Incremental Evaluation of Sliding-Window Queries over Data Streams*

Thanaa M. Ghanem[1] Moustafa A. Hammad[2] Mohamed F. Mokbel[3] Walid G. Aref[1] Ahmed K. Elmagarmid[1]

[1]Department of Computer Science, Purdue University.

[2]Department of Computer Science, University of Calgary.

[3]Department of Computer Science and Engineering, University of Minnesota.

Abstract

Two research efforts have been conducted to realize sliding-window queries in data stream management systems, namely, query *re-evaluation* and *incremental evaluation*. In the query re-evaluation method, two consecutive windows are processed independent from each other. On the other hand, in the incremental evaluation method, the query answer for a window is obtained incrementally from the answer of the preceding window. In this paper, we focus on the *incremental evaluation* method. Two approaches have been adopted for the incremental evaluation of sliding-window queries, namely, the *input-triggered* approach and the *negative tuples* approach. In the *input-triggered* approach, only the newly inserted tuples flow in the query pipeline and tuple expiration is based on the timestamps of the newly inserted tuples. On the other hand, in the *negative tuples* approach, tuple expiration is separated from tuple insertion where a tuple flows in the pipeline for every inserted or expired tuple. The *negative tuples* approach avoids the unpredictable output delays that result from the *input-triggered* approach. However, *negative tuples* double the number of tuples through the query pipeline, thus reducing the pipeline bandwidth. Based on a detailed study of the incremental evaluation pipeline, we classify the incremental query operators into two classes according to whether an operator can avoid the processing of negative tuples or not. Based on this classification, we present several optimization techniques over the *negative tuples* approach that aim to reduce the overhead of processing negative tuples while avoiding the output delay of the query answer. A detailed experimental study, based on a prototype system implementation,

*Walid Aref's research is supported in part by the National Science Foundation under Grants IIS-0093116 and IIS-0209120.

shows the performance gains over the *input-triggered* approach of the *negative tuples* approach when accompanied with the proposed optimizations.

Keywords: Data stream management systems, pipelined query execution, negative tuples.

1 Introduction

The emergence of data streaming applications calls for new query processing techniques to cope with the high rate and the unbounded nature of data streams. The *sliding-window* query model is introduced to process continuous queries in-memory. The main idea is to limit the focus of continuous queries to only those data tuples that are inside the introduced *window*. As the window slides, the query answer is updated to reflect both new tuples entering the window and old tuples expiring from the window. Two research efforts have been conducted to support sliding-window queries in data stream management systems, namely, query *re-evaluation* and *incremental evaluation*.

In the query *re-evaluation* method, the query is re-evaluated over each window independent from all other windows. Basically, buffers are opened to collect tuples belonging to the various windows. Once a window is completed (i.e., all the tuples in the window are received), the completed window buffer is processed by the query pipeline to produce the complete window answer. An input tuple may contribute to more than one window buffer at the same time. Examples of systems that follow the query re-evaluation method include Aurora [1] and Borealis [2]. On the other hand, in the *incremental evaluation* method, when the window slides, only the changes in the window are processed by the query pipeline to produce the answer of the next window. As the window slides, the changes in the window are represented by two sets of inserted and expired tuples. Incremental operators are used in the pipeline to process both the inserted and expired tuples and to produce the incremental changes to the query answer as another set of inserted and expired tuples. Examples of systems that follow the incremental evaluation approach include STREAM [3] and Nile [20].

In this paper, we focus on the *incremental evaluation* method. Two approaches have been adopted to support incremental evaluation of sliding-window queries, namely, the *input-triggered* approach and the *negative tuples* approach. In the input-triggered approach (ITA for short), only the newly inserted tuples flow in the query pipeline. Query operators (and the final query output) rely on the timestamps of the inserted tuples to expire old tuples [5, 23]. However, as will be

discussed in Section 3.1, ITA may result in significant delays in the query answer. As an alternative, the negative tuples approach (NTA for short) is introduced as a delay-based optimization framework that aims to reduce the output delay incurred by ITA [4, 21]. A negative tuple is an artificial tuple that is generated for every expired tuple from the window. Expired tuples are generated by a special operator, termed `EXPIRE`, placed at the bottom of the query pipeline (`EXPIRE` is a generalization of the operators `SEQ-WINDOW` in [4] and `W-EXPIRE` in [21]). For each inserted tuple in the window (i.e., *positive* tuple), say t, `EXPIRE` forwards t to the higher operator in the pipeline. `EXPIRE` emits a corresponding *negative* tuple t^- once t expires from the sliding window. As the *expired* tuple flows through the query pipeline, it undoes the effect of its corresponding *inserted* tuple.

Although the basic idea of NTA is attractive, it may not be practical. The fact that a negative tuple is introduced for every expired input tuple means doubling the number of tuples through the query pipeline. In this case, the overhead of processing tuples through the various query operators is doubled. This observation opens the room for optimization methods over the basic NTA. Various optimizations would mainly focus on two issues: (1) Reducing the overhead of processing the negative tuples. (2) Reducing the number of the negative tuples through the pipeline.

In this paper, we study the realization of the incremental evaluation approaches in terms of the design of the incremental evaluation pipeline. Based on this study, we classify the incremental relational operators into two classes according to whether an operator can avoid the processing of expired tuples or not. Then, we introduce several optimization techniques over the negative tuples approach that aim to reduce the overhead of processing negative tuples while avoiding the output delay of the query answer. The first optimization, termed the *time-message* optimization, is specific to the class of operators that can avoid the processing of negative tuples. In the *time-message* optimization, when an operator receives a negative tuple, the operator does not perform exact processing but just "passes" a time-message to upper operators in the pipeline. Whenever possible, the time-message optimization reduces the overhead of processing negative tuples while avoiding the output delay of the query answer.

Furthermore, we introduce the *piggybacking* approach as a general framework that aims to reduce the number of negative tuples in the pipeline. In the piggybacking approach, negative tuples flow in the pipeline *only* when there is no concurrent positive tuple that can do the expiration. Instead, if positive tuples flow in the query pipeline with high rates, then the positive tuples purge

the negative tuples from the pipeline and are *piggybacked* with the necessary information for expiration. Alternating between negative and piggybacked positive tuples is triggered by discovering fluctuations in the input stream characteristics that is likely to take place in streaming environments. Basically, the piggybacking approach *always* achieves the minimum possible output delay *independent* from the stream or query characteristics. In general, the contributions of this paper can be summarized as follows:

1. We study, in detail, the realization of the *incremental evaluation* approach in terms of the design of the incremental evaluation pipeline. Moreover, we compare the performance of the two approaches, ITA and NTA, for various queries. This comparison helps identify the appropriate situations to use each approach.

2. We give a classification of the incremental operators based on the behavior of the operator when processing a negative tuple. This classification motivates the need for optimization techniques over the basic NTA.

3. We introduce the *time-message* optimization technique that aims to avoid, whenever possible, the processing of negative tuples while avoiding the output delay of the query answer.

4. We introduce the *piggybacking* technique that aims to reduce the number of negative tuples in the query pipeline. The piggybacking technique allows the system to be stable with fluctuations in input arrival rates and filter selectivity.

5. We provide an experimental study using a prototype data stream management system that evaluates the performance of the *ITA*, *NTA*, *time-message*, and *piggybacking* techniques.

The rest of the paper is organized as follows: Section 2 gives the necessary background on the pipelined query execution model in data stream management systems. Section 3 discusses and compares ITA and NTA for the incremental evaluation of sliding-window queries. Detailed realization of the various operators is given in Section 4. A classification for the incremental operators along with the optimizations over the basic NTA are introduced in Section 5. Section 6 introduces the *piggybacking* technique. Experimental results are presented in Section 7. Section 8 highlights related work in data stream query processing. Finally, Section 9 concludes the paper.

2 Preliminaries

In this section, we discuss the preliminaries for sliding-window query processing. First, we discuss the semantics of sliding-window queries. Then, we discuss the pipelined execution model for the incremental evaluation of sliding-window queries over data streams.

2.1 Sliding-window Query Semantics

A sliding-window query is a continuous query over n input data streams, S_1 to S_n. Each input data stream S_j is assigned a window of size w_j. At any time instance T, the answer of the sliding-window query equals to the answer of the snapshot query whose inputs are the elements in the current window for each input stream. At time T, the current window for stream S_i contains the tuples arriving between times $T - w_i$ and T. The same notions of semantics for continuous sliding-window queries are used in other systems (e.g., [24, 27]). In our discussion, we focus on the time-based sliding window that is the most commonly used sliding window type. Input tuples from the input streams, S_1 to S_n, are timestamped upon the arrival to the system. The timestamp of the input tuple represents the time at which the tuple arrives to the system. The window w_i associated with stream S_i represents the lifetime of a tuple t from S_i.

Handling timestamps: A tuple t carries two timestamps, t's arrival time, ts, and t's expiration time, Ets. Operators in the query pipeline handle the timestamps of the input and output tuples based on the operator's semantics. For example, if a tuple t is generated from the join of the two tuples $t1(ts1, Ets1)$ and $t2(ts2, Ets2)$, then t will have $ts = max(ts1, ts2)$ and $Ets = min(Ets1, Ets2)$. In this paper, we use the CQL [4] construct RANGE to express the size of the window in time units.

2.2 Data Stream Queuing Model

Data stream management systems use a pipelined queuing model for the incremental evaluation of sliding-window queries [4]. All query operators are connected via first-in-first-out queues. An operator, p, is scheduled once there is at least one input tuple in its input queue. Upon scheduling, p processes its input and produces output results in p's output queue. The stream SCAN (SSCAN) operator acts as an interface between the streaming source and the query pipeline. SSCAN assigns

to each input tuple two timestamps, ts which equals to the tuple arrival time, and Ets which equals to $ts + w_i$. Incoming tuples are processed in increasing order of their arrival timestamps.

Stream query pipelines use incremental query operators. Incremental query operators process changes in the input as a set of inserted and expired tuples and produce the changes in the output as a set of inserted and expired tuples. Algebra for the incremental relational operators has been introduced in [18] in the context of incremental maintenance of materialized views (expiration corresponds to deletions). In order to process the inserted and expired tuples, some query operators (e.g., Join, Aggregates, and Distinct) are required to keep some state information to keep track of all previous input tuples that have not expired yet.

3 Pipelined-execution of Sliding-window Queries

In this section, we discuss two approaches for the incremental evaluation of sliding-window queries, namely ITA and NTA. As the window slides, the changes in the window include insertion of the newly arrived tuples and expiration of old tuples. ITA and NTA are similar in processing the inserted (or positive) tuples but differ in handling the expired (or negative) tuples. Basically, the difference between the two approaches is in: (1) how an operator is notified about the expiration of a tuple, (2) the actions taken by an operator to process the expired tuple, and (3) the output produced by the operator in response to expiring a tuple. In this section, we discuss how each approach handles the expiration of tuples along with the drawbacks of each approach.

3.1 The Input-triggered Approach (ITA)

The main idea in ITA is to communicate only positive tuples among the various operators in the query pipeline. Operators in the pipeline (and the final query sink) use the timestamp of the positive tuples to expire tuples from the state. Basically, tuple expiration in ITA is as follows: (1) An operator learns about the expired tuples from the current time T that equals to the newest positive tuple's timestamp. (2) Processing an expired tuple is operator-dependent. For example, the join operator just purges the expired tuples from the join state. On the other hand, most of the operators (e.g., Distinct, Aggregates and Set-difference) process every expired tuple and produce new output tuples. (3) An operator produces in the output only positive tuples resulted from

processing the expired tuple (if any). The operator attaches the necessary time information in the produced positive tuples so that upper operators in the pipeline perform the expiration accordingly.

A problem arises in ITA if the operator does not produce any positive tuples in the output although the operator has received input positive tuples and has expired some tuples from the operator's state. In this case, the upper operators in the pipeline are not notified about the correct time information, which results in a delay in updating the query answer. Note that upper operators in the pipeline should not expire any tuples until the operator receives an input tuple from the lower operator in the pipeline. Operators cannot voluntarily expire tuples based on a global system's clock. Voluntary expiration based on a global clock can generate incorrect results because an expired tuple, t_1, may co-exist in the window with another tuple, t_2, but t_2 may get delayed at a lower operator in the pipeline. An example demonstrating this incorrect execution when using a global clock is given in Appendix A.

The delay in the query answer is a result of not propagating the time information that is needed to expire tuples. The delay is unpredictable and depends on the input stream characteristics. In a streaming environment, a delay in updating the answer of a continuous query is not desirable and may be interpreted by the user as an erroneous result. As it is hard to model the input stream characteristics, the performance of the input-triggered approach is fluctuating.

Example: Consider the query Q_1 *"Continuously report the number of favorite items sold in the last five time units"*. Notice that even if the input is continuously arriving, the filtering condition, *favorite items*, may filter out many of the incoming stream tuples. In this case, the join operator will not produce many positive tuples. As a result, the upper operators in the pipeline (e.g., COUNT in Q_1) will not receive any notification about the current time and hence will not expire old tuples.

Figure 1 illustrates the behavior of ITA for Q_1. The timelines S_1 and S_2 correspond to the input stream and the output of JOIN, respectively. S_3 and C represent the output stream when using ITA and the correct output, respectively. The window w is equal to five time units. Up to time T_4, Q_1 matches the correct output C with the result 4. At T_5, the input "2" in S_1 does not join with any item in the table *FavoriteItems*. Thus, COUNT is not scheduled to update its result. S_3 will remain 4 although the correct output C should be 3 due to the expiration of the tuple that arrived at time T_0. Similarly, at T_6, S_3 is still 4 while C is 2 (the tuple arriving at time T_1 has expired). S_3 keeps having an *erroneous* output till an input tuple passes the join and triggers the scheduling

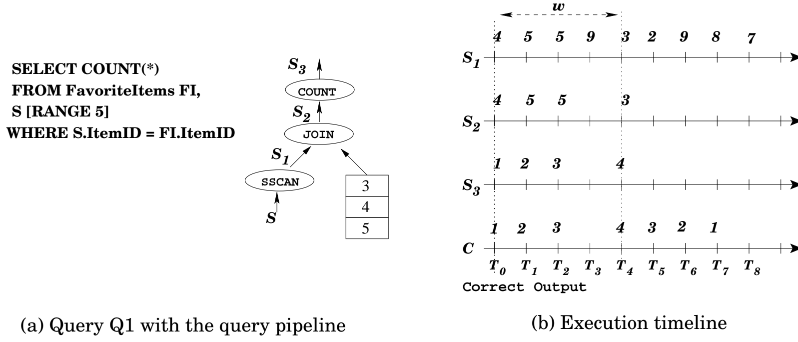


Figure 1: Input-triggered evaluation.

of COUNT to produce the correct output. This erroneous behavior motivates the idea of having a new technique that triggers the query operators based on either tuple insertion or expiration.

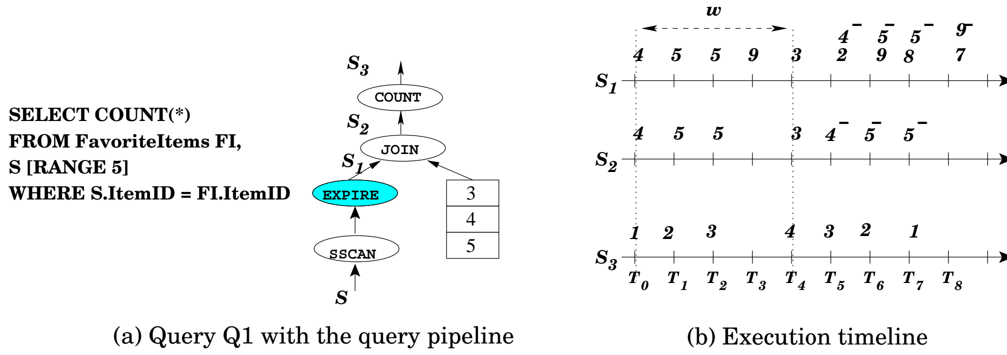
3.2 The Negative Tuples Approach (NTA)

The main goal of NTA is to separate tuple expiration from the arrival of new tuples. The main idea is to introduce a new type of tuples, namely negative tuples, to represent expired tuples [4, 21]. A special operator, `EXPIRE`, is added at the bottom of the query pipeline that emits a negative tuple for every expired tuple. A negative tuple is responsible for undoing the effect of a previously processed positive tuple. For example, in time-based sliding-window queries, a positive tuple t^+ with timestamp T from stream I_j with window of length w_j, will be followed by a negative tuple t^- at time $T + w_j$. The negative tuple's timestamp is set to $T + w_j$. Upon receiving a negative tuple t^-, each operator in the pipeline behaves accordingly to delete the expired tuple from the operator's state and produce outputs to notify upper operators of the expiration.

3.2.1 Handling Delays Using Negative Tuples

Figure 2b gives the execution of NTA for the example in Figure 2a (the negative tuples implementation of the query in Figure 1a). At time T_5, the tuple with value 4 expires and appears in S_1 as a negative tuple with value 4. The tuple 4^- joins with the tuple 4 in the *FavoriteItems* table. At time T_5, COUNT receives the negative tuple 4^-. Thus, COUNT outputs a new count of 3. Similarly at time T_6, COUNT receives the negative tuple 5^- and the result is updated.

The previous example shows that NTA overcomes the output delay problem introduced by ITA



(a) Query Q1 with the query pipeline

(b) Execution timeline

Figure 2: Negative tuples evaluation.

because tuple expiration is independent from the query characteristics. Even if the query has highly selective operators at the bottom of the pipeline, the pipeline still produces timely correct answers. On the other hand, if the bottom operator in the query pipeline has low selectivity then almost all the input tuples pass to the intermediate queues. In this case, NTA may present more delays due to the increase of waiting times in queues.

3.3 Invalid Tuples

In ITA, expired tuples are not explicitly generated for every expired tuple from the window but some tuples may expire before their *Ets* due to the semantics of some operators (e.g., set-difference) as will be explained in Section 4. In the rest of the paper, we refer to tuples that expire out-of-order as *invalid* tuples. Operators in ITA process invalid tuples in the same way as negative tuples are processed by NTA and produce outputs so that other operators in the pipeline behave accordingly. This means that even in ITA, some negative tuples may flow in the query pipeline.

4 Window Query Operators

Window query operators differ from traditional operators in that window operators need to process the expired tuples as well as the inserted tuples. Two issues should be distinguished when discussing window operators: operator *semantics* and operator *implementation*. Operator *semantics* defines the changes in operator's output when the input is changed (by inserting or deleting a tuple) while operator *implementation* defines the way the operators in the pipeline are coordinated to achieve the desired semantics. Operator semantics is independent from the approach (ITA or NTA) used for

query evaluation. Incremental semantics for various relational operators is defined in the context of incremental maintenance of materialized views [18]. On the other hand, operator implementation depends on whether ITA or NTA is used for query evaluation. In this section, we discuss the semantics and implementation issues for the various relational operators under ITA and NTA.

4.1 Incremental Evaluation

In this section, we use the incremental equations from [18] as a guide for discussing the semantics of the various window operators. Two equations are given for every operator, one equation gives the semantics when the input changes by *inserting* a tuple and the other equation gives the semantics when the input changes by *deleting* a tuple. In stream operators, inputs are streams of inserted and expired tuples. At any time point T, an input stream S can be seen as a relation that contains the input tuples that have arrived before time T and have not expired yet. After time T, an input positive tuple s^+ indicates an insertion to S, represented as $(S + s)$, and an expired tuple s^- indicates a deletion from S, represented as $(S - s)$. In the following, we assume the duplicate-preserving semantics of the operators. Tuples arriving to the system out-of-order can be stored in buffers and can be ordered using heartbeats [25]. Ordering tuples is beyond the scope of this paper.

4.2 Window Select $\sigma_p(S)$ and Window Project $\pi_A(S)$

$\sigma_p(S + s) = \sigma_p(S) + \sigma_p(s) \qquad \sigma_p(S - s) = \sigma_p(S) - \sigma_p(s)$
$\pi_A(S + s) = \pi_A(S) + \pi_A(s) \qquad \pi_A(S - s) = \pi_A(S) - \pi_A(s)$

The incremental equations for Select and Project show that both positive and negative tuples are processed in the same way. The only difference is that positive inputs result in positive outputs and negative inputs result in negative outputs. The equations also show that processing an input tuple does not require access to previous inputs, hence Select and Project are non-stateful operators. An output tuple carries the same timestamp and expiration timestamp as the corresponding input tuple. In ITA, Select and Project do not produce any outputs in response to an expired input tuple.

4.3 Window Join $(S \bowtie R)$

$(S + s) \bowtie R = (S \bowtie R) + (s \bowtie R) \qquad (S - s) \bowtie R = (S \bowtie R) - (s \bowtie R)$

Join is symmetric which means that processing a tuple is done in the same way for both input sides. The incremental equations for Join show that, like Select, Join processes positive and negative tuples in the same way with the difference in the output sign. Unlike Select, Join is stateful since it accesses previous inputs while processing the newly incoming tuples. The join state can be expressed as two multi-sets, one for each input. An output tuple from Join carries the semantics (windows) of two different streams. The timestamp of the output tuples is assigned as follows: the timestamp, ts, equals the maximum value of the timestamps for all joined tuples. The expiration timestamp, Ets, equals the minimum value of expiration timestamps for all joined tuples (output of the join should expire whenever any of its composing tuples expires). In ITA, Join does not produce any outputs in response to an expired input tuple.

4.4 Window Set Operations

We consider the duplicate-preserving semantics of the set operations as follows: if stream S has n duplicates of tuple a and stream R has m duplicates of the same tuple a, the union stream $(S \cup R)$ has $(n+m)$ duplicates of a, the intersection stream $(S \cap R)$ has $min(n,m)$ duplicates of a, and the minus stream $(S - R)$ has $max(0, n-m)$ duplicates of a.

4.4.1 Window Union $(S \cup R)$

$(S+s) \cup R = (S \cup R) + s \qquad (S-s) \cup R = (S \cup R) - s$

An input tuple to the union operator is produced in the output with the same sign. In ITA, Union does not produce any outputs in response to an expired tuple. Union is non-stateful since processing an input tuple does not require accessing previous inputs. An output tuple carries the same timestamp and expiration timestamp as the input tuple.

4.4.2 Window Intersection$(S \cap R)$

$(S+s) \cap R = (S \cap R) + (s \cap (R - S)) \qquad (S-s) \cap R = (S \cap R) - (s - (S - R))$

The intersection operator is symmetric. When a tuple s is inserted into stream S, s is produced in the output only if s has duplicates in the set "$R - S$" ("$R - S$" includes the tuples that exist in R and does not exist in S). On the other hand, when a tuple s expires, s should expire from the

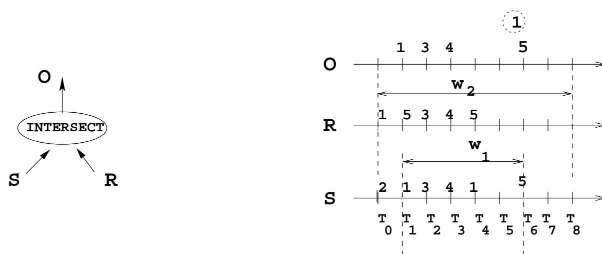


Figure 3: Window Intersection in ITA.

output only if s has no duplicates in the set "$S - R$".

When using ITA, Intersection needs to produce additional positive tuples in response to expiring a tuple. Figure 3 gives an example to illustrate this case. Assume that S and R are the two input streams and O is the output of Intersection. When the tuple "1" arrives in stream S at time T_1, a corresponding tuple "1" is produced in the output. At time T_6, the tuple with value "5" arrives to S and causes the expiration of the tuple "1". When the tuple "5" is propagated to the output stream, O, "5" causes the expiration of the tuple "1" from O as well. In this case, Intersection should produce another positive tuple with value 1 in the output stream to replace the expired tuple. A similar case happens in Distinct as will be described later.

4.4.3 Window Minus ($S - R$)

Case 1: $(S + s) - R = (S - R) + (s - (R - S))$ **Case 3:** $S - (R + r) = (S - R) - (r \cap (S - R))$

Case 2: $(S - s) - R = (S - R) - (s \cap (S - R))$ **Case 4:** $S - (R - r) = (S - R) + (r - (R - S))$

The minus operator is asymmetric, which means that processing an input tuple depends on whether the tuple is from S or R. The four cases for the input tuples are handled as follows:

- **Case 1**: An input positive tuple, s^+, from stream S is produced as a positive tuple in the output stream only if s does not exist in the set "$R - S$".

- **Case 2**: An input negative tuple, s^-, from stream S is produced in the output stream as a negative tuple only if s exists in the set "$S - R$". In ITA, the Minus operator does not produce any output in response to a tuple expiring from stream S.

- **Case 3**: An input positive tuple, r^+, from stream R results in producing a negative tuple s^- for a previously produced positive tuple s^+ when s is a duplicate for r and s exists in the

set "$S - R$". Note that the negative tuple s^- is an *invalid* tuple and is produced when using either ITA or NTA.

- **Case 4**: An input negative tuple, r^-, from stream R results in producing a positive tuple s^+ when s is a duplicate of r and s does not exist in the set "$R - S$". The positive tuple s^+ is produced in both ITA and NTA.

Minus is stateful since processing a positive or negative input tuple requires accessing previous inputs. In Cases 1 and 2, the output tuple carries the same timestamp as the input tuple. In Cases 3 and 4, the input tuple is from stream R while output tuple s is from stream S and carries timestamp from the stored s tuple.

4.5 Window Distinct ϵ

$\epsilon(S + s) = \epsilon(S) + (s - S) \qquad \epsilon(S - s) = \epsilon(S) - (s - (S - s))$

The semantics of the distinct operator states that an input positive tuple, s^+, is produced in the output only if s has no duplicates in S (i.e., s exists in the set "$s - S$"). An input negative tuple, s^-, is produced in the output only if s has no duplicates in the set "$S - s$". The Distinct operator is stateful. Similar to Intersection, when using ITA, Distinct may need to produce a positive tuple in response to expiring a tuple.

4.6 Window Aggregates and Group-By

The group-by operator maps each input stream tuple to a group and produces one output tuple for each non-empty group G. the output tuples have the form $< G, Val >$ where G is the group identifier and Val is the group's aggregate value. The aggregate value Val_i for group G_i is updated whenever the set of G_i's tuples changes, by inserting or expiring a tuple. Two tuples are produced to update the value of the group: an invalid tuple to cancel the old value and a positive tuple to report the new value. The behavior of Group-By is the same for both ITA and NTA and works as follows. When receiving an input tuple, s^+, or when a tuple expires, s^-, Group-By maps s to the corresponding group, G_s, and produces an invalid tuple, $< G_s, oldVal >^-$, to invalidate the old value of G_s, if G_s exists before, and another positive tuple, $< G_s, newVal >^+$, for the new value of G_s after aggregating s.

Aggregate operator's state: When using ITA, the aggregate operator stores all the input tuples in the operator's state. When using NTA, some aggregate operators (e.g., Sum and Count) do not require storing the tuples. These aggregates are incremental, and when receiving a negative tuple, the new aggregate value can be calculated without accessing the previous inputs. Other aggregates (e.g., MAX) require storing the whole input independent from using ITA or NTA.

4.7 Result Interpretation

In ITA, the output of a sliding-window query is a stream of positive tuples. Two timestamps are attached with each output tuple: a timestamp, ts, and an expiration timestamp, Ets. When a tuple with ts equals to T is received in the output, all previously produced tuples with Ets less than T should expire. The output of a sliding-window query should be stored in order to identify the expired tuples. In NTA, the output of a sliding-window query is a stream of positive and negative tuples. Each negative tuple cancels a previously produced positive tuple with the same attributes.

5 Negative Tuples Optimizations

Although the basic idea of NTA is attractive, it may not be practical. The fact that we introduce a negative tuple for every expired tuple results in doubling the number of tuples through the query pipeline. In this case, the overhead of processing tuples through the various query operators is doubled. This observation gives rise to the need for optimization methods over the basic NTA. The proposed optimizations focus mainly on two targets: (1) Reducing the overhead of processing the negative tuples. (2) Reducing the number of negative tuples through the pipeline.

Based on the study of the window query operators in Section 4, we classify the query operators into two classes according to whether an operator can avoid the complete processing of a negative tuple or not. Based on this classification, we propose optimizations to reduce the overhead of processing negative tuples whenever possible (target (1) above). In Section 6, we address optimizations to reduce the number of negative tuples in the pipeline (target (2) above). Before discussing the proposed optimizations, it is important to distinguish between two types of negative tuples: (1) expired tuples that are generated from the EXPIRE operator, and (2) invalid tuples that are generated from internal operators (e.g., Minus). Invalid tuples are generated out-of-order and have

to be fully processed by the various operators in the pipeline. The proposed optimizations aim to reduce the overhead of expired tuples and hence are not applied to invalid tuples.

5.1 Operator Classification

Based on the study of window operators in Section 4, we classify the window operators into two classes according to whether an operator can avoid the processing of negative tuples or not while guaranteeing a limited output delay.

- **Class 1:** The first class of window operators includes the operators in which an expired tuple repeats the output that was previously produced by the corresponding positive tuple. This class includes the following operators: Select, Project, Union, and Join. The only difference between the output in response to processing an expired tuple and the output in response to processing the corresponding positive tuple is in the tuple's sign. These operators can avoid processing the expired tuples and just "pass" the necessary time information to upper operators in the pipeline so that upper operators expire the corresponding tuples accordingly.
- **Class 2:** The second class of window operators includes the operators in which processing an expired tuple is different from processing the corresponding positive tuple. Example operators belonging to this class include: Intersection, Minus, Distinct, and Aggregates. Processing an expired tuple in this class may result in producing output tuples (positive or negative) even if the corresponding positive tuple did not produce any outputs. The operators in this class *must* perform complete processing of every expired tuple. One interesting observation is that most of the operators in this class are stateful operators, which means that the operator's state has a copy of every input tuple that has not expired yet. For such operators, it suffices to notify the operator of the necessary time information and the operator reads the expired tuples from the operator's state.

5.2 The "Time-message" Optimization

The goal of the "time-message" optimization is to reduce the overhead of processing negative tuples in Class-1 operators (especially Join) without affecting the output delay. Mainly, when a

Class-1 operator receives a negative tuple (or a tuple expires from the operator's state), instead of processing the tuple, the operator performs the following: (1) Delete the corresponding tuple from the operator's state (if any), and (2) Set a special flag in this tuple indicating that this tuple is a time-message and produce the tuple as output (an example demonstrating the time-message approach is given later in Section 5.3). The time-message tuple can be regarded as a special kind of heartbeat that is generated when a tuple expires.

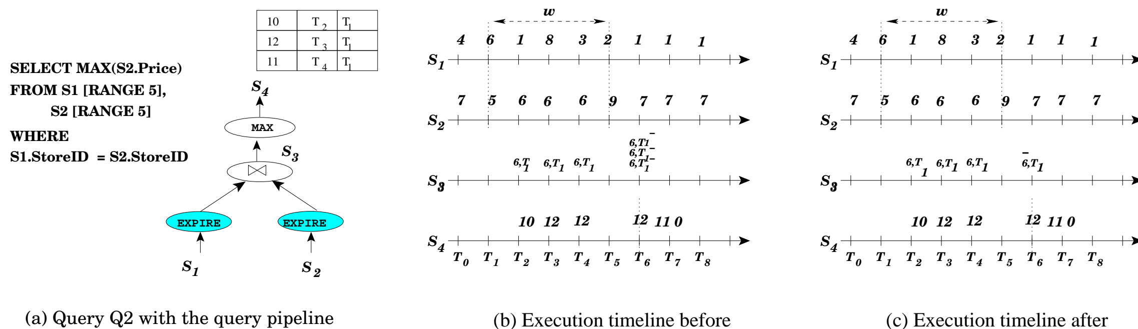
One problem in the time-message optimization as described is that if an operator sends a time-message for every expired tuple, then unnecessary messages may be sent even if their corresponding positive tuples have not produced any outputs before. This happens when, for example, the join filter is highly selective (i.e., when most of the input tuples do not produce join outputs). Filtering operators (e.g., Select and Join) are the source for unnecessary time-messages. Avoiding the unnecessary time-messages in the join operator is addressed in the next section (Section 5.2.1).

Avoiding the unnecessary time-messages in Select is achieved by merging the Select and `EXPIRE` operators into one operator. Mainly, in our implementation, Project and Select are merged into one operator. Moreover, Select is pushed down and is merged with the `Expire` operator. By pushing the selection into the `EXPIRE` operator, we achieve the following: (1) Reducing the size of the `EXPIRE` state since only tuples satisfying the selection predicate are stored, and (2) Producing negative tuples only for tuples satisfying the selection predicate. This means that Select generates exact negative tuples (and not just time-messages) without the overhead of re-applying the selection predicate.

Union is not a filtering operator and hence Union is not a source of unnecessary time-messages. Moreover, negative tuples do not encounter processing overhead in Union. These observations lead us to the fact that Join is the only Class-1 operator that uses and benefits from the time-message optimization. In the rest of the paper, we will use the terms "time-message" and "join-message" interchangeably.

5.2.1 Time-messages in the Join Operator

The join operator is the most expensive operator in the query pipeline. Without the time-message optimization, Join would normally reprocess negative tuples in the same way as their corresponding positive tuples. Given the fact that a negative tuple joins with the same tuples as the corresponding



(a) Query Q2 with the query pipeline (b) Execution timeline before (c) Execution timeline after

Figure 4: The Join-message Technique.

positive tuple and the high cost of the join operation, the time-message technique aims to avoid re-executing the join with the negative tuples. To achieve this, the join operator keeps some state information to avoid unnecessary messages.

Algorithm and Data Structures: Upon receiving a positive tuple t, the join operator inserts t in the join state and joins the tuple with the other input(s). In addition to processing t, the join operator keeps some information with t in the state to indicate whether t has produced join results or not. Upon receiving a negative tuple, instead of re-performing the join operation, the time-message optimization performs the following steps: (1) Removes the corresponding positive tuple from the join state, (2) Checks whether the corresponding positive tuple has produced join results before, (3) If join results were produced, the join operation sets a flag in this tuple indicating that this tuple is a time-message and produces the message as output. The information to be kept with every positive tuple depends on the type of the join operator as described below.

Joining a stream with a table: In this case, only stream tuples will have negative counterparts. To process the negative tuples efficiently, the join operator keeps a table (Joined Tuples Table, JTT) in a sorted list (sorted on the timestamp). When a positive tuple produces join results, the expiration timestamp of this positive tuple is entered in JTT. Only one copy of the expiration timestamp is entered in JTT even if more than one tuple have the same expiration timestamp. At most, the size of this table is equal to the window size. When a negative tuple is to be processed, the join operator checks whether there is an expiration timestamp in JTT that is equal to the expired tuple timestamp. If found, then a time-message is sent and the corresponding timestamp is removed from JTT. Note that only one time-message is produced for every timestamp value. If the tuple timestamp is not in JTT then the negative tuple is simply ignored. Notice that a

join-message is more beneficial in the case when a stream tuple joins with more than one tuple or when more than one tuple has the same expiration timestamp.

Joining two streams: When the join operator joins two tuples t_i^+ from S_1 and t_j^+ from S_2, the resulting tuple t^+ should expire whenever either t_i^+ or t_j^+ expire. Assume that t_i^+ expires first. To expire, t^+, only the join-message for t_i^+ is needed. To avoid unnecessary join-messages, a reference count will be kept with every tuple t_x in the corresponding hash table in the join state. This reference count indicates the number of output tuples that expire when t_x expires. The reference count of a tuple t_x is incremented by one when tuple t_x joins with tuple t_y and t_x has the minimum timestamp. When the join operator is scheduled and a negative tuple is to be processed, the corresponding positive tuple is deleted from the hash table and the reference count associated with it is checked, if greater than zero then a join-message for this tuple is emitted. The pseudocode for the join operator after adding the reference count is given in Algorithm 1. Figure 5 gives an example on the reference count. When the join operator joins tuple t_i from Stream S_1 (with timestamp T_1) with tuple t_j from Stream S_2 (with timestamp T_3), the join operator increments the reference count of t_i. At time T_6, tuple t_i from S_1 expires. Since the reference count of t_i is one then a join-message will be sent. No messages will be sent when t_j expires since t_j's reference count is zero.

Note that one time-message is produced for all input tuples that have the same expiration timestamp. The join operator avoids producing time-messages with the same timestamp by keeping the timestamp of the last emitted join-message in a variable, termed $lastTM$. Before producing another time-message with time $currentTM$, the join operator checks the value of $lastTM$. If $currentTM$ is greater then $lastTM$ then the current time-message is emitted and the value of $lastTM$ is set to $currentTM$, otherwise, the current message is ignored.

5.3 Processing Time-messages

When an operator receives a negative tuple with the time-message flag set, the operator learns that all positive tuples that have expiration timestamps equal to the message's timestamp are expired and acts accordingly. This can be achieved in the same way as expiring tuples in ITA, i.e., by scanning the operator's state and expiring all tuples that carry the same expiration timestamp (Ets) as that of the join-message. If the operator's state is sorted on the Ets attribute of the tuples, then this scan should not be costly. Non-stateful Class-1 operators (e.g., Union) just

pass the time-message to the output. As will be discussed in the next section, the time-message optimization imposes an additional memory overhead for non-stateful Class-2 operators.

The join-message optimization is designed with two goals in mind: (1) Reduce the work performed by the join operator when processing a negative tuple, and (2) Reduce the number of negative tuples emitted by the join operator. Note that the join-message achieves its goals as follows: (1) Negative tuples are "*passed*" through the join operator without probing the other hash table(s). (2) Only one message is emitted for every processed negative tuple independent from its join multiplicity. Moreover, one join-message is emitted for tuples having similar expiration timestamps. A large number of negative tuples can be avoided in the case of one-to-many and many-to-many join operations, which are common in stream applications, for example, in on-line auction monitoring [26].

Example: Figure 4 gives an example of the join-message approach. Figure 4a is the query pipeline. Two input streams $S1$ and $S2$ are joined. Both streams have the same input schema: <ItemId, Price, StoreID>. The sliding windows for the two streams are of the same size and are equal to five time units each. In the figure, the table beside the MAX operator gives MAX's state. The table consists of three columns: the first column is for the value used in the MAX aggregation (S2.Price), and the second column is for the tuple timestamp and the third column is for the tuple expiration timestamp (other attributes may be stored in the state but are omitted for clarity of the discussion). Figure 4b gives the tuples in the pipeline when using NTA and before applying the join-message optimization. The values on the lines represent the joining attribute (StoreID). Figure 4c gives the tuples in the query pipeline after applying the join-message optimization. A tuple with joining attribute value 6^+ arrives at S_1 at time T_1. Three subsequent tuples from S_2 (at times T_2, T_3 and T_4) join with the tuple 6^+ (at time T_1) from S_1. The output of the join has an expiration timestamp equals to that of the tuple that expires first from the two joining tuples. In this example, the output of the join carries expiration timestamp T_1. At time T_6, tuple 6^+ from S_1 expires. In NTA (Figure 4b), JOIN will perform the join with tuple 6^- and output three negative tuples. The three tuples are processed by MAX independently. As mentioned in Section 4.6, MAX will output a new output after processing each input tuple (positive or negative). When applying the join-message optimization, (Figure 4c), the join operator sends a join-message with timestamp T_1 to its output queue. Upon receiving the join-message, MAX scans its state and expires all tuples

Algorithm 1 The Modified W-Join Algorithm
Input: *t_i : Incoming tuple from stream S_i. H_1, H_2: Hash tables for S_1 and S_2 represent the join operator state.*
Algorithm

1) If t_i is a positive tuple
2) B_x = hash(t_i)
3) Insert t_i in the bucket B_x in the hash table H_i
4) For each tuple t_j in bucket B_x in the other hash table
5) If t_j joins with t_i
6) output a positive join output tuple t^+ for (t_i and t_j) with:
7) $t^+.ts = max(t_i.ts, t_j.ts)$
8) $t^+.Ets = min(t_i.Ets, t_j.Ets)$
9) If ($t_j.Ets < t_i.Ets$)
10) Increment reference count of t_j by one
11) Else Increment reference count of t_i by one
12) Else if t_i is an expired tuple
13) B_x = hash(t_i)
14) Delete the tuple t_i from the bucket B_x
15) If reference count of $t_i > 0$
16) if $t_i.ts > lastTM$
17) $lastTM = t_i.ts$
18) Send a join-message with timestamp = $t_i.ts$

with expiration timestamp T_1 and produces a new output after processing each expired tuple.

5.4 Discussion

As can be seen from the previous example and explanations, the join-message optimization reduces the CPU cost of negative tuples in the join operator. On the other hand, the join-message optimization encounters a little additional memory overhead. The memory overhead is due to the reference counter that is kept with tuples in the join state. The reference counter is an integer and its size can be neglected in comparison with the tuple size. Moreover, the memory overhead is offset by the great savings in CPU by avoiding the re-execution of the join for negative tuples.

The join-message optimization does not encounter memory overhead for the operator above the join if this operator is stateful (e.g., Join or Distinct). The memory overhead of the join-message optimization is worth considering only when the join operator is followed by a non-stateful Class-2 operator (i.e., the subtractable aggregates: Sum, Count, and Average). Unlike NTA, when the join-message optimization is applied, these aggregates have to store the input tuples in a state. But, as will be discussed next, for high input rates, NTA gives very high output delays due to

tuples flooding the pipeline. Based on these observations, the decision on whether to use the join-message optimization or the basic NTA with these aggregate queries involves a compromise among memory, CPU, and output delay. The decision should be based on the available resources and the characteristics of the input stream.

6 The Piggybacking Approach

As described in Section 3.2, the main motivation behind NTA is to avoid the output delay that is incurred in ITA. The output delay comes from either the low arrival rate or highly selective operators (e.g., Join and Select). Thus, in the case of high arrival rates and non-selective operators, the overhead of having negative tuples is unjustified. In fact, in these cases, ITA is preferable over NTA. In many cases, data stream sources may suffer from fluctuations in data arrival, especially in unpredictable, slow, or bursty network traffic (e.g., see [29]). In addition, due to the streaming nature of the input, data distribution is unpredictable. Hence, it is difficult to have a model for operator selectivity [22].

In this section, we present the piggybacking approach for efficient pipelined execution of sliding-window queries. The goal of the piggybacking optimization is to *always* achieve the minimum possible output delay *independent* from the input stream characteristics. This goal is achieved by dynamically adapting the pipeline as the characteristics of the input stream change.

In the piggybacking approach, time-messages and/or negative tuples flow in the query pipeline only when they are needed. The main idea of the piggybacking optimization is to reduce the number of tuples in the pipeline by merging multiple negative tuples and/or time-messages into one time-message. Moreover, positive tuples are piggybacked with the time-messages if they co-exist in a queue. By reducing the number of tuples in the pipeline, we also reduce the memory occupied by the queues between the operators and reduce the cost of inserting and reading tuples from queues. A similar notion of piggybacking is used in [2] to reduce the memory needed to process a query. The piggybacking optimization is realized by changing the queue insertion operation such that, at any time, the queue will include at most one time-message. The piggybacking approach works in two stages as follows:

Producing a piggybacking flag. When an operator produces an output tuple t (either

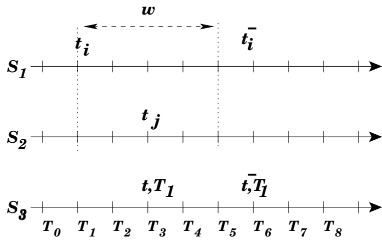


Figure 5: Reference Count Example.

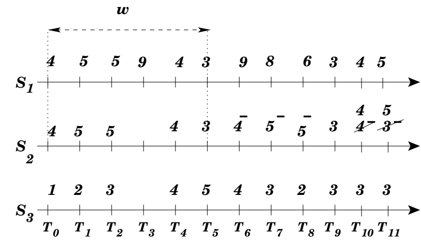
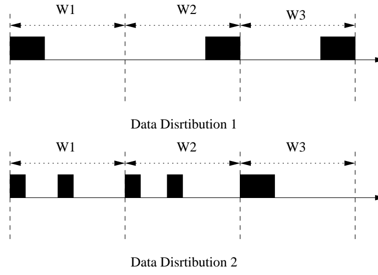


Figure 6: The Piggybacking Approach.

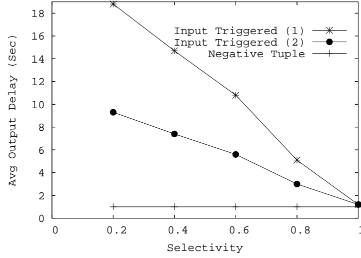
positive, negative, or time-message) in the output queue, the insertion operation of the queue works as follows: first checks if there are any time-messages in the queue (which is the input queue of the next operator in the pipeline). If there is at least one time-message, the insertion operation performs two actions: (1) The output tuple t is tagged by a special flag $PGFlag$, (2) All the time-messages in the output queue are purged. The timestamp of the tagged tuple is a time-message that is used in the second stage to direct the execution of the pipelined query operators. Notice that (1) only time-messages are purged from the queue but invalid tuples remain, (2) at any time, the queue will include at most one time-message, and (3) the time-message is the bottom most tuple in the queue.

Processing the piggybacking flag. When a query operator receives a tuple t (either positive, negative, or time-message) at time T, it checks for the $PGFlag$ in t. If the input tuple is not tagged by the piggybacking flag, the query operator will act exactly as NTA and the time-message optimization. However, if the incoming tuple is tagged by the piggybacking flag, the query operator acts as ITA, described in Section 3.1. This means that all tuples stored in the operator state with expiration timestamp less than or equal T should expire. The idea is, if there are many positive tuples, then there is no need to communicate explicit time-messages in the pipeline. In the case that processing the incoming tuple t does not result in any output (e.g., filtered with the Join), we output a time-message that contains only the timestamp and the piggybacking flag so that operators higher in the pipeline behave accordingly.

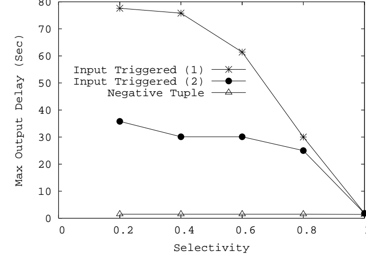
The piggybacking flag ($PGFlag$) is a generalization of the time-message, described in Section 5.2.1. The main difference is that a time-message with timestamp T is responsible for expiring tuples with expiration timestamp T, while a $PGFlag$ with timestamp T is responsible for expiring all the tuples with expiration timestamps *less than or equal* to T.



(a) Data Distribution



(b) Avg Output Delay



(c) Max Output Delay

Figure 7: Q1: Effect of Selectivity and Data Distribution.

Example: Figure 6 gives an example on the piggybacking approach. This example uses the same query of Figure 2a. The example shows that when JOIN is highly selective (in the period T_6 to T_8) negative tuples are passed to COUNT for immediate expiration of tuples with values 4, 5, and 5. At time T_{10}, JOIN emits tuple 4^- immediately followed by tuple 4^+. If tuple 4^+ is emitted before COUNT reads 4^-, then 4^+ will delete 4^- from the queue and COUNT will read only tuple 4^+. While processing 4^+, COUNT checks the input tuple's (4^+) timestamp and knows that a tuple with value 4 (that is stored in COUNT's state) should expire. Then, COUNT emits the new answer reflecting the expiration of 4 and the addition of 4. The same happens at time T_{11}. This example shows that the answer update will have the minimum possible delay.

The piggybacking approach is designed with the following goal in mind: "always achieve the minimum possible output delay independent from the input stream or query characteristics". This goal is achieved as follows: (1) the time information is propagated (using time-messages) in the pipeline once they are generated without waiting for positive tuples, and (2) the time information is merged with the positive tuples whenever possible. Basically, the piggybacking optimization self-tunes the query pipeline by alternating between both NTA and ITA.

6.1 Discussion

In our prototype, operators in the pipeline are scheduled using the round-robin approach (RR). In RR, an operator runs for a fixed amount of time before releasing the CPU to the next operator. During an operator run, the operator processes tuples from the operator's input queue and produces tuples in the operator's output queue. The piggybacking approach results in minimizing the number of tuples produced in the output queue during an operator's run since time-messages are merged together or merged with positive tuples. This reduction in queue sizes has the benefit of reducing the memory usage by the pipeline and reducing the overhead of reading tuples from the queue.

There are several other operator scheduling techniques, e.g., FIFO, chain [7] and train [11]. The reduction in the queue size gained by using the piggybacking approach depends on which scheduling policy is used. For example, if the FIFO scheduling is used, then the piggybacking optimization does not provide any performance gains over NTA. This is because in the FIFO scheduling, one tuple is processed in the pipeline at a time and tuples are not accumulating in the intermediate queues. On the other hand, for scheduling policies that allow tuples to accumulate in the output queues (e.g., RR, chain, or train), the piggybacking optimization achieves performance gains over NTA. In other words, the piggybacking optimization is orthogonal to the scheduling policy. Under all scheduling policies, in the worst case, the piggybacking approach performs the same as NTA.

7 Experiments

In this section, we present experimental results from the implementation of our algorithms in a prototype data stream management system, Nile [20]. We compare the performance of NTA with ITA and show how the proposed optimizations enhance the performance further.

7.1 Experimental Setup

The prototype system is implemented on Intel Pentium 4 CPU 2.4 GHz with 512 MB RAM running Windows XP. The system uses the pipeline query execution model for processing queries over data streams. The query execution pipeline is connected with the underlying streaming source via the stream scan operator SSCAN. The `EXPIRE` operator is implemented as part of the SSCAN

operator. The local selection predicates for each stream are pushed inside the `EXPIRE` operator. Different operators in the pipeline communicate with each other via a network of FIFO queues. Tuples are tagged with a special flag to indicate whether the tuple is positive, negative, or invalid. Each operator in the pipeline runs as an independent thread. Operators in Nile are scheduled using a round-robin scheduling where each operator runs for a fixed amount of time to consume tuples from the operator's input queue. Once the input queue of the operator is exhausted or the operator's time slot is finished, the next operator is scheduled.

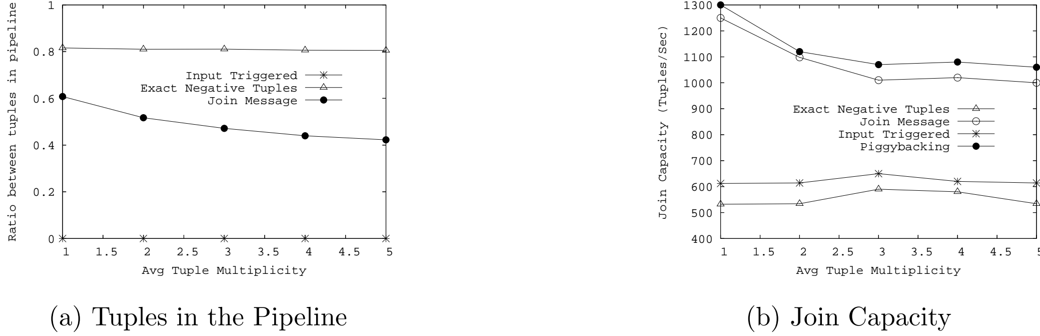
We use the average and max output delay as a measure of performance. The output delay is defined as the delay between the arrival/expiration of a tuple and the appearance of its effect in the query answer. For example, assume that in Q1 (Figure 2), a tuple $t1$ arrives to the system at time T. COUNT produces an output tuple after adding the value of $t1$ at time $T + d$, then this tuple encounters an output delay of d units of time.

Workload queries: We use the two queries, Q1 (Figure 2) and Q2 (Figure 4) to evaluate the proposed techniques. The stream *SalesStream* used in the queries has the same following schema: (StoreID, ItemID, Price, Quantity, Timestamp). We use randomly generated synthetic data. The inter-arrival time between two data items follows the exponential distribution with mean λ tuples/second. The arrival rate of the input streams is changed by varying the parameter λ of the exponential distribution. A timestamp is assigned to a tuple when the tuple arrives to the server.

Synthetic data generation: For the input streams, the number of distinct items is set to 1200. For Query Q1, the table FavoriteItems is changed to achieve the desired selectivity. The distribution of the data items inside the window is randomly generated (if not mentioned otherwise). For Query Q2, we achieve the desired join selectivity by controlling the values of the join attribute (StoreID). For example, if the window size is set such that the window will contain 100 tuples, then the StoreID values in the first stream are randomly generated in the range 1 to 100 and in the second stream in the range 50 to 150. Such data distribution guarantees a selectivity of 0.005 for all windows.

7.2 ITA vs. NTA

In this section, we compare the performance of ITA and NTA for various data distributions. Figure 7 gives the effect of changing the selectivity of JOIN in Q1 (Figure 2a). Figure 7b gives the average output delay while Figure 7c gives the maximum output delay. We run the experiment for two data



(a) Tuples in the Pipeline

(b) Join Capacity

Figure 8: Effect of the Join-message.

distributions as shown in Figure 7a. In this experiment, the input rate is fixed at 50 tuples/second, the window size is 30 seconds and the selectivity varies from 0.1 to 1. For the same selectivity value, the data distribution in Figure 7a shows how the qualified tuples are distributed in the window. In Data Distribution 1, the qualified tuples are accumulated at one end of the window and some windows may not have any qualified tuples. On the other hand, in Data Distribution 2 the qualified tuples are scattered along the window width. The experiment shows that the output delay in ITA is highly affected by the selectivity and the data distribution. For low selectivity, ITA shows high output delay since COUNT will not expire old tuples until a new input tuple qualifies the join. The output delay for ITA is higher in the case of Data Distribution 1 because the range between two qualified tuples is bigger than that in Data Distribution 2. The output delay for ITA decreases considerably when either the selectivity increases or when tuples are scattered in the window since qualified tuples pass the join and COUNT is scheduled more often. In general, the output delay in the case of ITA is unpredictable and is highly affected by the input characteristics. The experiment also shows that NTA does not depend on the selectivity or data distribution since tuple expiration takes place even if no input tuples pass the join. As the input characteristics in streaming environments are always changing, ITA is not suitable to use. In the rest of the experiments we omit ITA.

7.3 The Join-message Optimization

Figure 8 illustrates how the join-message optimization reduces the overhead of processing negative tuples. This experiment uses Query Q2 (Figure 4a). The input rate is 50 tuples/second for each

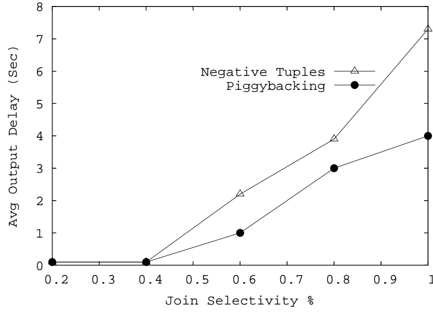
stream. The window is 30 seconds and the join selectivity is fixed to 0.01. The tuple's join multiplicity ranges from 1 to 5. To understand how to get different tuple multiplicity for the same join selectivity, assume the number of tuples in each window is 100, then for a join selectivity of 0.01, 100 tuples will be output from the join in each window (100/100*100). The 100 output tuples can result if 100 tuples from the first stream each joining with one tuple from the second stream (i.e., tuple multiplicity equals to 1). The 100 output tuples can also result if 50 tuples from the first stream each joining with 2 tuples from the second stream (i.e., tuple multiplicity equals to 2).

Figure 8a gives the ratio between the number of negative and positive tuples in the join output queue. The number of tuples in the queue is an indication about memory usage by the queue. Also, the number of negative tuples represents the overhead associated with NTA. This overhead is always zero for ITA. The overhead is almost equal to one in NTA since one negative tuple is processed for every positive tuple (in the figure, it is not exactly one since some negative tuples may have not been processed yet at the time the measurement is taken). The join-message optimization reduces the number of negative tuples emitted from the join operator to the next operator in the pipeline (MAX). The reduction increases as the tuple join multiplicity increases. Figure 8b gives the average join capacity. The join capacity is defined as the number of tuples processed by the join operator per second. Figure 8b shows that: (1) the join capacity is almost the same for NTA and ITA because of the exact processing of negative tuples, and (2) the join capacity is doubled when using the join-message optimization (shown in the join message and piggybacking lines in Figure 8b) because the negative tuples do not perform the exact join.

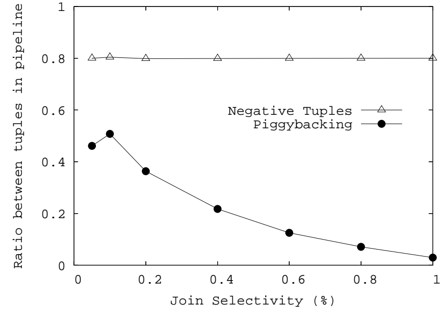
Figure 8b illustrates that the join capacity is independent of tuple multiplicity. In the symmetric hash join between two streams $S1$ and $S2$, an input tuple from $S1$ probes only one bucket in the hash table for $S2$. The probing cost is negligible compared to the cost of performing the join and constructing the output tuple. The join capacity is independent of the tuple multiplicity because the join selectivity is fixed and the number of output tuples is independent of the tuple multiplicity.

7.4 The Piggybacking Approach

This section shows the performance of the piggybacking optimization (accompanied by the join-message optimization). Implementing the piggybacking approach requires only a slight modification to the implementation of the queues connecting operators in the pipeline (as described in Section 6).



(a) Performance Enhancement



(b) Overhead Reduction

Figure 9: Performance of Piggybacking.

7.4.1 Performance Enhancement

Figure 9a compares the output delay of NTA and the piggybacking approach for Query Q2 (Figure 4a). The input rate is fixed to 200 tuples per second while varying the join selectivities from 0 to 1%. The figure illustrates that for lower selectivity, which corresponds to high output rates from the join operator, NTA encounters more output delays since the queues are flooded with positive and negative tuples. For low selectivity values (which corresponds to lower output rates from the join), NTA and the piggybacking approach give the same output delay since fewer number of tuples flow in the queues and hence there is no waiting time. In general, the piggybacking approach gives the minimum possible output delay in all arrival rates and all selectivities since it communicates the negative tuples only when necessary.

7.4.2 Reducing Overhead

This experiment shows how the piggybacking approach reduces the number of negative tuples in the pipeline. Reducing the number of negative tuples in the pipeline means reducing the memory usage by the queues. Figure 9b gives the ratio between the number of negative tuples and the number of positive tuples processed by the MAX operator in Query Q2. We vary the join selectivity as the input rate is fixed to 200 tuples per second. In NTA, the ratio is almost one since one negative tuple is processed for every positive tuple. In the piggybacking approach, the ratio decreases for lower selectivity. The reason is that positive tuples flow in the query pipeline with high rate and

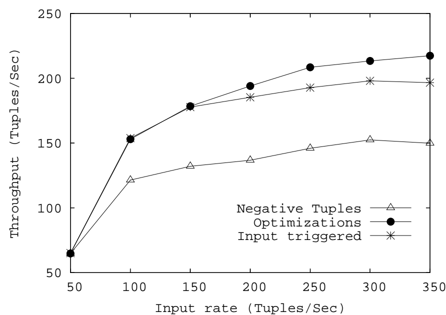


Figure 10: Throughput for Query Q_2.

hence purge negative tuples (if any) from the queue.

7.5 System throughput

In this section we compare the system throughput for Query Q2 (Figure 4a) when using ITA, NTA, and the optimizations (piggybacking + join message optimizations). The system throughput is measured as the number of output tuples produced by the query pipeline per second. Figure 10 gives the throughput when varying the input rate from 50 to 350 tuples per second. Figure 10 illustrates that for small arrival rates (50 tuples/second or less) the three approaches give the same throughput. In the three approaches, the system load is low for small arrival rate and hence all the input tuples can be processed as fast as they arrive. As the input rates increase, NTA encounters more processing overhead than ITA and piggybacking because the queues are flooded with positive and negative tuples. As a result, the throughput of NTA is lower than the throughput of the other approaches. Figure 10 also illustrates that the throughput of the optimizations approach is slightly higher than that of ITA. The reason is that the tuples are processed faster in the optimizations approach due to the join message optimization. Figure 10 illustrates that the system reaches a saturation level at which the number of output tuples is fixed even if the input rate increases. The saturation level corresponds to the maximum number of tuples per second that the pipeline can process. The saturation level of NTA is lower than that of ITA and the optimizations approach.

Notice that the throughput measure does not distinguish between the query processing approaches for low arrival rates. For fluctuating input streams, the maximum output delay can be considered a more illustrative measure since it differentiates the approaches from each other for both low and high arrival rates.

8 Related Work

Stream query processing is currently being addressed in a number of research prototypes. Examples include Aurora [1], which is later extended to Borealis [2], NiagaraCQ [15], TelegraphCQ [12], PSoup [14], NILE [20, 21] and STREAM [3]. These research prototypes address various issues in processing queries over data streams. All these research prototypes have recognized the need for sliding windows to express queries over data streams. For a survey about the requirements for stream query processing, refer to [8, 16].

Window-aware query operators have been addressed many times in the literature. Examples of algorithms for processing window aggregates include [5, 1, 13] and examples of algorithms for window join include [23]. The previous work in this subject addresses the processing of a single window operator but does not address the processing of a whole query pipeline. Aurora [1] uses the window re-evaluation approach to evaluate window aggregates. In the window re-evaluation approach, a computation state is initialized whenever a window is opened, that state is updated whenever a tuple arrives, and the state is converted into a final result when the window closes. An input tuple updates and is stored in more than one computation state in the same time. In this paper we focus on the incremental evaluation pipeline. Incremental evaluation for Join is addressed in [23], where ITA is used to invalidate tuples from the join state when a new tuple arrives. However, the authors in [23] do not address how to expire tuples from the operators above the join. Also, [23] does not address the output delay problem.

The traditional query optimization goal does not apply to continuous queries. Rate-based optimization is introduced in [30]. The goal of the optimization is to maximize the output rate of a query. In [6], the authors introduce a framework for conjunctive query optimization. The goal of the optimization is to find an execution plan that reduces the resource usage. None of these optimization techniques consider reducing the output delay as an optimization goal. Moreover, these optimization frameworks consider only ITA. Applying these optimization frameworks over NTA is an interesting area for future work. The time-message and piggybacking optimizations reduce the CPU and memory utilization of NTA, hence they can be categorized under the class of optimizations to reduce resource utilization. [17] introduces two optimizations to eliminate the overhead of negative tuples. The first optimization re-orders the query plan (i.e., pulls-up the Minus operator and pushes-down the Distinct operator). The second optimization uses a hyprid approach

of both NTA and ITA (referred to as direct). The reordering optimization can be used with our optimizations and is orthogonal to the focus of this research. Moreover, our proposed optimizations avoid the use of the direct approach because of its unpredictable output delay as explained in 3.1.

Recent research efforts focus on introducing new “artificial” kinds of tuples that flow through the query pipeline. Examples of such tuples include *delete messages* [2], *DStream* [4], *Negative Tuples* [21], *heartbeats* [25], and *punctuation* [28]. The main idea of these artificial tuples is to notify various pipelined operators of a certain event (e.g., expiring a tuple, synchronizing operators, or end of sequence of data). STREAM [3] and Nile [20, 21] use NTA to expire tuples. Negative tuples have been used in other systems, e.g., Borealis [2] for automatic data revision where a negative tuple is sent by the streaming source to delete an erroneous positive tuple. Although not mentioned explicitly, NiagaraCQ [15] uses a notion similar to negative tuples when processing stream deletions. All the previous works either uses ITA or NTA. Our work is considered the first to automatically adapt the pipeline to switch between ITA and NTA based on the underlying stream characteristics.

Punctuation is another form of artificial tuples [28]. A punctuation marks the end of a subset of the data and is used to purge state and to unblock blocking operators. Processing stream constraints is another way to discover and purge unneeded tuples from operators’ states [9]. Unlike negative tuples, the tuples purged by the punctuation (or stream constraints) are not re-processed and do not affect the query answer. Moreover, both [28] and [9] assume prior knowledge of the input stream characteristics and utilize this knowledge in generating the appropriate punctuation.

An operator-level heartbeat [25] is a way for time synchronization. A heartbeat is sent along the query pipeline so that the operators learn the current time and process input tuples accordingly. The goal of the heartbeats is to order tuples arrived out-of-order. Heartbeat generation assumes knowledge of the characteristics of the input streams and is independent from the data distribution or the query. The time-message optimization we propose in this paper can be regarded as a special kind of heartbeat that has a different goal and different generation policies than the heartbeats in [25]. Time-messages are generated based on the data distribution and query selectivity and flow in the pipeline only when there are tuples to expire. Moreover, time-messages can be merged with positive tuples. The goal of time-messages is to reduce the output delay of the query.

Processing negative tuples in the query pipeline to update the query answer is closely related to the traditional incremental maintenance of materialized views [19, 10]. The design of our window

operators is based on the differential approach for incremental view maintenance [18] where change propagation equations are designed for the various relational operators [18]. The equations specify how an operator should process an inserted or expired tuple.

9 Conclusions

Incremental query evaluation has been adopted by data stream management systems as a coordination scheme among various pipelined query operators. In this paper, we focus on the two approaches for incremental query evaluation, namely, the input-triggered approach (ITA) and negative tuples approach (NTA). We study the realization of the incremental evaluation pipeline in terms of the design of the incremental relational operators. We show that although NTA avoids the shortcomings of ITA (i.e., large output delays), NTA suffers from a major drawback. Negative tuples double the number of tuples in the query pipeline, hence the pipeline bandwidth is reduced to half. We classified incremental operators into two classes according to whether an operator can avoid the processing of a negative tuple or not. Based on the operator classification, we presented two optimization techniques to enhance the performance of NTA. The first optimization, namely the time-message optimization, mainly focuses on the join operator subtree. The main idea is to avoid the re-execution of the expensive join operation with negative tuples. The second optimization, namely the piggybacking optimization, self-tunes the query pipeline to work in either ITA or NTA according to the characteristics of the tuples flowing in the query pipeline. With the piggybacking approach, the query pipeline gets the benefits of both ITA and NTA. Experimental results based on a real implementation of ITA, NTA, time-messages, and piggybacking approaches inside a prototype data stream management system show that the join-message optimization enhances the performance of negative tuples by a factor of two. Based on the input rate and/or join selectivity, the piggybacking optimization always traces the best performance of either ITA or NTA.

References

[1] D. J. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *The International Journal on Very Large Data Bases, VLDB Journal*, 12(2):120-139, 2003.

[2] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[3] A. Arasu et al. Data-Stream Management: Processing High-Speed Data Streams, chapter STREAM: The Stanford Data Stream Management System, Springer-Verlag, New York 2005.

[4] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *In Proceedings of the International Workshop on Database Programming Languages, DBPL*, 2003.

[5] A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*, 2004.

[6] A. Ayad and J. F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *SIGMOD*, 2004.

[7] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator Scheduling for Memory Minimization in Data Stream Systems. In *SIGMOD*, pages 253–264, 2003.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS*, 2002.

[9] S. Babu, U. Srivastava, and J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Transactions on Database Systems, TODS*, 29(3):545–580, 2004.

[10] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, 1986.

[11] D. Carney, U. Cetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB*, pages 838–849, 2003.

[12] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[13] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.

[14] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *The International Journal on Very Large Data Bases, VLDB Journal*, 12(2):140–156, 2003.

[15] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[16] L. Golab and M. T. Ozsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2), June 2003.

[17] L. Golab and M. T. Ozsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD*, 2005.

[18] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD*, 1995.

[19] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[20] M. A. Hammad et al. Nile: A Query Processing Engine for Data Streams (Demo). In *ICDE*, 2004.

[21] M. A. Hammad, T. M.Ghanem, W. G. Aref, A. K. Elmagarmid, and M. F.Mokbel. Efficient Pipelined Execution of Sliding Window Queries over Data Streams. In *Purdue University Technical Report, CSD TR 03-035*, June 2004.

[22] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *SIGMOD*, 1999.

[23] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating Window Joins over Unbounded Streams. In *ICDE*, 2003.

[24] R. Motwani et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR*, 2003.

[25] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *PODS*, 2004.

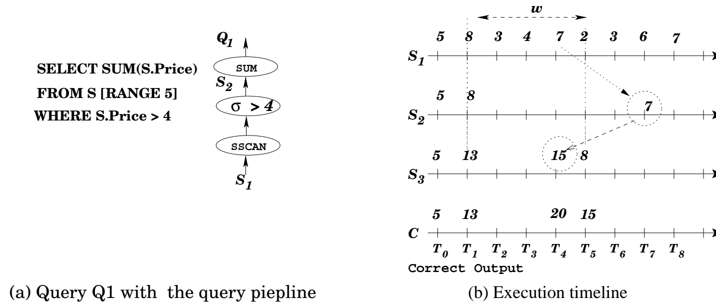
[26] U. Srivastava and J. Widom. Memory-Limited Execution of Windowed Stream Joins. In *VLDB*, 2004.

[27] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, 1992.

[28] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 15(3):555–568, 2003.

[29] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. In *SIGMOD*, 1998.

[30] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, 2003.



(a) Query Q1 with the query piepline (b) Execution timeline

Figure 11: Expiration based on global clock

A Global Clock Approach

This appendix gives an example to show that a query may produce incorrect answers if the operators depend on a global clock to expire tuples. The example in Figure 11 is an aggregate query (SUM) over an input stream S1. Figure 11a gives the query pipeline and Figure 11b gives the execution time line. Stream $S3$ represents the output of the SUM operator while stream C represents the expected output.

In this example, a delay of three clock-ticks takes place between the time that the tuple 7 is received at $S1$ and the time it is received at $S2$. The tuple 7 has a timestamp T_4 which equals the time 7 arrives to $S1$. Due to scheduling and the different operator processing speeds, the tuple 7 does not arrive at the SUM operator until time T_7. If SUM is scheduled between T_5 and T_7, SUM will expire tuple 5 and produce an *incorrect* SUM 8 in $S3$ at time T_5. Moreover, when SUM is scheduled at time T_7 or after, SUM will receive the delayed tuple 7 that has a timestamp T_4. This means that SUM processes and produces tuples in a nondeterministic timestamp order. The negative tuples approach solves this problem because the positive tuple 7 is generated at time T_4 while the negative tuple 5 is generated at time T_5 and the two tuples will arrive to the SUM operator in the correct timestamp order.