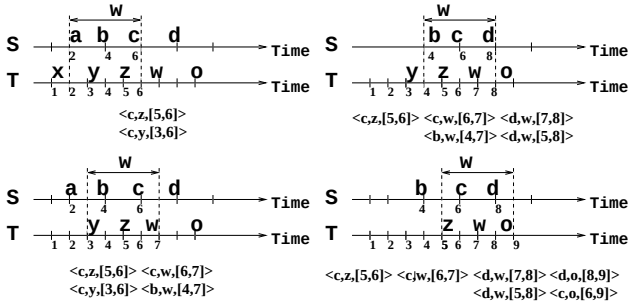


Figure 8. W-Join.

Figure 8 assuming a single sliding window of size $five$ clock ticks. The extension to for sliding window queries with multiple windows is straightforward.. The output tuples are presented at each execution times. The W-Join execution at time 8 starts at top digram in the second column of Figure 8.

4.6 Processing of Invalid and Negative Tuple

To appropriately handle *negative* tuples, all pipelined query operators need to be equipped with special procedures. In this section, we present the modifications of all window operators to support *negative* tuples in a pipelined query plan. As it is a major advantage in the *Negative Tuple* approach, the required modification are simple and easy to implement.

W-DISTINCT operator. Upon receiving a *negative* tuple t^-, we distinguish between two cases: (1) t^+ was reported as distinct in the output stream. Therefore, t^- must be reported as *negative* tuple in the output stream. In addition, t^- may generate a positive output similar to the case when expiring an old tuple in Algorithm 4.1(Steps 1-11). (2) t^+ was not reported as distinct in the output stream and therefore, no need to report t^- in the output stream.

W-MINUS operator. Processing an incoming *negative* tuple t^- is similar to that of processing an expired tuple (Steps 3-19 of Algorithm 4.2). Only one slight addition need to be considered, which is the case when t^- appears in S and was reported before as output (e.g., $t \in O_S$). In this case, we produce t^- in the output stream.

Project (with duplicates) and Select operators. A *negative* tuple t^- is processed in the same way as a *positive* tuple t^+ (e.g., project out some attributes or apply the selection predicate).

W-Group-By and W-Aggregate operators. Processing a *negative* tuple includes two steps: (1) Removing tuple t^+ from the stored state of the operator, i.e., the set of tuples within the window. (2) Generating a new output value that represents the new aggregate value over the new stored state.

W-Join operator. After removing the corresponding tuple t^+ from the stored state, t^- from one stream joins with matching tuples in the other stream and produces *negative* joined tuples (if any).

5 Prototype Implementation

To study the performance of the proposed approaches and algorithms, we implemented TPA and NTA inside a prototype stream query processing engine, Nile [16]. Streaming is introduced using an abstract data type *stream-type* that can present source data types with streaming capabilities. To connect a query execution plan with an underlying stream, we use the *StreamScan operator* to communicate with the stream table and retrieve new tuples. The operators communicate with each other through a network of FIFO queues. We implement a scheduler to schedule operators according to the requirements of TPA and NTA. We implement the W-DISTINCT algorithm presented , the W-Set, W-Aggregate, W-Group-By, and W-Join algorithms in Section 4. Invalid tuples are tagged with special flags to distinguish them from input tuples (negative tuples are tagged similarly).

We implement the W-Expire operator as part of the StreamScan operator, which is scheduled upon the arrival of an input tuple. The query execution plan is constructed using multi-level binary join operations on the streams and relations in the FROM clause. The Aggregate, Group-By and DISTINCT operators are added as separate operators. The window specifications are added as special constructs for the query syntax as given in Q_1 and Q_5 of Figures 9.

6 Experiments

In this section, we compare the performance of the proposed Time Probing Approach (TPA) and the proposed Negative Tuple Approach (NTA) against a *modified* version of the Input Triggered Approach (ITA for short). In this modified version, we maintain the correctness by producing delayed output tuples (though delayed) when a tuple is received by the operator. In addition, ITA uses the same set of operators as in TPA, however, without probing.

Our measures of performance are the average and maximum of the output response time and the output throughput. To show the performance of the proposed approaches on various window operators while using a single window and multiple windows, we con-

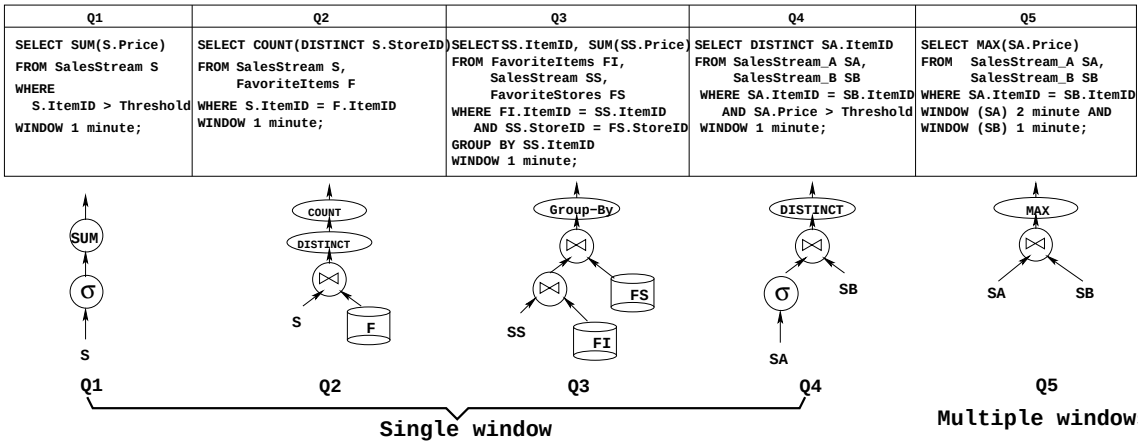


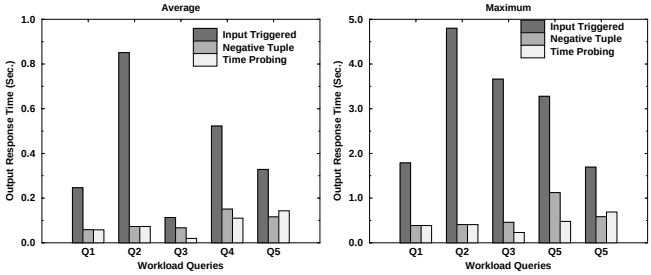
Figure 9. Workload queries and their execution plans

sider a workload of five different queries given in Figure 9. The SalesStream in the queries has the following schema: (StoreID, ItemID, Price, Quantity, Tuple-order), where StoreID identifies the retail store, ItemID is the sold item identifier, Price and Quantity are information about the sold item. Tuple-order is as described in Section 3. In addition, we use two relational tables that store favorite items (FavoriteItems) and favorite stores (FavoriteStores). The schema for FavoriteItems and FavoriteStores tables is a single attribute (primary key) for ItemsID and StoreID, respectively. Unless mentioned otherwise, the predicate selectivity for Q_1 and Q_4 are set to 0.25. The window join selectivity in Q_4 is 0.6 (the overall selectivity in Q_4 is $\sim$ 0.1) and the join selectivity in Q_2, Q_3, and Q_5 is 0.1. All the experiments are run on Intel Pentium 4 CPU 2.4 GHz with 512 MB RAM running Windows XP. We use synthetic data streams where the inter-arrival time between two data items follows the exponential distribution with mean λ tuples/second. In Section 6.2, we use real data streams that represent online transactions from Wal*Mart stores [3]. The arrival rate of the real data streams is bursty with average 1 tuple/second (during peak sales time).

6.1 Summary of Workload Queries

Figure 10 gives the output response time for the five workload queries (Figure 9) when scheduled using ITA, TPA, and NTA. We use synthetic input streams with an average arrival rate of 20 tuples/second. In all queries, ITA incurs significant delays (0.85 seconds on average and 4.8 seconds maximum). For sliding window queries with *a single-window* (e.g., Q_1 to Q_4),

[3]Data was supplied to Purdue University by Wal*Mart and NCR Corporations



(a) (b)

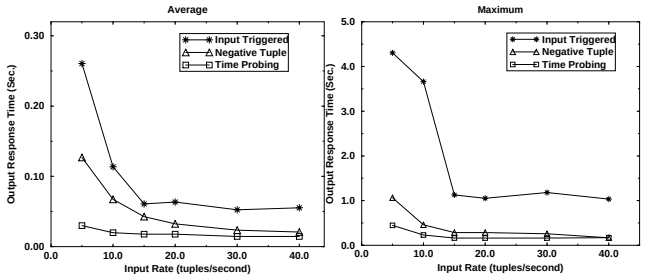
Figure 10. Summary for workload queries

TPA provides the smallest output response time, followed by NTA. For Q_3 and Q_4, NTA has higher output response time compared with that of TPA. The reason is that both queries include expensive operators (e.g., a join followed by a Group-By in Q_3 and W-Join in Q_4). Processing both new and negative tuples by such expensive operators increases processing time and output response time. For Q_5, which uses *multiple windows*, *NTA has the smallest* output response time. This is the result of the increasing overhead of scanning the buffer of the MAX operator by ITA and TPA compared to NTA. In the following sections, we present a set of experiment on sliding window queries with a single window followed by a second set of experiments on sliding window queries with multiple windows.

6.2 Single-window Queries

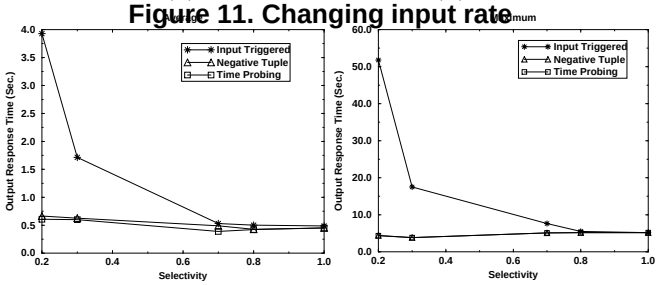
Variable input rate

Figure 11 gives the performance of the scheduling approaches for the range of arrival rates between 5 and 40 tuples/second and when applied for Q_3 (Q_1, Q_2, and Q_4 give similar performance measures as Q_3). TPA has better performance followed by NTA and then ITA.



(a) (b)

Figure 11. Changing input rate



(a) (b)

Figure 12. Changing selectivity

The reason is the same as described in Section 6.1. The performance of all approaches converges as we increase input arrival rates. This behavior is expected. Higher input rates produce more tuples that propagate up the pipeline, hence, refreshing the stored state of window operators and producing output tuples with shorter delays. However, ITA still provides a higher output response time compared to the other approaches. The main reason is that ITA is greatly constrained by the underlying operator selectivity. We repeated all the experiments using the Wal*Mart real data streams and obtained comparable results and trends to those of synthetic data. For space limitations we omit these results.

Changing Selectivity

In this experiment, we use the *real data streams* from Wal*Mart stores. Figure 12(a) and 12(b) give the effect of changing the selectivity (from 0.2 to 1) on the average and maximum output response times, respectively, when using the various scheduling approaches. We present only the results of the experiment for Q_1 since similar performance is obtained from the other queries. We set the single-window size to 10 minutes. With a low selectivity of 0.2 (i.e., high filtering), ITA has increased response time (4 seconds on average and maximum of approximately one minute). TPA and NTA have significant improvement (an order of magnitude decrease in average response time). With the increase in selectivity, all the scheduling approaches have low output response time. This is a result of having more tuples through the pipeline. With selectivity 1.0 (i.e., no filtration), all the scheduling approaches have

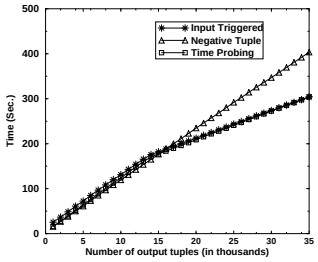


Figure 13. Output Throughput

the same performance. This indicates that for simple queries that do not have any filtration, ITA can be a candidate for scheduling. Other scheduling approaches have a slight increase in the output response time with the increase in selectivity. This is mainly due to the additional processing overhead incurred by TPA and NTA.

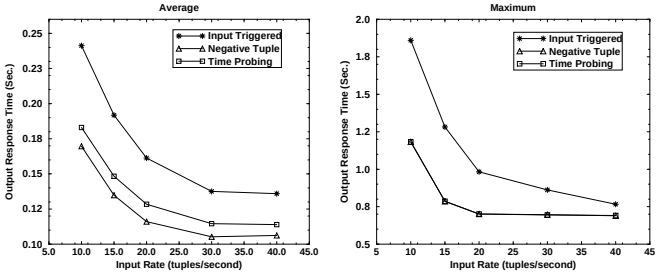
High Input Rate

In this experiment, we measure the execution speed of the various scheduling approaches. We run the workload queries using a very high input rate (more than the maximum capacity of the available system resources). Then, we measure the number of tuples produced by each approach at time advances. Figures 13 gives the execution time needed by the scheduling approach to output up to 40K tuples for Q_4. We use synthetic data streams with an arrival rate of 2056 tuples/second. ITA and TPA provide comparable execution times followed by NTA, since at high data rates no delays are expected at the output. Therefore, TPA is scheduled similar to ITA (no probes) and produces similar performance.

6.3 Multiple-Window Queries

Variable Input Rate

Figure 14(a) gives the average and maximum output



(a) (b)

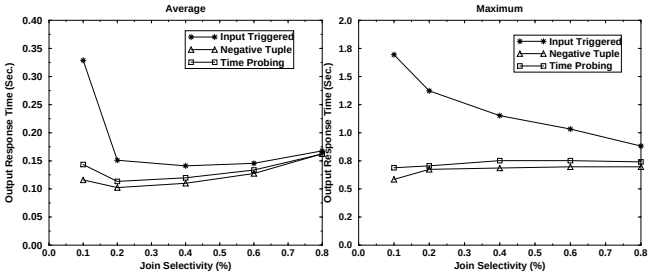
Figure 14. Changing input rate

response times of the scheduling approaches for the range of input rates between 10 and 40 tuples/second and when applied for Q_5. The join selectivity is set to

0.2. *NTA* has the best performance, followed by TPA, and then ITA. This graph supports our conclusion in Section 3.2.2, where for multiple windows, NTA outperforms both TPA and ITA. The reason is that TPA and ITA use sequential scanning for expiring old tuples from the intermediate buffer of the MAX operator. Alternatively, NTA uses hashing based on the tuple's value to identify expired tuples. For high input rate, the performance degrades in all approaches. However, ITA has the highest response time since the buffers (in the join and the MAX operators) increase in size due to the infrequent refreshing. NTA has better performance than TPA due to the fast expiration in the case of NTA. Similar to the case of average response time, Figure 14(b) illustrates a similar trend for the case of the *Maximum* response time for all three approaches. However, NTA and TPA incur almost the same maximum response times since we assume that the system can keep up with the arrival rate and that no indefinite postponement is expected.

Changing Selectivity

Figure 15 gives the performance of the scheduling ap-



(a) (b)

Figure 15. Changing selectivity

proaches for the range of join selectivity between 0.1 and 0.8, when applied for Q_5. *NTA* has the best performance, followed by TPA, and then ITA. The performance converges for all approaches (similar to the single-window case) as we increase selectivity, which increases the input rate to the intermediate buffer. This behavior is expected since higher input rates produce more tuples to propagate up the pipeline, hence refreshing the stored state of the window operators and producing output tuples with lower response time. However, the output response times increase in all approaches for high selectivity since the processing times of tuples increase (more tuples to compare with in the intermediate buffer of the MAX operator).

High Input Rate

In this experiment we use Q_5 and apply the same settings as for the corresponding single window experiment in Section 6.2. As illustrated in Figure 16, NTA has the best throughput *for high data rates* followed by ITA and TPA. TPA and ITA have the same perfor-

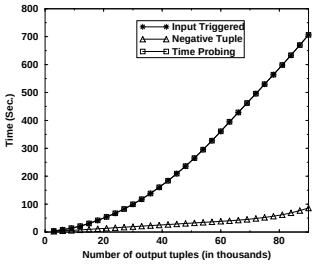


Figure 16. Output Throughput

mance since TPA performs almost no probing. As time advances, all approaches start to encounter higher delays because of the size increase in the inter-operator queues. However, the rate of increase is much lower for the case of NTA due to the efficient expiration approach (no sequential scanning).

6.4 Memory Requirement - Discussion

In terms of memory requirements, TPA and ITA use lower memory during execution compared to NTA. However, NTA has many desirable properties in terms of memory usage such as the following: First, for aggregate functions such as SUM, COUNT, and Average, the amount of stored state can be significantly reduced. Instead of storing a *whole* window of tuples in the case of TPA and ITA, it reduces to storing a *single* value (two values for the average) in the case of NTA, since the positive/negative tuples can be used to increment/decrement the stored values. Second, the W-Expire operator can be integrated as part of the join buffers in W-Join queries. Therefore, W-Expire will incur no additional memory. Furthermore, to optimize memory usage by the operators' *queues*, our proposed scheduling approaches can utilize the techniques proposed in [1, 4].

7 Related Work

In this section, we discuss the related work in the areas of sequence databases, temporal databases, and continuous query evaluation of streams and append-only relations.

Sequence databases and temporal databases are well-studied areas of research in the database literature. Seshadri et al. [22] present the SEQ model and its implementation for sequence databases. In this work, a sequence is defined as a set with a mapping function to an ordered domain. Jagadish et al. [17] provide a data model for chronicles (i.e., sequences) of data items and discuss the complexity of executing a view described by the relational algebra operators. The focus of both these efforts was on *stored* time-ordered

data rather than on the pipelined processing of live data streams. Snodgrass [23] addresses handling time in traditional databases. His seminal work includes SQL formulation to evaluate complex predicates and joins over the time attributes. Temporal join [26] and Band-Join [10] are join operators that use a distance-guided predicate (similar to window join). Push-based execution of query operators as execution threads connected by queues is listed by Graefe [13] as one design alternative that is followed by traditional database systems. Duplicate-elimination and the effect of early DISTINCT operators on reducing processing work is addressed in [7]. Early work on extending database systems to process Continuous Queries is presented in Tapestry [24] that investigated the incremental evaluation of queries over append-only databases.

Stream query processing is being addressed currently in a number of prototype systems, e.g., Aurora [3], Gigascope [8], Nile [16], STREAM [21], and Telegraph [5]. These projects have recognized the need for sliding windows in order to make queries over data streams practical. However, to date, these systems have not detailed how they address the problems resulting from pipelined sliding-window queries over data streams. Thus, our work is largely complementary to these other projects. Finally, work on punctuating data streams [25] is related to NTA. However, punctuations have been used to delineate among groups of tuples rather than to refer to a single tuple as in NTA.

8 Conclusions

We have described a correctness measure for the pipelined execution of sliding window queries. We proposed two scheduling approaches to guarantee the correct execution. The Time Probing Approach (TPA) synchronizes the local clock of each operator based on the most recent processed or probed tuple. The Negative Tuple Approach (NTA) uses the new idea of propagating special tuples (negative tuples) to undo the effect of the expired tuples. TPA gives the best performance for sliding-window queries with a single-window for low as well as high input rates. NTA gives the best performance for sliding window queries with multiple windows for low as well as high input rates. Between the two proposed approaches, the Negative Tuple Approach was the simplest to implement. We described the various combinations of input and output tuples using the positive-negative tuple paradigm. This classification helps in identifying various classes of sliding-window operators. We presented one example of an incremental algorithm for the window DISTINCT operator. We performed experiments based on an implementation of the two proposed approaches and algorithms in Nile, a prototype stream query processing engine. The results show that the proposed scheduling algorithms provide more than an order of magnitude reduction in output response time when compared to the Input Triggered Approach. Remarkably, this performance is achieved for input data streams with low arrival rates.

References

[1] B. Babcock, S. Babu, , M. Datar, and et al. Chain: Operator scheduling for memory minimization in stream systems. In *SIGMOD, June*, 2003.

[2] S. Babu, R. Motwani, K. Munagala, and et al. Adaptive ordering of pipelined stream filters. In *SIGMOD, Jun.*, 2004.

[3] D. Carney, U. Cetintemel, M. Cherniack, and et al. Monitoring streams - a new class of data management applications. In *VLDB, Aug.*, 2002.

[4] D. Carney, U. Cetintemel, A. Rasin, and et al. Operator scheduling in a data stream manager. In *VLDB, Sep.*, 2003.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, and et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR, Jan.*, 2003.

[6] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB, Aug.*, 2002.

[7] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB, Sep.*, 1994.

[8] C. D. Cranor, T. Johnson, O. Spatscheck, and et al. Gigascope: A stream database for network applications. In *SIGMOD, June*, 2003.

[9] M. Datar, A. Gionis, P. Indyk, and et al. Maintaining stream statistics over sliding windows. In *ACM-SIAM Symposium on Discrete Algorithms*, 2002.

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

[11] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD, June*, 2001.

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

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

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

[15] M. A. Hammad, T. Ghanem, W. Aref, and et al. Efficient pipelined execution of sliding window queries over data streams. In *http://www.cs.purdue.edu/homes/mhammad/TR.pdf. Technical Report CSD TR#02-010, June*, 2004.

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

[17] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *PODS, May*, 1995.

[18] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE, Feb.*, 2003.

[19] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE, Feb.*, 2002.

[20] S. Madden, M. J. Franklin, J. M. Hellerstein, and et al. The design of an acquisitional query processor for sensor networks. In *SIGMOD, June*, 2003.

[21] R. Motwani, J. Widom, A. Arasu, and et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR, Jan.*, 2003.

[22] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB, Sep.*, 1996.

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

[24] D. Terry, D. Goldberg, D. Nichols, and et al. Continuous queries over append-only databases. In *SIGMOD, June*, 1992.

[25] P. A. Tucker, D. Maier, T. Sheard, and et al. Exploiting punctuation semantics in continuous data streams. In *TKDE, 15(3):555-568, May*, 2003.

[26] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *ICDE, Feb.*, 2002.