Efficient Execution of Sliding-Window Queries Over Data Streams*

Moustafa A. Hammad
Purdue University
mhammad@cs.purdue.edu

Walid G. Aref
Purdue University
aref@cs.purdue.edu

Michael J. Franklin
University of California at Berkeley
franklin@cs.berkeley.edu

Mohamed F. Mokbel
Purdue University
mokbel@cs.purdue.edu

Ahmed K. Elmagarmid
Purdue University
ake@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 first show how the straightforward application of traditional pipelined query processing techniques to sliding window queries can result in inefficient and incorrect behavior. We then present three alternative execution techniques that guarantee correct behavior for pipelined sliding window queries and develop new algorithms for correctly evaluating window-based duplicate-elimination, Group-By and Set operators in this context. We implemented all of these techniques in a prototype data stream system and report the results of a detailed performance study of the system.

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 an incremental processing of input data streams. Combined with various types of non-blocking query operators, this incremental processing results in a system that can continuously provide query answers on-the-fly, even when the input streams are effectively never-ending.

A number of recent research efforts have introduced algorithms for windowed versions of one or more relational operators [3, 6, 9, 20, 22]. Current techniques, however, are limited in two crucial aspects: First, algorithms have been proposed for only a few query operators such as window join [6, 14] and various windowed aggregates [10, 13, 24]. Second, the focus has been on the execution of individual operators, and thus, the interaction of multiple operators in a pipelined *query plan* has largely been ignored.

1.1 Considering Query Plans

In this paper, we examine this larger context of windowed-query processing over data streams. When considering full query plans (i.e., pipelines of operators) there are several key factors that need to be addressed:

- **How time progresses** - In a query plan, the operator that ultimately generates the output (i.e., at the top of the plan) may be separated from the operators that initially receive the input data streams (i.e., at the bottom of the plan) by a number of intermediate operators. The propagation (or lack thereof) of input events through the plan can impact the windowed behavior of the output.
- **Delays in the pipeline** - Differences in scheduling or required work at various operators and subtrees will cause different parts of the plan to make progress at different rates, thereby complicating the progression of windows through time.
- **Actions to take when tuples arrive** - When operators are arranged in a pipeline, the correct execution of a given operator depends on the correct execution of its children. Thus, operators must correctly communicate events to their parents. Some of the required actions may be non-obvious. For example, when a MINUS operator receives a new tuple on its right-hand input, it may need to emit a "negative tuple" indicating to its parents that a tuple that was formerly emitted is no longer valid.
- **Actions to take when tuples expire** - Likewise, when tuples expire from a window, actions may be required to inform parent operators of this event. For example, when a tuple expires from a DISTINCT operator, a new tuple may need to be propagated to its parents.

Many window variants have been proposed. One consideration is the unit in which the windows are expressed. There are two basic approaches: *time-oriented* windows are defined using some notion of a clock that ticks independently

*Submitted for publication, Nov., 2003. The paper also appears on the Department of Computer Sciences, Purdue University, technical report CSD TR#03-035, Dec., 2003.

of tuple arrivals (e.g., $w = 1$ minute), while *tuple-oriented* windows are defined based on tuple counts (e.g., $w = 50$ tuples). A second consideration is the way in which the bounds of the window move. Alternatives include *landmark* windows [13], where one end of the window stays fixed while the other moves, *sliding* windows, where the window size remains fixed and the two window boundaries move smoothly in unison one element or time unit at-a-time, and *hopping* or *tumbling* windows where the window size remains fixed but the window boundaries can move in a discontinuous fashion. Each type of window has its own semantics and implementation issues. For concreteness, in this paper we focus on one particular (and we believe, common) window type, namely, *sliding windows* that are based on *time*.

1.2 Contributions and Overview

The contributions of the 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.

2. We propose three approaches for incrementally handling the pipelined execution of window query operators: *Time Probing*, *Negative Tuples*, and *Hybrid*. The notion of negative tuples is novel, and we believe that efficient implementations of this notion could have widespread applicability in data stream query processing engines.

3. In order to support complete query plans, we describe new algorithms for the windowed DISTINCT, windowed Group-by, and windowed set operators.

4. We implement these techniques and algorithms in a prototype stream database system.

5. We describe an extensive set of experiments that compare the proposed approaches using the prototype.

The rest of the paper is organized as follows. Section 2 describes the execution framework while identifying and motivating the need for sliding-window stream query processing. In Section 3, we introduce the three approaches for dealing with the pipelined execution of several window operators. Section 4 introduces the correctness measure and describes algorithms for a variety of window operators. An implementation prototype of the proposed algorithms and the three approaches are discussed in Section 5. Section 6 presents an extensive list of experiments that show the effect of the various approaches. Related work in stream query processing is discussed in Section 7. Section 8 contains concluding remarks.

2. PRELIMINARIES

2.1 Context and Environment

In this paper, we consider a centralized architecture for stream query processing in which data streams continuously arrive to be processed against a set of standing continuous queries. Streams are considered to be unbounded sequences of data items, which are time-stamped with the value of the current system clock when each data item arrives at the system. In some applications, data items may arrive with their own timestamp and in any order (e.g., out of order). Stream processing systems such as Aurora [1] propose the use of a slack interval to account for the out of order arrival and produce ordered input stream. Once the input is in order, our proposed approaches can be used for these applications.

As stated in the Introduction, our focus in this paper is on sliding window queries defined in terms of time units. Our methods support Window Queries (WQ) containing a single such window that is applied across all input streams. Our methods also support queries combining such stream data with regular (i.e., non-streamed) relations.

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 ordered by some timestamp, the output tuples also appear as a stream ordered by a timestamp. 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 MAX operation) depends on receiving an ordered arrival of input tuples. Otherwise, we may expire an old tuple early (e.g., potentially report an erroneous sequence of maximum values). We will elaborate on this point further in the next section. (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.

A single WQ 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 [19], Aurora [3] and STREAM [22].

2.2 Correctness and Measures of Performance

To judge the correct execution of sliding window queries, we first recall that a change in the sliding time window, w, at an input stream may involve adding new tuples or expiring old tuples, or both. A correct execution of a WQ with a window w 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 clause removed. (2) The input to Q is the set of input tuples that arrived during the time interval $T - |w|$ and T. Similar notions of correctness have been proposed in other systems, e.g., [26, 22].

For WQ whose output corresponds to queries that produce a single value for each execution (e.g., aggregate queries) the output in WQ is the most recent tuple in the output stream [1]. For WQ whose output corresponds to queries that produce multiple tuples for each execution (e.g., join, Group-By, DISTINCT, and Set queries) the output of WQ accumulates at the output stream over a period of ex-

[1]In our system if an aggregate value expires and no input tuples exist in the aggregate window, a NULL tuple is produced.

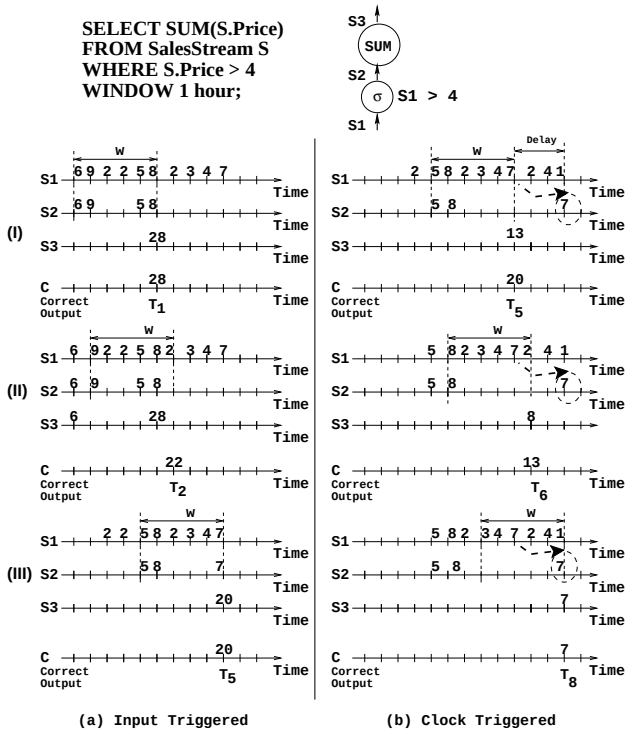


Figure 1: Q_1.

ecution time that is determined by the window size. Therefore, the output of WQ at time T is the set of all the recently produced (or to be produced) tuples in the output stream with timestamp between $T - |w|$ and T. We refer to the output in this case as the *the window-output at* T.

Note that, in an ideal case, the answers for WQ for a time T would be available instantaneously. We refer to T in this case as the *output create-time,* T_{create}. However, real systems produce the output with some delay. We refer to the output time in this case as the *output release time,* $T_{release}$. An important objective of any WQ execution approach is to minimize the *output delay* (i.e., $T_{release} - T_{create}$).

2.3 Motivating Example: Retail Transactions

Consider a retail company with tens of stores in a single geographic region. In a local headquarters of the company; the stream of sales transactions from the individual stores is collected over time for the purpose of on-line monitoring and analysis. A stream database system processes the stores' transactions to control real-time inventory, monitor on-line transactions, and recommend on-line discount policies. This scenario is modeled as multiple data streams, where each stream (SalesStream) represents a group of related stores. The schema of the transaction stream has the form (StoreID, ItemID, Price, Quantity, Timestamp), where StoreID identifies the retail store, ItemID is the sold item identifier, Price and Quantity are information about the sold item. TimeStamp indicates the arrival time of the tuple; timestamps are described in greater detail in the subsequent section.

Problems with basic scheduling mechanisms

Q_1: For SalesStream S, continuously report the total sales of a set of items with prices greater than 4 in the last hour.

Figure 1 gives the SQL representation and pipelined execution plan of Q_1. Two straightforward scheduling approaches can be utilized to schedule the execution of the operators in Q_1: *Input Triggered* and *Clock Triggered*. In the Input Triggered approach, operators are scheduled only when they have input to process. This corresponds to traditional push-based scheduling of pipelined query plans. In contrast, in the *Clock Triggered* approach, operator is scheduled to execute based on a regular clock tick.

Figure 1(a) illustrates the Input Triggered approach for Q_1. $S1, S2$, and $S3$ represent the input stream, the output stream after the selection 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. We present the streams at times T_1, T_2, and T_5 in parts (I), (II), and (III) of Figure 1(a). 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.

On the other hand, in the *Clock Triggered* approach, each operator is scheduled at every time tick. While this approach provides the expected output with no delays, the practical implementation of this approach may produce *nondeterministic* results (i.e., results that depend on delays introduced by the system). Figure 1(b) illustrates the Clock Triggered approach for Q_1. 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 $S2$[2]. 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 applying the Clock Triggered approach, 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 values of 20 and 13 (in Stream C at time T_5 and T_6, respectively) never appear in Stream $S3$. Thus, the Clock Triggered approach would result in a *nondeterministic output.*

Invalidation on tuple arrival.

Q_2: For each sold item in SalesStream S and not in SalesStream R, continuously report the maximum sold quantity for the last hour.

Figure 2(a) gives the SQL representation and pipelined

[2]Such delays are likely to occur as each operator is scheduled independently and tuples incur different processing speeds by different operators.

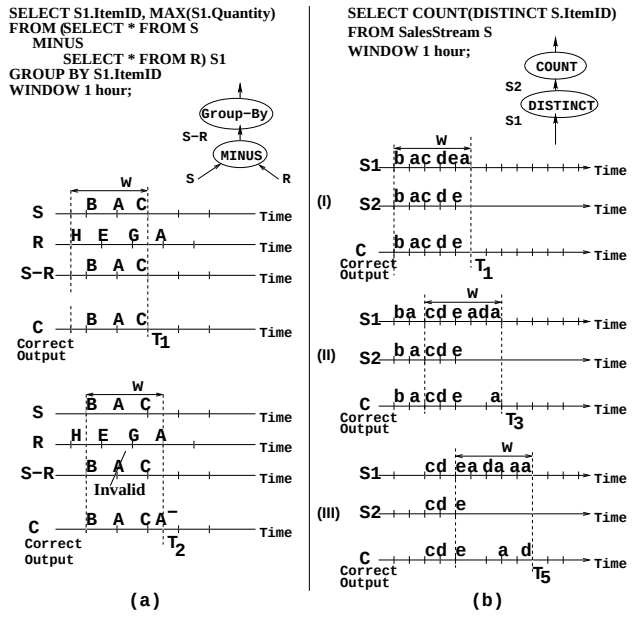


Figure 2: (a)Q_2 and (b)Q_3.

execution plan for Q_2. S and R represents 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. However, at T_1, A is added to R and therefore, A in no longer a valid output in $S - R$. Notice that A was still within the current window when it became *invalid*. In this case, the MINUS operator needs to invalidate a previously reported output tuple. It can do so by generating an *invalid* output tuple. We represent the invalid output tuple as A^- in the correct output of Stream C at T_2. A^- removes any effect of the previously output A in Stream C.

Note that in this scheme parent operators of the 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 tuples*.

Answer generation on tuple expiration

Q_3: For SalesStream S, continuously count the number of distinct items sold in the last hour.

Figure 2(b) gives the SQL representation and pipelined execution plan for Q_3. $S1$, $S2$, 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 T_1. However, at T_3, tuple a in Stream $S2$ expires. Since tuple a was one of the distinct tuples in Stream $S2$, the window-output at Stream $S2$ in 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 a) 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 is mainly due to the fact that the operator ignores the effect of tuple expiration. As the window slides, the input tuples in $S1$ and their corresponding output tuples in $S2$ are expired. While the expired output tuples are still duplicates in the current window, $S2$ provides no new output tuples to replace them. Thus, the output of DISTINCT becomes erroneous.

Discussion. The incremental evaluation of Q_1, Q_2 and Q_3 illustrate two major findings: (1) Straightforward scheduling approaches, e.g., the input- and Clock Triggered approaches may either skip output tuples, produce output tuples after long delays, or produce incorrect output. (2) The intuitive semantics of some window operators, e.g., window MINUS and window DISTINCT, turn out to produce incorrect results. To the best of our knowledge, none of the previous stream processing system address both of these issues.

We have developed our stream query processor with the following objectives:

1. To provide scheduling approaches for pipelined window operators that adhere to the correctness measure of execution and avoids long periods of delayed answers.

2. To classify sliding-window operators and provide a a correct model of execution for the various classes.

We address both of these in the following section.

3. PIPELINED EXECUTION TECHNIQUES

In this section, we present our three approaches for scheduling window operators in pipelined query execution plans. The first approach (Time Probing) avoids the incorrect execution of the Clock Triggered approach presented in Figure 1(b). The second approach (Negative Tuple) avoids problems associated with the Input Triggered approach, such as the delay of output tuples presented in Figure 1(a). Finally, the *Hybrid* approach, aims (as would be expected) to obtain the advantages of both approaches while avoiding their drawbacks.

The approaches require all tuples (both input and intermediate) to be timestamped. In our model, these timestamps are represented as intervals using two attributes associated with each tuple: minimum timestamp $minTS$ and maximum timestamp $maxTS$. Timestamps are assigned by the query processor. For input tuples (i.e., base tuples read directly from a stream) $minTS$ and $maxTS$ are both set to the current system clock time upon entering the query processor. This time is called the create time of the tuple T_{create}. Intermediate tuples that are created by Join operators are assigned timestamps as follows: $maxTS$ is set to the largest of the $maxTS$ values of the tuples that contribute to the intermediate tuple and $minTS$ is set to the smallest of the $minTS$ values of those tuples.

For reasons that are explained in the following sections, the correctness of the three approaches requires that query operators always produce their output tuples with $maxTS$ monotonically increasing. This is important to maintain the notion of ordering during the pipeline execution.

3.1 Time Probing

Similar to the Clock Triggered approach described in the previous section, the Time Probing approach (TPA for short) uses a clock to enable operators to be scheduled even if their input queues are empty. However, unlike the Clock Triggered approach, with TPA, an operator expires an old tuple t only if it can be guaranteed that no tuple that arrives

subsequently at the operator will have a timestamp (either $minTS$ or $maxTS$) within the window size from t.

To verify this condition, the window operator searches down (probes) to its descendants in the pipeline for the oldest tuple, say t_o, that has been processed. Since tuples always arrive at an operator in increasing order of $maxTS$, the window operator can use the $maxTS$ of t_o to determine whether or not t can be expired. Let $|w|$ be the window size, then tuple t is expired during a time probe only if

$$t_o.maxTS - t.minTS > |w|$$

To implement TPA, every operator in the pipeline must store the value of $maxTS$ corresponding to the last processed (or probed) tuple. We refer to this value as the *LocalClock* of the operator. Furthermore, operators must provide a mechanism to report their LocalClocks to their parent operator in the pipeline. We extend the traditional operator iterator interface (i.e., *Open(), GetNext(), and Close()*), to include a new operator *GetTime()*, which returns the value of the LocalClock. GetTime() for the leaf operator (Scan operator) in the pipeline returns the current system clock if no tuples are in the input stream.

Figure 3(a) illustrates this process (we omit the Scan operator to simplify the figure). Each operator periodically calls GetTime() on its immediate child operators (one level calling - no recursion) thereby updating its LocalClock.

The following code-segment illustrates the steps of executing a unary operator when scheduled using TPA. The input tuple has the form: t_n =<List_of_attributes, TO = Tuple-Order>. *ExecOp* is the operator algorithm (described in detail in Section 4) and *State* is the set of local variables of the operator (e.g., stored tuples, flags, etc.).

```
If exists new tuple t_n at the input stream
    LocalClock = t_n.TO.maxTS
    ExecOp(t_n, LocalClock, State)
Else
    LocalClock = ChildOperator.GetTime()
    ExecOp(NULL, LocalClock, State)
EndIF
```

For binary operators (e.g., joins), *LocalClock* is the smallest maxTS of its input (or probed) children.

TPA may be hard to implement in Input Triggered (i.e., push-based) schedulers of traditional query processors as well as in recent stream data systems [2, 4]. In the following section, we propose a scheduling approach that overcomes this drawback.

3.2 Negative Tuple

The Negative Tuple approach (NTA for short) is inspired by the fact that, in general, window operators need to process *invalid tuples*. Recall that the MINUS operator in Q_2 of Figure 2(a) should invalidate tuple A at time T_2 to provide correct output. Higher operators in the pipeline could have processed tuple A and should invalidate their corresponding output as well. One can consider that an expired tuple is a form of an invalid tuple, where an expired tuple is invalid if it is no longer part of the current window. Therefore, we propose a *uniform* framework for executing window operators using the notion of *negative tuples*. In this section, we focus on how to create and propagate negative tuples among pipelined query operators. In Section 4.5 we describe the details of how each query operator processes a negative tuple.

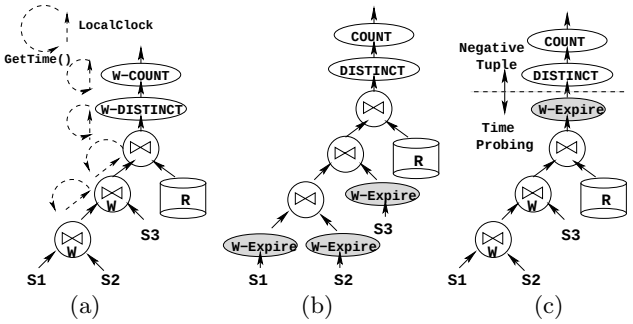


Figure 3: A query plan with different scheduling approaches (a) Time Probing approach (window only operators) (b)Negative Tuple approach with no window operators (c) Hybrid approach.

One interesting observation (that we further explain in Section 4.5) is that having negative tuples reduces the overall complexity of designing a query operator. This is a consequence of the fact that the operator no longer needs the window constraint to guide its execution, e.g., to expire an old tuple or to perform a binary operation. Moreover, operators using NTA are scheduled in a way that is exactly similar to those in the *Input Triggered* approach, i.e., when a tuple arrives at their input streams).

Since negative tuples are synthetic tuples, we add a new operator, *W-Expire*, to generate negative tuples for each input stream. W-Expire is placed at the start of the query pipeline and stores all tuples that have arrived in the last window. W-Expire adds new tuples to its stored buffer then forwards them to the next operator in the pipeline. Furthermore, W-Expire produces a negative tuple t^- when a stored tuple t_e expires. t^- has the same attributes of t_e and is tagged with a special *flag* that indicates that the tuple is negative. The tuple-order of t^- is as follows: minTS equals $t_e.minTS$ and maxTS equals $t_e.minTS + |w|$.

W-Expire is scheduled either when an input tuple arrives or a stored input is expected to expire. We illustrate NTA for a continuous query in Figure 3(b).

NTA has many advantages. It provides a uniform scheduling interface for query operators and eliminates the overhead of expiring tuples and verifying the window constraint by each query operators. However, a naive and straightforward implementation of NTA results in an additional overhead. NTA includes a new operator per input stream (W-Expire) with extra memory, execution and scheduling overhead. Furthermore, tuples may be processed twice in the pipeline (i.e., the first time as new tuples and the second time as negative tuples), therefore doubling the number of tuples traveling the pipeline. We overcome the first overhead by implementing W-Expire as part of the Scan operator. Therefore, W-Expire is not scheduled separately.

In the following section we present a *hybrid* approach that reduces the overhead of NTA.

3.3 Hybrid

One important observation is that join operations do not produce a new output when their tuples expire (notice how this is different from Group-By and Aggregate operations, which have to produce a new output when their tuples expire). As a result, the join operation in the NTA approach

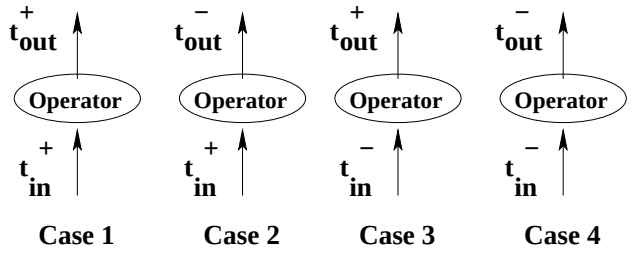


Figure 4: Different relationships between tuples in input and output streams.

simply propagates the negative tuples to the parent operator in the pipeline. Furthermore, the join cost for processing negative tuples and constructing the output is significant. Therefore, by *pulling-up* the W-Expire operator just after the join operations, the join overhead will decrease. Furthermore, W-Expire would reduce the complexity of its parent operators (especially when followed by multiple window operators such as the DISTINCT, Group-By, Aggregate and Set operations). Moreover, pushing the join operation down the pipeline is a common approach in stream processing systems to increase the possibility of sharing the join between multiple concurrent queries [8, 16].

The operator scheduling in the hybrid approach is as follows: The pipelined position of W-Expire constitutes a *partition point* in the query plan where all operators that follow W-Expire are scheduled using the NTA approach. However, the Time Probing approach schedules all operators in the child sub-tree that is rooted at the partition point. We illustrate the hybrid approach in Figure 3(c).

4. STREAM-IN STREAM-OUT WINDOW OPERATORS

In this section, we present window-based algorithms for window DISTINCT, window Group-By, window Set and window join operations. All presented algorithms adhere to the measure of correctness introduced in Section 2.2 (i.e., operators always maintain their *window-output* tuples equivalent to the output of a corresponding snapshot query). Throughout this section, we use the term *positive tuples or* t^+ for new tuples appended to the stream and the term *negative tuples or* t^- for expired or invalid tuples from the stream. t^- removes the previous effect of the corresponding expired or invalid tuple. Based on the type of input and output tuples, we can identify the following four cases: (Figure 4)

- *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.

ALGORITHM 4.1. **W-DISTINCT Algorithm**

```
1)  For all expired tuples,t_e =< D_e, TO_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 < D_e, TO_e > from DL
5)        Probe H using the values in D_e
6)        If a matching tuple < D_e, TO_h > is found in H
             /* A duplicate of the expired tuple
                still exists in the current window.
                If multiple tuples have the same D_e,
                choose one with the maximum TO_h.minTS */
7)           Add < D_e, [TO_h.minTS, TO_e.minTS + |w|] >
                to DL and to the output stream.
8)        EndIf
9)     EndIf
10)    Delete t_e
11) EndFor
12) If new tuple < D_n, TO_n > exists at the input stream
13)    Probe H using the values in D_n
14)    If no matching tuple is found in H
          /* Tuple < D_n, TO_n > is distinct */
15)       Add < D_n, TO_n > to DL and to the output stream
16)    EndIf
17)    Add < D_n, TO_n > to H
18) EndIf
```

Different cases are relevant to different window operators. For example, Cases 1 and 2 can arise in all window operators (e.g., window aggregate and window join). In the next sections, we describe execution models for window operators with different cases. We do not plan to cover all possible operators. However, we present sample algorithms for a set of window operators having different combinations of the above cases.

4.1 Window DISTINCT (W-DISTINCT)

Algorithm. Algorithm 4.1 gives the pseudo code of the hash-based W-DISTINCT operator that takes an input of the form $< D, TO >$, where D represents the values of the distinct attributes and TO is the tuple-order. The basic idea of the algorithm is to produce a new tuple t to replace an expired tuple t_e, if t_e was produced before as a distinct tuple and t is a duplicate for t_e (Case 3 in Figure 4). The tuple-order of the output tuples is assigned at either Step (7) or Step (15) of the Algorithm. The W-DISTINCT algorithm handles Case 1 and Case 4 in Figure 4. The W-DISTINCT Algorithm uses the following data structures:

- Hash table, H: stores the distinct tuples in the current window. The size of H never exceeds the window size.
- Distinct List, DL: stores all output distinct tuples sorted by their minTS.

Example. Figure 5 utilizes the example presented in Figure 2(b) (Stream $S1$ only). Stream $S2'$ is the proposed correct execution. We use the syntax $< a, [6, 7] >$ to refer to tuple a with tuple-order [6,7]. For equal values of the tuple-order we use one timestamp (e.g., $< a, 2 >$ for $< a, [2, 2] >$). The window size $|w|$ equals six clock ticks. In Figure 5 tuple $< a, 2 >$ expires at clock time 8. However, tuple $< a, 2 >$ was already reported as distinct. Since at clock time 8, value a is still one of the distinct values in the current window,

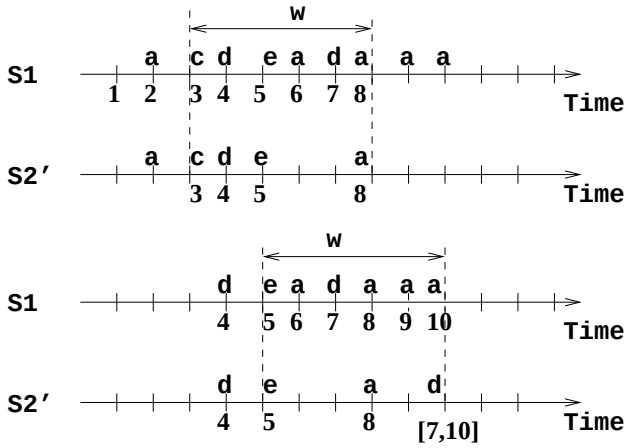


Figure 5: Evaluation of W-DISTINCT operator.

we output tuple $< a, 8 >$ as a new distinct tuple (Stream $S2'$ at time 8). While at time 8 we have two choices for the distinct tuple a (i.e., $< a, 6 >$ and $< a, 8 >$), to optimize the performance of the algorithm, we always choose (from a set of duplicate tuples) the tuple that is expected to expire last. In this way, output distinct tuples will have extended lifetime before expiring. Similar to the case when tuple $< a, 2 >$ expires, we output tuple $< d, [7, 10] >$ to replace tuple $< d, 4 >$ that expires at clock times 10. Notice that we choose the tuple-order of tuple $< d, [7, 10] >$ such that minTS equals the minTS of tuple $< d, 7 >$ in $S1$ and maxTS equals the current time.

Discussion. One interesting observation in W-DISTINCT is that the output rate from the W-DISTINCT operator never exceeds $\frac{n\_key}{|w|}$, where n_key is the total number of distinct tuples in the input stream and $|w|$ is the window size in time units. Therefore, for 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.

4.2 Window Aggregate and Group-By

Similar to W-DISTINCT, the correct execution of window aggregate (W-Aggregate) and window Group-By (W-Group-By) may produce a new positive output tuple when a tuple expires (Case 3 in Figure 4). In this section, we focus on the W-Group-By operator as W-Aggregate is a special case of W-Group-By with a single group.

Algorithm. Algorithm 4.2 gives the pseudo code of W-Group-By operator. The W-Group-By algorithm utilizes 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: $(< \mathcal{G}, \mathcal{A}, \mathcal{TO} >, \mathcal{GH})$.

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 have 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 $a_i \in \mathcal{A}$ 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 $AggrFn_1(.), \ldots, AggrFn_n(.)$. Also, we also use $\mathcal{V}$ to refer to set of results after applying function $\mathcal{F}$ on the group tuples.

ALGORITHM 4.2. **W-Group-By Algorithm**

```
1)  For all expired tuples, (< G_e, A_e, TO_e >, GH_e), in H
2)     Remove (< G_e, A_e, TO_e >, GH_e) from H
3)     Probe H with values in G_e
4)     If found a matching entry (< G_e, A_h, TO_h >, GH_e)
          /* The group still has non-expired tuples
             and should report its new aggregate values */
5)        Apply F_e for tuples in group G_e to get V.
6)        Add < G_e, V, [GH_e.minTS, TO_e.minTS + |w|] >
             to the output stream
7)     Else /* The Group expires */
8)        Add < G_e, NULL, TO_e.minTS + |w| >
             to the output stream
9)        Delete GH_e
10)    EndIf
11)    Delete (< G_e, A_e, TO_e >, GH_e)
12) EndFor
13) If exists new tuple < G_n, A_n, TO_n > at the input stream
14)    Probe H with values in G_n
15)    If found a matching entry, (< G_n, A_h, TO_h >, GH_n)
          /* Do nothing */
16)    Else /* Tuple < G_n, A_n, TO_n > forms a new group */
17)       Create GH_n for G_n
18)    EndIf
19)    Add (< G_n, A_n, TO_n >, GH_n) to H
20)    Apply F_n for tuples in group G_n to get V.
21)    Add < G, V, [GH_e.minTS, TO_n.max] >
          to the output stream
22) EndIfd
```

Discussion. To comply with the measure of correctness in Section 2.2, our proposed incremental evaluation of W-Group-By has the following properties:

1. W-Group-By reacts for every change in the input window contents.
2. The W-Group-By produces a NULL tuple for a group G that is no longer part of the current output (i.e., all tuples $\in G$ expire).
3. Operators followed by W-Group-By are able to distinguish those tuples that belong to the current W-Group-By result from the output stream.

To address the last property, W-Group-By assigns the tuple-order for the output tuples (either Step (8) or Step (21) of the Algorithm) such that only the output tuples that belong to the 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. This assumption must be considered by any operator that accepts the output from W-Group-By as its input (e.g., nested subqueries).

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 the old tuples. On the other hand, the window MINUS (W-MINUS) and window INTERSECT (W-INTERSECT) operators are quite involved. W-INTERSECT has similar Cases to those of W-Group-By. Therefore, in this section, we focus on the W-MINUS operator.

Algorithm. Algorithm 4.3 gives the pseudo code of the duplicate-preserving W-MINUS operator (e.g., MINUS ALL). The duplicate-free version of the operator can be easily implemented by having a W-DISTINCT operator following the W-MINUS. Invalid tuples are tagged with a special *flag* to be distinguished from output tuples. The invalid tuple is important only for the operators that follow the W-MINUS (if any). The processing of invalid tuples is described in detail in Section 4.5. The W-MINUS Algorithm stores the input tuples for streams S and R in the hash tables H_S and H_R, respectively. We consider input tuples of the form: $< \mathcal{A}, \mathcal{TO} >$, where $\mathcal{A}$ represents the attribute values and TO is the tuple-order. The tuple-order for an output tuple is specified at Step(7), Step(19) and Step (27) of the Algorithm while the invalid tuple are generated at Step (27).

Discussion. 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 the example in Figure 2(a), W-MINUS can produce invalid output tuple as it receives a new input tuple (*Case* 2). Furthermore, W-MINUS can produce new output tuples when a previously input tuples expires (*Case* 3). For example, consider an expired tuple t_e from Stream R that has no duplicates in R. However, t_e has duplicate tuples in Stream S. All duplicate tuples in S must be reported as new output when tuple t_e expires. In addition to *Cases* 2 and 3, W-MINUS also exhibits *Cases* 1 and 4. Therefore, the W-MINUS operator presents all the cases of Figure 4.

ALGORITHM 4.3. **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 must
         be processed in the order of their expiration */
2)    If t_e from stream R
3)      Remove t_e from H_R
4)      If no duplicates for t_e exists in H_R
          /* Expiring tuple from R may generate
             new output tuples from S */
5)        Probe H_S with values in A_e
6)        For all matching tuples, < A_e, TO_h > in H_S
7)          Add < A_e, [TO_h.minTS, TO_e.minTS + |w|] >
              to the output stream
8)        EndFor
9)      EndIF
10)   Else/* t_e is from stream S */
11)     Remove t_e from H_S
12)   EndIF
13)   Delete t_e
14) EndFor
15) If exists new tuple t_n =< A_n, TO_n > at the input stream of S
16)   Add t_n to H_S
17)   Probe H_R with values in A_n
18)   If no matching is found in H_R
19)     Add t_n to the output stream
20)   EndIf
21) EndIf
22) If exists new tuple t_n =< A_n, TO_n > at the input stream of R
23)   Add t_n to H_R
24)   If t_n is unique in H_R
25)     Probe H_S with values in A_n
26)     For all matching tuples < A_n, TO_h > in B_S
27)       Add invalid tuple < A_n, [TO_h.minTS, TO_n.maxTS] >
            to the output stream
29)     EndFor
28)   EndIf
30) EndIf
```

4.4 Window Join (W-Join)

Binary join iterates over all tuples in one input source (the outer data source) and 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, 18].

W-Join needs to address *Cases* 1 and 4 in Figure 4. 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 input tuples. The maxTS equals the maximum value of maxTS for all joined input tuples. Figure 6 illustrates the symmetric evaluation of W-Join assuming window size of *five* clock ticks. The output tuples are presented at each execution time. The W-Join execution at time 8 starts at top diagram in the second column of Figure 6.

4.5 Processing of Invalid and Negative Tuples

The proposed algorithms for window operators do not consider the case that they may receive an invalid or a negative tuple. Therefore, we modify the proposed algorithms for window operators to process invalid and negative tuples as follows (we refer to the original tuple (before invalidation) as t and the corresponding invalid or negative tuple as t^-): For project (with duplicates) and select operators, t^- is processed in the same way as t (e.g., project out some attributes or apply the selection predicate). For operators that maintain stored state (e.g., join and aggregate), t is first removed from the stored state of the operator (e.g., hash table, list, ... etc). Afterwards, the processing of t^- differs depending on the type of the operator.

For the W-DISTINCT operator, t^- falls into two cases: In the first case, t^+ was reported as distinct in the output stream (i.e., t^+ is found in the distinct list). Therefore, t^- must be reported again as *invalid* in the output stream. In addition, t^- may generate a new positive output (similar to case when expiring an old tuple in Algorithm 4.1, Steps 1-11). In the second case, t^+ was not reported as distinct in the output stream (i.e., t^+ is found in the distinct list). Therefore, there is no need to report t^- in the output stream.

For W-Group-By and W-Aggregate, the processing of t^- would generate a new output. The generated output represents the aggregate value over the new stored state (after removing tuple t^+). For a W-MINUS operator between two streams S and R, the processing of t^- is as follows: If t^- appears in input stream S, then t^- is produced in the output if t^+ matches no tuples in H_R. If t^- appears in input

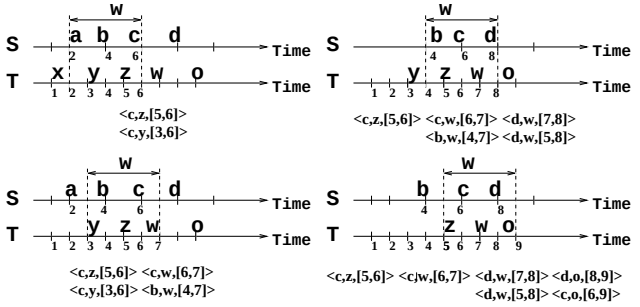


Figure 6: W-Join.

stream R, and no duplicate exists for t^+ in H_R then for each matching tuple in S produce a corresponding invalid output tuple. Finally, for W-Join, t^- from one stream joins with matching tuples in the other stream and produces *invalid* joined tuples (if any).

5. PROTOTYPE IMPLEMENTATION

To study the performance of the proposed approaches and algorithms, we implemented them inside a prototype stream data management system, based on a version of PREDATOR [23] extended for stream processing. 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 introduce a *StreamScan operator* to communicate with the stream table and retrieve new tuples. To schedule the operators, we implement each operator (including the StreamScan operator) as a separate thread that is scheduled preemptively by the operating system. The operators communicate with each other through a network of FIFO queues. Although our proposed approaches and algorithms can be adapted easily to work with user-controlled scheduling, we choose the operating system thread library for the following reasons: (1) Simplicity (no need for a complex user-defined scheduler). (2) To accommodate new types of operators recently proposed for stream query processing with intra-scheduling capabilities such as XJoin [28], PMJ [12], [21], and Shared W-Join [16].

We implemented the W-DISTINCT, W-Aggregate, W-Group-By and W-Set algorithms presented in Section 4. For the W-Join operator, we used a hash-based implementation similar to [14]. We augmented each window operator with the capability to process invalid tuples. Invalid tuples are tagged with special flags to distinguish them from input tuples (negative tuples are tagged similarly). The window operators in the Time Probing approach are scheduled periodically.

To study the Negative Tuple approach, we implemented the negative tuple version of all the above-mentioned operators. We added two implementations of the W-Expire: (1) as part of the StreamScan operator and (2) as an independent operator. We use the first implementation of W-Expire in the Negative Tuple approach. The Hybrid approach uses the second implementation of W-Expire. In the Negative Tuple approach, the arrival of an input tuple triggers operator scheduling.

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. We introduce the Set operators to the original code of PREDATOR.

The window specification is added as a special construct for the query syntax as was shown in the examples of Figures 1 and 2.

6. EXPERIMENTS

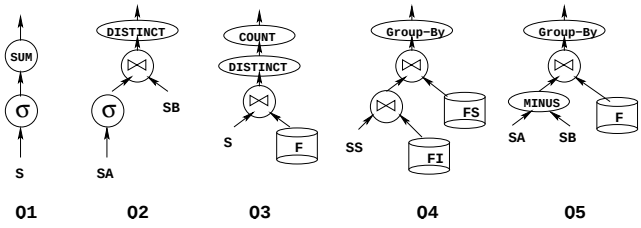


Figure 7: Execution plans for workload queries.

In this section, we compare the performance of the three proposed approaches for pipelined sliding window queries (i.e., Time Probing, Negative Tuples, and Hybrid approaches) against the Input Triggered approach. Our measures of performance are the average and maximum output delay (described in Section 2.2). To show the performance of the proposed approaches on different window operators (e.g., the window operators described in Section 4), we consider a workload of five different queries given in Table 1. We present the execution plans for the workload queries in Figure 7. The queries follow the schema from the motivating application in Section 2.3. 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.

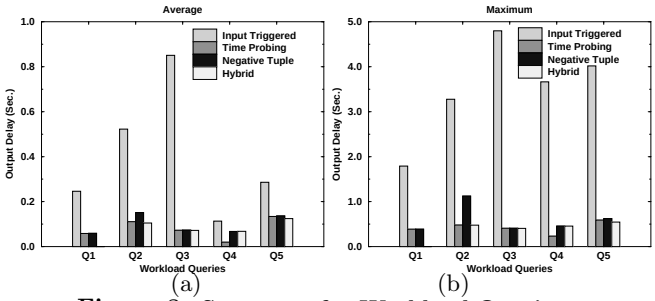
Unless mentioned otherwise, the predicate selectivity for Q_1 and Q_2 are set to 0.25. The window join selectivity in Q_2 is 0.6 (the overall selectivity in Q_2 is ≈ 0.1) and the join selectivity in Q_3, Q_4, and Q_5 is 0.1. All the experiments were run on Intel Pentium 4 CPU 2.4 GHz with 512 MB RAM 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.

6.1 Different Query Workloads

Figure 8 gives the output delay for the five workload queries (Table 1) when scheduled using the Input Triggered, Time Probing, Negative Tuple, and Hybrid approaches. We using synthetic input streams with average arrival rate of 10 tuples/second. The Hybrid approach is not applicable for Q_1 where there are no joins operators. In all queries, the Input Triggered approach incurs significant delays (0.85 seconds on average and 4.8 seconds maximum). The Time Probing, Negative Tuple and Hybrid approached give comparable performance. However, the Time Probing approach always provides the smallest output delay, followed by the Hybrid approach and finally the Negative Tuple approach. For Q_2, the Negative Tuple approach has higher response time compared with those of the Time Probing and Hybrid approaches. The reason is that Q_2 includes a W-Join (an expensive operator). Processing both new and negative tuples by the W-join increases processing time and output delay. The Hybrid approach gives improved performance

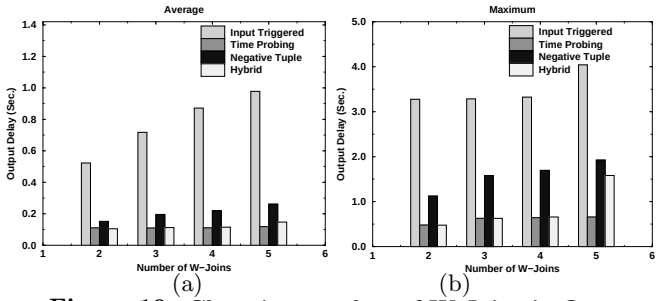
Table 1: Workload Queries.

Q_1	Q_2	Q_3	Q_4	Q_5
SELECT SUM(S.Price) FROM SalesStream S WHERE S.ItemID > Threshold Window 1 minute;	SELECT DISTINCT SA.ItemID FROM SalesStream_A SA, SalesStream_B SB WHERE SA.ItemID = SB.ItemID AND SA.Price > Threshold Window 1 minute;	SELECT COUNT (DISTINCT S.StoreID) FROM SalesStream S, FavoriteItems F WHERE S.ItemID = F.ItemID Window 1 minute;	SELECT SS.ItemID, SUM(SS.Price) FROM FavoriteItems FI, SalesStream SS, FavoriteStores FS WHERE FI.ItemID = SS.ItemID AND SS.StoreID = FS.StoreID Group By SS.ItemID WINDOW 1 minute;	SELECT S.ItemID, SUM(S.Price) FROM FavoriteItems F, (SELECT ItemID, Price FROM SalesStream_A MINUS SELECT ItemID, Price FROM SalesStream_B) as S WHERE F.ItemID = S.ItemID Group By S.ItemID WINDOW 1 minute;



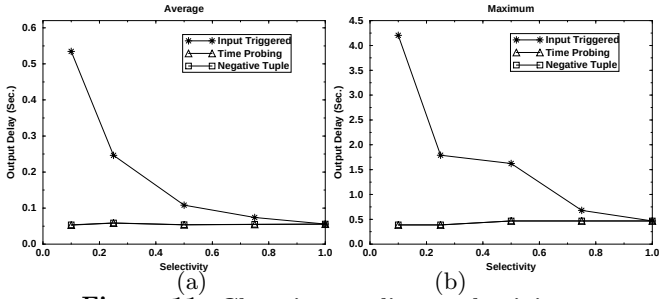
(a) (b)

Figure 8: Summary for Workload Queries.



(a) (b)

Figure 10: Changing number of W-Joins in Q2.



(a) (b)

Figure 11: Changing predicate selectivity.

similar to that of Time Probing. For Q_4, the Time Probing approach has better performance compared to the Negative Tuple and Hybrid approaches. The reason is that the W-Group-By operator is an expensive operator. Doubling the traffic bandwidth by processing new and negative tuples results in an increased overhead when using the Negative Tuple and Hybrid approaches.

Figure 9 gives the performance of the scheduling approaches for different arrival rates (from 5 to 40 tuples/second) when applied for Q_3 and Q_4. Other queries give similar performance measures as Q_3 and Q_4. The performance of all approaches converges as we increase input arrival rates. This behavior is expected where higher input rates produce more tuples to propagate up in the pipeline. Hence, refreshing the stored state of window operators and producing output tuples with shorter delays. However, the Input Triggered approach still provides higher output delay compared to the other approaches. The main reason is that the Input Triggered approach is much constrained by the underlying operator selectivity. Q_3 shows little improvement in the Input Triggered approach while increasing the input rate, mainly because Q_3 has a DISTINCT operator followed by the Aggregate operator (COUNT). The DISTINCT operator, as illustrated in Section 4.1, regulates the output rate even for higher input rates (i.e., never exceeds a threshold output rate even while increasing the input arrival rate). Therefore, the improvement of performance in the Input Triggered approach significantly reduces as the DISTINCT operator is executed earlier in the pipeline. The Time Probing approach has the lowest output delay in all the queries with little improvement as we increase the input rate.

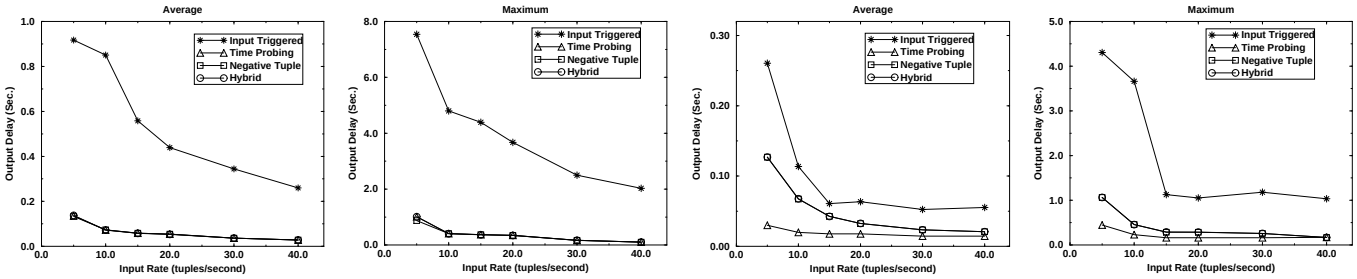
6.2 Depth of the Pipeline

Figures 10(a) and 10(b) give the average and maximum output delay incurred by Q_2 when increasing the number of joined streams from 2 to 5. In Figure 10(a), the Input Triggered approach gives the worst performance. This is mainly because as the pipeline gets deeper, a less number of tuples is expected to travel up the pipeline. Thus, the Input Triggered approach will not be refreshed frequently. The Negative Tuple approach has lower performance compared with the Hybrid and Time Probing approaches. The main reason is that the Negative Tuple approach almost doubles the number of tuples flowing in the pipeline. Thus, as the number of W-join operators increases, the overhead of doubling the traffic increases. The Hybrid approach performs similar to to that of Time Probing up to pipeline depth *four*. For a pipeline with depth five, the Time Probing approach has better performance than that of the Hybrid approach. This is mainly the result of the scheduling overhead introduced by the Hybrid approach (W-Expire imposes one additional operator to schedule).

The maximum output delay in Figure 10(b) has similar trend as that of Figure 10(a), however, with higher values of the output delays. This experiment illustrates the effectiveness of the Time Probing, Negative Tuple, and Hybrid approaches over the Input Triggered approach. Also, the experiments illustrate the superior performance of the Hybrid approach compared to the Negative Tuple approach.

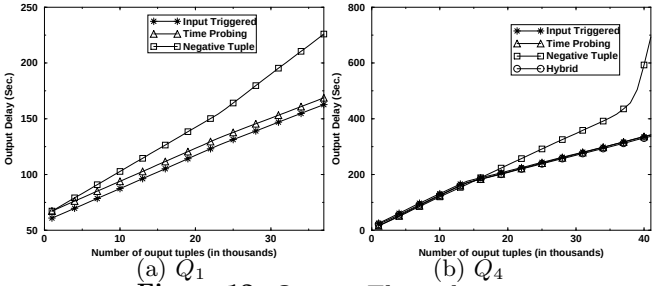
6.3 Changing Selectivity

Figure 11(a) and 11(b) give the effect of changing the selectivity (from 0.1 to 1) on the average and maximum output delay, respectively, when using the different scheduling approaches. We present only the results of the experiment



(a)Q_3(Average) (b)Q_3(Maximum) (c) Q_4(Average) (d)Q_4(Maximum)

Figure 9: Output delay for Q_3 and Q_4.



(a) Q_1 (b) Q_4

Figure 12: Output Throughput.

for Q_1 since similar performance is obtained from the other queries. With the increase in selectivity, all the scheduling approaches have low output delays. This is a result of having more tuples through the pipeline. With selectivity 1.0 (i.e., no filtration), all the scheduling approaches have the same performance. This indicates that for simple queries that do not have any filtration, the Input Triggered approach can be candidate for scheduling. Other scheduling approaches have a slight increase in the output delay with the increase in selectivity. This is mainly due to the additional processing overhead incurred by the Time Probing, Negative Tuple, and Hybrid approaches.

6.4 Output Throughput

In this section, we measure the execution speed of the various scheduling approaches. To measure the execution speed, we run the workload queries using 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 in a unit time. Figures 12(a) and 12(b) give the execution time needed by the scheduling approach to output up to 40K tuples for Q_1 and Q_4, respectively. Notice that, for Q_1 the Hybrid approach is inapplicable (no joins). We use synthetic data streams with arrival rate of 2056 tuples/second.

For Q_1, the Input Triggered approach provides the smallest execution time followed by Time Probing. The main reason is that Time Probing includes additional overhead of probing child operators. The Negative Tuple approach clearly gives higher execution time that increases as we receive more output tuples. For Q_4, the Input Triggered, Time Probing and Hybrid approaches provide comparable performance with little advantage for Time Probing at higher output values. The execution time of the Negative Tuple approach increases exponentially for the same reason as in Q_1.

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. [24] present the SEQ model and 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 [25] addresses handling of time in traditional databases. His seminal work includes a SQL formulation to evaluate complex predicates and joins over the time attributes. Temporal join [29] and Band-Join [11], are join operators that use a distance-guided predicate (similar to window join).

Industrial-strength DBMS with extensible index structures and optimized buffer management can be considered as strong candidates to evaluate temporal queries on large stored sequences of input data. Push-based execution of query operators as execution threads connected by queues is listed by Graefe in [15] as one design alternative 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 [26], which investigated the incremental evaluation of queries over append-only databases. None of these efforts addressed the execution of queries with windows.

As stated previously, stream query processing is currently being addressed in a number of systems such as Aurora [3], Telegraph [5] and STREAM [22]. All of these projects have recognized the need for windows to make queries over data streams practical. To date, however, these systems have focused on input-triggered approaches and have not detailed how they address the problems with that approach that we identified earlier in the paper. Thus, our work is largely complementary to these other projects.

Finally, work on punctuating data streams [27] is related to our Negative Tuple approach, however, such punctuations as described in that work have been used to delineate among groups of tuples, rather than referring to a single tuple as in our approach. Thus, the optimizations that we proposed in the Hybrid scheme were not investigated in that work.

8. CONCLUSIONS

Pipelined execution of sliding window queries is at the

core of emerging architectures for continuous query processing over data streams. Correct execution of multiple windowed and pipelined operators is essential for implementing a reliable query execution engine and benchmarking the performance among different stream processing systems.

We have described a correctness measure for the pipelined execution of sliding window queries. We proposed three scheduling approaches to guarantee the correct execution. The Time Probing approach synchronizes the local clock of each operator based on the most recent processed or probed tuple. The Negative Tuple approach uses the new idea of propagating a special tuple (negative tuple) to undo the effect of the expired tuples. The Hybrid approach mixes the techniques in Time Probing and Negative Tuple to improve performance. Among all the proposed approaches, the Negative Tuple approach was the simplest to implement. We also described the different relationships between input and output tuples using the positive-negative tuple paradigm. This helps in identifying various classes of sliding window operators.

We presented incremental algorithms for window DISTINCT, window Group-By, and Window MINUS as examples of different classes of sliding window operators. We described how each operator processes both positive and negative tuples to maintain correct execution. We performed experiments based on an implementation of the three proposed approaches and algorithms in a prototype stream DBMS. The results showed that the proposed scheduling algorithms provide more than an order of magnitude reduction in output delays when compared to the Input Triggered scheduling approach. Remarkably, this performance is achieved at low input stream arrival rate.

In terms of future work, we believe that the negative tuple, in addition to being essential for correct execution, provides a uniform framework to describe sliding window operations. Based on the idea of the Negative Tuple approach, we plan to study traditional and new optimizations in stream query processing and how it can be applied for processing sliding window queries over data streams.

9. ADDITIONAL AUTHORS

10. REFERENCES

[1] D. Abadi, D. Carney, U. Cetintemel, and et al. Aurora: A new model and architecture for data stream management. In *VLDB Journal, August*, 2003.

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

[3] D. Carney, U. Cetintemel, M. Cherniack, and et al. Monitoring streams - a new class of data management applications. In *VLDB Conference, 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 *1st CIDR Conf., Jan 2003, Asilomar, CA.*, 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 *Proceedings of the Twentieth International Conference on Very Large Databases*, Santiago, Chile, 1994.

[8] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE, Feb.*, 2002.

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

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

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

[12] J.-P. Dittrich, B. Seeger, D. S. Taylor, and et al. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB*, 2002.

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

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

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

[16] M. A. Hammad, M. J. Franklin, W. G. Aref, and et al. Scheduling for shared window joins over data streams. In *VLDB*, 2003.

[17] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *PODS*, 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*, 2003.

[21] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, 2004.

[22] R. Motwani, J. Widom, A. Arasu, and et al. Query processing, approximation, and resource management in a data stream management system. In *1st CIDR Conf., Jan.*, 2003.

[23] P. Seshadri. Predator: A resource for database research. *SIGMOD Record*, 27(1):16–20, 1998.

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

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

[26] D. Terry, D. Goldberg, D. Nichols, and et al. Continuous queries over append-only databases. In

SIGMOD, 1992.

[27] 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.

[28] T. Urhan and M. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin 23(2)*, 2000.

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