Query Processing using Negative Tuples in Stream Query Engines

Thanaa M. Ghanem  
Moustafa A. Hammad  
Mohamed F. Mokbel  
Walid G. Aref  
Ahmed K. Elmagarmid

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{ghanemtm,hammad,mokbel,aref,ake}@cs.purdue.edu

The concept of Negative Tuples (or delete tuples) has been adopted widely in data stream management systems to process continuous sliding window queries efficiently. The main idea is to produce a negative tuple for each expired tuple form the sliding window. Thus, various query operators can update their state based on the expired tuples. Negative tuples avoid the non-deterministic output delays that result from the commonly used input-triggered approach. However, negative tuples double the number of tuples through the query pipeline, thus reducing the pipeline bandwidth. In this paper, we present two optimization techniques that aim to reduce the overhead of the negative tuples approach. The first optimization, namely the join message optimization, is concerned with the join operator subtree where it filters out some of the negative tuples and avoid re-executing the joins on the negative tuples. The second optimization, namely the piggybacking optimization, self-tunes the query pipeline to work in either the input-triggered or negative tuples schemes according to the rate of the tuples flowing in the query pipeline. Both optimizations can be applied independently or together to enhance the performance of negative tuples. A detailed experimental study, based on a prototype system implementation, shows the performance gains of both optimizations over the negative tuples and the input-triggered approaches.

1. INTRODUCTION

The emergence of data streaming applications (e.g., sensor network, stock market, and tickers) calls for new query processing techniques to cope with the high rate and 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 should be updated to reflect both the new tuples entering the window and the old tuples expiring from the window. Geared towards supporting continuous sliding-window queries, data stream management systems (e.g., Aurora [2], Borealis [1], CAPE [26], NiagaraCQ [13], NILE [20], STREAM [25], and TelegraphCQ [10]) provide the so-called window-aware operators. Window-aware operators (e.g., window-join and window-aggregates) are modifications of their counterpart traditional operators to support sliding-window queries. The main difference in window-aware query operators is the need to process tuples expired from the window as well as the newly incoming tuples.

Window-aware query operators have relied on the input-triggered approach where the timestamp of the newly incoming tuples is used to expire old tuples [23, 5]. However, as will be discussed in Section 4.1, the input-triggered approach may result in significant delays in the query answer. As an alternative, the negative tuples approach was introduced as a delay-based optimization framework that aims at reducing the output-delay incurred by the input-triggered approach [4, 20]. A negative tuple is an artificial tuple that is generated by an operator EXPIRE in the query pipeline (generalization for operator SEQ-WINDOW in [4] and operator W-EXPIRE in [20]). For each regular data tuple (i.e., positive tuple), say t, EXPIRE emits a corresponding negative tuple t− once t becomes out of the sliding window. As the negative tuple flows through the query pipeline, it removes the effect of its corresponding positive tuple.

Although the basic idea of the negative tuples approach is attractive, it may not be practical. The fact that we introduce a negative tuple for every expired positive tuple means doubling the number of tuples through the query pipeline. Each negative tuple exactly repeats and undoes the effect of its corresponding positive tuple. In this case, the overhead of processing tuples through the various query operators is doubled. This observation opens the room for optimization methods to the basic negative tuples approach. Various optimizations would mainly focus on two issues: (1) Reducing the overhead of re-processing the negative tuples. (2) Reducing the number of the negative tuples through the query pipeline.

In this paper, we put the negative tuples under a magnifying glass where we show its detailed realization in terms of its generation and its processing in various operators. We show how in some cases the negative tuples improves the query answer. Then, we introduce the join message optimization that reduces the overhead of processing the negative tuples in the join operator. Furthermore, we introduce the piggybacking approach that self-tunes the query pipeline by alternating between the usage of input-triggered and negative tuples approaches. Alternating between the two approaches is triggered by discovering fluctuations in the input arrival rate that is likely to take place in streaming environments. In general, the contributions of this paper can be summarized as follows:

1. We compare the performance of the input-triggered
and negative tuples approaches for different queries. This comparison helps to identify the appropriate situations to use each approach.

2. We introduce the \textit{join message} optimization technique that aims to reduce the number of negative tuples through the query pipeline.

3. We introduce the \textit{piggybacking} technique that aims to self-tune the query processing in data stream management systems by alternating between the \textit{input-triggered} and \textit{negative tuples} approaches. This self-tuning features allows the system to be stable with fluctuations in input arrival rates and filter selectivity.

4. We provide an extensive experimental study using a prototype data stream management system that evaluates the performance of the \textit{input-triggered, negative tuples, join messages, and piggybacking} techniques.

The rest of the paper is organized as follows: Section 2 highlights related work in data stream query processing. Section 3 gives the necessary background on the pipeline query execution model. Section 4 describes and compares the input-triggered and the negative tuples approaches for pipelined execution of sliding window queries. Optimizations over the basic \textit{negative tuples} approach are introduced in Section 5. Section 6 studies the \textit{piggybacking} technique. Experimental results are presented in Section 7. Finally, Section 8 concludes the paper.

2. RELATED WORK

Stream query processing is currently being addressed in a number of research prototypes. Examples include Aurora [2, 9], which is later extended to Borealis [1], NiagaraCQ [13], TelegraphCQ [10], PSoup [12], NILE [20] and STREAM [25]. 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 [7, 16].

Window-aware query operators have been addressed many times in the literature. The previous work in this subject addresses the processing of a single window-aware operator but does not address the processing of a whole query pipeline. In [19], the authors introduce classes of time window joins where the window constraints are different among different pairs of input streams. In [23], the authors use the input-triggered approach to invalidate tuples from the join state when a new tuple arrives based on the window semantics. However, they do not address how to invalidate tuples for the operators above the join since an output from the join carries the semantics of two different windows. Multi-way window join is introduced in [17]. Memory-limited execution of window join is addressed in [27]. Window aggregates are another important class of operators [11, 22, 5]. Indexing sliding windows over data streams has been addressed in [15].

The traditional query optimization goal is to reduce the query execution time. This goal does not apply to continuous queries over data streams. Optimization techniques over data streams have different goals. 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. Query optimization using content-based routing is introduced in [7]. The goal of the optimization is to reduce the query execution time. None of these optimization techniques use reducing the output delay as an optimization goal.

Since the characteristics of a data stream change continuously, adaptive query processing is necessary. Adaptive continuous queries are addressed in [24]. [31] introduces algorithms for dynamic plan migration for plans with stateful operators. Constraint-aware adaptive query processing is introduced in [26].

Recent research efforts focus on introducing new "artificial" kinds of tuples that flow through the query pipeline. Examples of such tuples include delete messages [1], DStream [4], Negative Tuples [20], heartbeats [27], and punctuation [28]. The main idea of these artificial tuples is to notify various pipelined operators with a certain event (e.g., expiring a tuple, synchronizing operators, or end of sequence of data). STREAM [3] and Nile [20] use the negative tuples approach to expire tuples from window-aware operators. Negative tuples have been used in other systems e.g., Borealis data stream management system [1] for automatic data revision. A negative tuple is sent by the streaming source to delete an erroneous positive tuple. Although, not mentioned explicitly, NiagaraCQ [13] system uses a notion similar to negative tuples while they are processing deletions to data streams. The work on punctuating data streams [28] is related to the negative tuple approach. A punctuation marks the end of a subset of the data. The difference between negative tuples and punctuation is that a negative tuple indicates the expiration of one tuple while a punctuation indicates the expiration for a set of tuples. Joining punctuated data streams has been addressed in [14]. Processing stream constraints is another way to discover expired tuples [8]. Stream constraints are generated using stream monitoring and stream summarizations. An operator-level heartbeat [27] is a way for time synchronization since a heartbeat is sent along the query pipeline so that the operators learn the current time and process input tuples accordingly. Heartbeats are processed only when there is an input in the operator’s input queue and hence is input-triggered and does not address the output delay problem.

3. DATA STREAM QUEUING MODEL

Data stream management systems use a pipelined queuing model for the coordination among different query operators [4]. The queuing model of data streams is a modification of the one used in traditional database management systems [18]. All query operators are connected via first-in-first-out queues. Each operator, say \( p \), has its own input queue \( I_p \) that receives the incoming input tuples to the operator \( p \). \( p \) is scheduled once there is at least one data input in its input queue \( I_p \). Upon scheduling, \( p \) processes its input and produces an output result. The output of \( p \) is either sent as a query result to the user (if \( p \) is the top operator in the pipeline) or is inserted in \( p \)’s output queue, which is the input queue of the next operator in the pipeline.

The bottom stream operator in any continuous query pipeline over data streams is the SCAN operator. The
stream SCAN (SSCAN) operator acts as an interface between the streaming source and the query pipeline. SSCAN is responsible for reading tuples from the streaming source. The newly incoming tuples are inserted into the input queue of the first operator in the pipeline. Each tuple is assigned a timestamp that indicates the current time. No two tuples from the same stream can have the same timestamp. Incoming tuples are processed in increasing order of their timestamps. Figure 1 gives an SQL query and the corresponding pipelined query plan. Notice that queue $I_3$ is both the input queue for the SUM operator and the output queue for the join operator. Some query operators (e.g., join, aggregates, and distinct) are required to keep some state information. Basically, in continuous sliding window queries, such operators need to keep track of all tuples in the current window $w$. However, the maintenance of up-to-date state information is challenging.

Traditional pipelined query operators are designed mainly to deal with data resident on disk. To cope with the streaming queuing model, traditional pipelined query operators should be modified (e.g., see [23]). In addition to processing the new incoming tuples, windowed-pipelined query operators need to address three main challenges:

1. Updating the state information of each query operator. State information needs to be updated by expiring (i.e., deleting) old tuples that become outside of the sliding window $w$ as well as adding the new incoming tuples. The challenge is how to trigger the change of status. An operator may continuously check the time to decide about expiring some data. However, continuously checking the time reduces the CPU utilization.

2. The action to be taken when a tuple expires. Once an operator decides to expire (delete) one of the input tuples from the operator’s state, the query operator needs to know what to do with the expired tuples. The required action could be as simple as just dropping the expired tuple, or as complicated as reprocessing the expired tuple and producing a new answer.

3. Forwarding the effect of the expired tuples to the next operator in the query pipeline.

Query operators handle these three issues differently depending on the operator’s semantics. For correct and efficient execution of the query, window-aware query operators should be designed to address these three challenges efficiently.

4. PIPELINED-EXECUTION OF SLIDING-WINDOW QUERIES

In this section we discuss the two approaches for pipelined execution of sliding-window queries, namely input-triggered and negative tuples approaches. We discuss how each approach addresses the three challenges presented in Section 3 along with the drawbacks of each approach.

4.1 The Input-Triggered Approach

The main idea of the commonly used input-triggered approach is to schedule different query operators only when they have input tuples in their input queues. Thus, an incoming input tuple triggers the processing of a query operator. Once an operator starts to process an input tuple, it knows the current time $T$ at the streaming source from the tuple’s timestamp. Processing a new input tuple includes two main tasks: (1) expiring tuples based on the current time, the query operator can decide which of the tuples in its state information (if any) are expired. This can be performed by scanning all data tuples in the operator’s state and removing any tuples with timestamp less than $T - w$ where $w$ is the length of time window in the continuous sliding-window query. (2) processing the new tuple itself.

The action to be taken for expired tuples is operator dependent. For example, aggregate operators (e.g., SUM and COUNT) need to process the expired tuples as well as the new incoming tuples. For example, in the COUNT operator, the number of expired tuples should be subtracted from the answer and the new answer should be reported. The output of the operator will carry the current time information so that the next operator in the pipeline can behave accordingly.

The input-triggered approach has a major flaw where it can result in non-deterministic delay in the query answer. 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. The delay in the query answer comes form the fact that unlike traditional continuous queries where the query answer changes only with the arrival of new input data, the answer of continuous time-based sliding window queries depends on the query time as well as on the input to the query. By being time-dependent, the answer of a sliding-window query may change even if no new input arrives to the outstanding query. This is because some tuples may expire.

For example, consider the query $Q_1$ “Continuously, report the total sales of items with price greater than 4 in the last hour”. Even if there are no sales for a certain time period $T$, the query answer may change because some parts of the answer become too old (i.e., more than one hour old). The same problem may arise if the query includes a highly selective operator (e.g., select or join) at the bottom of the query pipeline. In this case even if the input is continuously arriving, the filtering operator will filter out many of the incoming stream tuples and hence the upper operators in the pipeline will not get any notification about the arrival of a new tuple. Thus, the query answer will not be updated until one tuple passes the filter.

Figure 2 illustrates the behavior of the input-triggered
approach for \(Q_1\). The time lines \(S_1\) and \(S_2\) correspond to the input stream and the output of the selection operator, respectively. \(Q_1\) and \(C\) represent the output stream using the input-triggered approach and the correct output, respectively. The window \(w\) is represented as the last five tuples. Up to time \(T_3\), \(Q_1\) matches the correct output \(C\) with the result 28. At \(T_4\), the input “2” in \(S_1\) does not pass the selection filter. Thus, the SUM operator will not be scheduled to update its result. Thus, \(Q_1\) will remain 28 although the correct output \(C\) should be 22 due to the expiration of the value 6. Similarly, at \(T_5\), \(Q_1\) is still 28 while \(C\) is 13 (the tuple with value “9” has expired). \(Q_1\) keeps having an erroneous output till tuple “6” is received where it triggers the scheduling of the SUM operator to produce the correct output “14”. This erroneous behavior motivates the idea of having a new technique that triggers the query operators either based on input change or time change.

### 4.2 The Negative Tuples Approach

The main goal of the negative tuples approach is to separate between tuple expiration and the arrival of new input tuples. The main idea is to introduce a new type of tuples, namely negative tuples, to represent expired tuples. The approach distinguishes between two types of tuples: Positive and Negative tuples. Positive tuples are those regular tuples that are received from the input streams. Negative tuples are a special kind of tuples that are produced from the query pipeline itself. A special operator `EXPIRE` will be added at the bottom of the query pipeline. `EXPIRE` will emit a negative tuple for every expired positive tuple. A negative tuple is responsible for cancelling the effect of a previously processed positive tuple. For example, in time-based sliding-window queries with window of length \(w\), a positive tuple \(t^+\) with a timestamp \(T\) will be followed by a negative tuple \(t^-\) at time \(T + w\). The negative tuple \(t^-\) will follow the foot steps of the corresponding positive tuples. Upon receiving a negative tuple, each operator in the pipeline will behave accordingly to expire the effect of the old tuple. Various query operators will process negative tuples in the following ways:

- **Selection** and **Join** operators handle negative tuples in the same way as positive tuples. The only difference is that the output will be in the form of a negative tuple. Optimizations are introduced in Section 5.1 to avoid re-performing the join for negative tuples.

- **Aggregates** update their aggregate value by dropping the positive tuple corresponding to the received negative tuple.

- **The Distinct** operator reports a negative tuple at the output only if the corresponding positive tuple is in the results reported recently.

The negative-tuples approach is designed with the following goals in mind:

1. Providing a scheme that efficiently coordinates among various query operators within a pipelined query execution plan.
2. Avoiding the delay in updating the continuous sliding-window query answer that may happen with the input-triggered approach.
3. Providing a scheme that is practical and general enough to be easily implemented.

The first goal is achieved by introducing the notion of negative tuples. The second goal is achieved by the fact that the newly introduced negative tuples synchronize the pipeline so that there is no need to wait for any input to trigger the processing. The third goal is achieved by encapsulating the window semantics in a new operator `EXPIRE`. Then, traditional pipelined operators are modified to accommodate the concept of negative tuples independent from the window semantics. Figure 3 gives an example of a query plan using the negative tuples approach. Note that all operators in the pipeline read negative tuples, process them, and may emit negative tuples as output.

Figure 4 gives the pseudo code for the symmetric hash join using the negative tuples approach. Notice that the

(a) Query Q1 with the query pipeline

(b) Execution time line

Figure 2: Input-Triggered Evaluation.

Figure 3: The Negative Tuples Approach.
Algorithm Symmetric Hash Join
Input:
- $H_1$ and $H_2$: Join operator's state of two hash tables for the streams $S_1$ and $S_2$
- An incoming tuple $t_i$ from stream $S_i$

Begin
- If $t_i$ is a positive tuple
  1. $B_x = \text{hash}(t_i)$
  2. Insert $t_i$ in the bucket $B_x$ in the hash table $H_x$
  3. For each tuple $t_j$ in bucket $B_x$ in the other hash table
     - If $t_j$ joins with $t_i$ emit a positive join output tuple $t^+$ for $(t_i$ and $t_j)$ with:
       (a) $t^+.\text{timestamp} = \max(\text{timestamp}(t_i),\text{timestamp}(t_j))$
       (b) $t^+.E\text{timestamp} = \min(\text{timestamp}(t_i),\text{timestamp}(t_j))$
- Else if $t_i$ is a negative tuple
  1. $B_x = \text{hash}(t_i)$
  2. Delete the tuple $t_i$ from the bucket $B_x$
  3. For each tuple $t_j$ in bucket $B_x$ in the other hash table
     - If $t_j$ joins with $t_i$ emit a negative join output tuple for $(t_i$ and $t_j)$ with timestamp = timestamp($t_i$)

End.

Figure 4: Pseudo Code for the Join.

The output of the join operator carries the semantics of two different tuples [31]. The tuple output from the join of two tuples $t_i$ and $t_j$ will have a timestamp equal to $\max(\text{timestamp}(t_i),\text{timestamp}(t_j))$. Assumming the same window size for both streams, the tuple with the minimum timestamp will expire first. The output tuple will expire with the tuple of minimum timestamp. This observation introduces the expiration timestamp. Each positive tuple in the pipeline will carry an additional field named, Etimestamp (Expiration timestamp). Handling Etimestamp by different operators is straightforward and beyond the scope of this paper. Expiration timestamp will be used in Sections 5.1 and 6.

4.3 Handling Delays using Negative Tuples

Figure 5b gives the execution of the negative tuples approach for the example in Figure 5a (the negative tuples implementation of the query in Figure 2a). At time $T_5$, the tuple with value 6 expires. Thus, it appears in $S_1$ as a negative tuple with value 6. The tuple $6^-$ passes the selection filter as it follows the footsteps of tuple $6^+$. At time $T_2$, the SUM operator receives a negative tuple with value 6. Thus, SUM updates its output value to 22. Similarly at time $T_6$, SUM receives a negative tuple with value 9. Thus the result is updated to 13.

The previous example shows that the negative tuple approach overcomes the output delay problem introduced by the input-triggered approach. In the negative tuples approach, tuple expirations are independent from the query characteristics. This means that, even if the query has

highly selective operators at the bottom of the pipeline, we still can produce timely correct answers. On the other hand, if the bottom operator in the query pipeline has low selectivity, then it will pass almost all input tuples to the intermediate queues in the pipeline. In this case, the negative tuples approach may present more delays due to increasing the waiting times in operator queues.

5. NEGATIVE TUPLES OPTIMIZATIONS

Although the basic idea of the negative tuples approach (as described in Section 4.2) is attractive, it may not be practical. The fact that we introduce a new kind of tuples through the query pipeline may result in doubling the number of tuples through the query pipeline. For example, for each incoming positive (regular) tuple, the EXPIRE operator produces its corresponding negative tuple. Doubling the number of tuples through the query pipeline is cumbersome if each negative tuple exactly repeats and undoes the effect of its corresponding positive tuple. In this case, the overhead of processing tuples through the various query operators is doubled. This problem is worst in the case of join subtrees in a query evaluation plan.

This observation opens the room for optimization methods to the basic negative tuples approach. Various optimizations would mainly focus on two issues: (1) Reducing the overhead of re-processing the negative tuples. A query operator should be "smart" enough to remember or at least keep some hints from the processing of each positive tuple $t^+$ to avoid a complete re-processing of the corresponding negative tuple $t^-$. (2) Reducing the number of the negative tuples through the query pipeline. These optimizations can be addressed either at each operator level, or addressed at the source operator EXPIRE.

For the rest of this section, we propose the join message technique to reduce the overhead of processing negative tu-
ples in the join operator. The proposed technique addresses the two issues discussed above. The join message aims to avoid re-processing of a negative tuple and hence reduces the overhead of handling the negative tuples as well as reduces the number of negative tuples emitted from the join operator. As will be explained within the experimental results in Section 7 the join message optimization technique enhances the performance of the negative tuples approach.

5.1 The Join Message Optimization

In this section, we present the join message optimization for reducing the overhead of negative tuples in the query pipeline. The join message optimization is specific for the join operator, which is one of the most expensive operators in the query pipeline. Without this optimization, the join operator would normally re-process the negative tuples in the same way as their corresponding positive tuples. Given the highly expensive cost of the join operation, the join message technique aims to avoid re-processing of the negative tuples. Instead, the join operator will keep some state information in which the newly coming negative tuple can use to avoid the join processing. Then, once a negative tuple arrives to the join operator, a join message will be passed directly to the query operator above the join without performing the join and hence avoid the high cost of join processing.

5.1.1 Algorithm and Data Structure

Upon receiving a positive tuple, the join message optimization acts exactly as the traditional join algorithm in Figure 4. However, upon receiving a negative tuple, instead of re-performing the join operation, the join message optimization performs the following steps: (1) Remove the corresponding positive tuple from the join’s state, (2) Set a special flag in this tuple indicating that this tuple is a join message (3) Put the join message in the join’s output queue.

When the operator above the join receives a negative tuple with the join message flag set, it learns that all positive tuples that resulted previously from joining with this tuple are expired and acts accordingly. This can be achieved by scanning the operator’s state and expiring all tuples that carry the same expiration timestamp (Etimestap) as the join message. If the operator’s state is sorted on the Etimestap then this scan should not be costly. If the operator above the join does not carry a state (e.g., a Select), then it will just pass the join message to its output queue. If the join operator is the last operator in the pipeline, then the join message is sent directly to the output stream. The join message in this case can be interpreted as a negative tuple that expires a group of previously emitted positive tuples (the positive tuples having the same Etimestap as the join message).

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) The negative tuples are “passed” through the join operator without probing the other hash table. (2) Only one negative tuple is emitted for every processed negative tuple independent from its join multiplicity.

5.1.2 Example

Figure 6 gives an example on the join message approach. Figure 6a is the query pipeline. Notice that in the sliding-window model, aggregate operators keep a state containing the tuples in the current window. The table beside the SUM operator gives its state. The table consists of two columns: the first column is for the value and the second column is for the tuple Etimestap (other attributes may be stored in the state but are omitted for clarity of the discussion). Figure 6b gives the stream of tuples in the pipeline when using the negative tuples approach and before applying the join message optimization. Figure 6c gives the stream of tuples in the query pipeline after applying the join message optimization. Tuple $t^+$ arrives at Stream $S_1$ at time $T_1$. Three subsequent tuples from $S_2$ (at times $T_2$, $T_3$ and $T_4$) join with the tuple $t^+$ (at time $T_1$) from Stream $S_1$. The output of the join will have Etimestap of the tuple that will expire first from the two joining tuples. In this example, the output of the join will carry Etimestap $T_1$ since tuple $t^+$ from $S_1$ will expire first. At time $T_5$, tuple $t^+$ from Stream $S_1$ expires. In the negative tuples approach (Figure 6b), the join operator will perform the join with tuple $t^+$ and output three output negative tuples. When applying the join message optimization, (Figure 6c), the join operator sends a join message with timestamp $T_1$ to its output queue. Upon receiving the join message, the SUM operator scans its state and expires all tuples with Etimestap $T_1$ and produces a new output.

5.1.3 Reducing the number of join messages

One problem in the join message approach as described in the previous section is that if the join operator sends a join message for every incoming negative tuple, then unnecessary messages may be sent even if their corresponding positive tuples did not produce any join results before. This happens when the join filter is highly selective (i.e., when most of the input tuples do not produce join outputs).

In this section, we propose techniques to send only the necessary join messages. Join operators can be classified into two classes: (1) joining a stream with a table and (2) joining two streams.

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 will keep a table (Joined Tuples Table, JTT) in a sorted list (sorted on the timestamp). When a positive tuple is processed and produces join results, then the timestamp of this positive tuple is entered in JTT. At most, the size of this table is equal to the window size.

When a negative tuple is to be processed, the join checks whether there is a tuple with the same timestamp in JTT. If found, then this tuple has produced join results previously, then a join message will be sent for this tuple and its timestamp is removed from JTT. 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. The join message will be responsible for expiring all the previously produced join results.

Joining two streams: When the join operator joins two tuples $t^+_i$ from $S_1$ and $t^+_i$ from $S_2$, the resulting tuple $t^+_i$ should expire whenever either $t^+_i$ or $t^+_i$ expire. Assume that $t^+_i$ will expire first. To expire $t^+_i$, only the join message for $t^+_i$ is needed.
Algorithm **Symmetric Hash Join**

**Input:**
- $H_1$ and $H_2$: Join operator’s state of two hash tables for streams $S_1$ and $S_2$
- An incoming tuple $t_i$ from Stream $S_i$

**Begin**

- If $t_i$ is a positive tuple
  1. $B_x = \text{hash}(t_i)$
  2. Insert $t_i$ in the bucket $B_x$ in the hash table $H_i$
  3. For each tuple $t_j$ in bucket $B_x$ in the other hash table
     - If $t_j$ joins with $t_i$ emit a positive join output
       tuple $t^+$ for $(t_i$ and $t_j)$ with:
         (a) $t^+.\text{timestamp} = \max(\text{timestamp}(t_i), \text{timestamp}(t_j))$
         (b) $t^+.\text{timestamp} = \min(\text{timestamp}(t_i), \text{timestamp}(t_j))$
     - If timestamp $t_j < \text{timestamp}(t_i)$ increment reference count of $t_j$ by one
     - Else increment reference count of $t_i$ by one

- Else if $t_i$ is a negative tuple
  1. $B_x = \text{hash}(t_i)$
  2. Delete the tuple $t_i$ from the bucket $B_x$
  3. If reference count of $t_i > 0$ then send a join message with timestamp = timestamp($t_i$)

**End.**

**Figure 7: Pseudo Code for the Modified Join.**

To avoid the unnecessary join messages, a reference count will be kept with every tuple $t_i$ in the hash table. This reference count indicates the number of output tuples that should 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 $t_y$ has the minimum timestamp. The pseudo code for the join operator after adding the reference count is shown in Figure 7.

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 it is greater than zero then a join message for this tuple is emitted.

**Example:** Figure 8 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 $T_3$), it will increment the reference count of $t_i$. At time $T_5$, tuple $t_i$ from Stream $S_1$ 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 its reference count is zero.

6. **THE PIGGYBACKING APPROACH**

As described in Section 4.2, the main motivation behind the negative tuples approach is to avoid the output delay that is incurred in the input triggered approach. 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, the input-triggered approach is preferable over the negative tuples approach.

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 [21].

In this section, we present the Piggybacking approach for efficient pipelined execution of sliding-window queries. A similar notion of piggybacking was used in [1] to reduce storage needed to process a query. The main idea of the Piggybacking optimization is to self-tune the query pipeline by alternating between both the negative tuples and input triggered approaches. Thus, in the Piggybacking approach, negative tuples are processed only when they are needed. If positive tuples are flowing in the query pipeline with high...
rates (i.e., higher than the operators’ processing rates), then every operator will always find positive tuples in its input queue. In this case, the positive tuples will carry the required information for expiration and there is no need for the negative tuples. In other words, there is no need to expire explicitly each tuple by having an explicit negative tuple. Instead, the information for expiring some tuples (timestamp) can be piggybacked with one of the incoming positive tuples.

The piggybacking approach works in two stages:

**Producing a piggybacking flag.** Once an operator produces an output tuple \( t \) (either positive or negative), it checks if there is any negative tuple in its output queue (i.e., the input queue of the next query operator in the pipeline). If there is at least one negative tuple in the output queue, we perform two operations: (1) The newly produced tuple \( t \) is tagged by a special flag \( PGF\text{ag} \), (2) All the negative tuples in the output queue are purged. The timestamp of the tagged tuple is used in the second stage to direct the execution in the pipelined query operators.

**Processing the piggybacking flag.** Once a query operator receives a tuple \( t \) (either positive or negative) at time \( T \), it checks for the \( PGF\text{lag} \) in \( t \). If the input tuple is not tagged by the piggybacking flag, the query operator will act exactly as the negative tuple approach, described in Section 4.2. However, if the incoming tuple is tagged by the piggybacking flag, the query operator will act as the input-triggered approach, described in Section 4.1. This means that all tuples stored in the operator state with a timestamp less than \( T - w \) should expire (no negative tuples will arrive for these tuples later). Notice that waiting for a negative tuple for triggering expiration is performed only when there is a lack of inputs (either by the network or selective operators). If there is no lack of inputs, then there is no need for processing negative tuples. In the case that processing the incoming tuple \( t \) does not result in any output (e.g., filtered with the select or join criteria), we output a null message that contains only the timestamp and the piggybacking flag so that later operators in the pipeline behave accordingly.

The piggybacking flag (\( PGF\text{lag} \)) is a generalization of the join message described in Section 5.1. The main difference is that a join message with timestamp \( T \) is responsible for expiring tuples with Etimestamp \( T \), while a \( PGF\text{lag} \) with timestamp \( T \) is responsible for expiring all the tuples with Etimestamp less than \( T - w \).

**Example:** Figure 9 gives an example on the piggybacking approach. This example uses the same query of Figure 5a. The example shows that when the select operator is highly selective (in the period \( T_3 \) to \( T_4 \)) negative tuples are passed to the SUM operator for immediate expiration of tuples with value 6 and 9. At time \( T_4 \), the select operator emits tuple 5 immediately followed by tuple 7+. If tuple 7+ is emitted before the SUM operator reads 5−, then 5 will delete 5− from the queue and SUM will read only tuple 7+. While processing 7+, SUM checks its timestamp and knows that tuple with value 5 (that is stored in SUM’s state) should expire. Then, SUM emits the new answer reflecting the expiration of 5 and the addition of 7. The same happens at time \( T_{11} \). This example shows that the delay in the answer update will be the minimum possible delay.

### 7. EXPERIMENTS

In this section we present experimental results from the implementation of our algorithms in a prototype data stream management system. We compare the performance of the negative tuples approach with the input-triggered approach and show how the proposed optimization further enhances the performance.

#### 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 SSCAN operator. The \( EXPIRE \) operator is implemented as part of the SSCAN operator that is scheduled upon the arrival of input tuples. 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 or negative.

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 5), a tuple \( t \) arrives at the system at time \( T \). The SUM produces an output tuple after adding the value of \( t \) at time \( T + d \), then this tuple encountered output delay of \( d \) units of time.

We compare the input-triggered and negative tuples approaches for various operator selectivities and different input arrival rates.

The stream SalesStream used in the queries has the following schema: (StoreID, ItemID, Price, Quantity, Timestamp). We use randomly generated synthetic data. The Timestamp follows the exponential distribution and is assigned to the tuple when the tuple arrives to the server.

#### 7.2 Input Triggered vs. Negative Tuples

In this section we show how the negative tuples approach improves the performance for queries that have highly selective operators.

##### 7.2.1 Effect of operator selectivity

Figure 5 gives the effect of changing the selectivity of the select operator in Q1 (Figure 5a). Figure 10a gives the average output delay while Figure 10b gives the maximum delay.
output delay. The input rate is fixed to 50 tuples/second and the selectivity changes from 0.1 to 0.6. For low selectivity, the input-triggered approach shows high output delay since the SUM operator will not expire old tuples until a new input tuple passes the selection filter. The negative tuples approach does not depend on the selectivity since tuple expiration takes place even if no input tuples pass the selection filter.

Figure 11 gives the average and maximum output delay for the join query Q2 (Figure 6a). When the join is highly selective, many input tuples will be filtered out and will not reach the SUM operator. This causes the high output delay in the input-triggered approach. The negative tuples approach encounters less output delays since the negative tuples are sent out to update the query answer even if the input tuples are filtered out.

7.2.2 Effect of arrival rates

This experiment gives the effect of different input rates for query Q1 (Figure 5a). Figure 12 gives the average and maximum output delay for the two approaches. The arrival rate changes from 10 to 50 tuples/second and the selectivity is fixed to 0.4. For low arrival rates, the input-triggered approach encounters higher output delays. The negative tuples approach encounters less output delays when the rate of arrival increases.

7.3 The Join Message Optimization

Figure 13 illustrates how the join message optimization reduces the overhead of processing negative tuples.

In this experiment the join selectivity is set to 0.01 while the tuple’s join multiplicity ranges from 1 to 5. For example, assume the number of tuples in each window is 100, then for a join multiplicity of 0.01, 100 tuples will be output from the join in each window (100/100*100).

Figure 13a shows the ratio between the number of negative tuples and positive tuples. The number of negative tuples represents the overhead associated with the negative tuples approach. This overhead is always zero for the input triggered approach. The overhead is almost equal to one in the negative tuples approach 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 measurement was taken).
Figure 12: Q2: Effect of Arrival Rate.

Figure 13: Effect of the Join Message.

The Piggybacking Approach

This section shows that the piggybacking optimization self-tunes the system to work in either the negative tuples approach or the input-triggered approach according to the system’s load.

7.4.1 Performance enhancement

Figure 14 compares the output delay for the three approaches (Input-triggered, negative tuples and piggybacking) for various arrival rates and selectivities. Figure 14a is for Query Q2 (Figure 6a) with varying input rates while the join selectivity fixed at 0.01. Figure 14b is for the same query while varying the join selectivities with the input rate fixed at 50 tuples/second. The figure illustrates that for low arrival rates or high selectivity, the negative tuples approach gives less output delay since the expiration is performed independent from the arrival of new input tuples. On the other hand, when the input arrival rate increases, the negative tuples approach encounters more output delays since the queues are flooded by positive and negative tuples. The piggybacking approach gives the minimum possible output delay in all arrival rates and all selectivities since it processes the negative tuples only when necessary.

7.4.2 Reducing overhead

This experiment shows how the piggybacking approach reduces the number of negative tuples processed by operators in the pipeline. Figure 15 shows the ratio between the number of negative tuples and the number of positive tuples processed by the SUM operator in Query Q2. We vary the arrival rate and join selectivity in Figure 15a and Figure 15b, respectively. For the input-triggered approach, the ratio is always zero since there are no negative tuples processed. In the negative tuples approach, the ratio is almost one since one negative tuple is processed for every positive tuple. In
the piggybacking approach, the ratio decreases for higher input rates and lower selectivity. The reason is that positive tuples are flowing in the query pipeline with high rate and purging negative tuples (if any) from the queue before being processed.

8. CONCLUSIONS

Negative tuples have been adopted by data stream management systems as a coordination scheme among various pipelined query operators. In this paper, we show that although negative tuples succeed to avoid the shortcomings of the traditional input-triggered approach (e.g., non-deterministic output delays), negative tuples suffer from a major drawback. Negative tuples double the number of tuples in the query pipeline, hence the pipeline bandwidth is reduced to half. Thus, we presented two optimization techniques that enhanced the performance of the negative tuples approach. The first optimization, namely the join message optimization, is concerned with the join operator subtree. The main idea is to filter out some of the negative tuples from the join operator input queue, thus, avoiding to re-execute the expensive join operation. The second optimization, namely the piggybacking optimization, self-tunes the query pipeline to work in either the input-triggered or the negative tuples schemes according to the rate of the tuples flowing in the query pipeline. With piggybacking, the query pipeline got the benefits of both the traditional input-triggered and the negative tuples approach. Both optimizations can be applied independently or together to enhance the performance of negative tuples. Experimental results based on a real implementation of input-triggered, negative tuple, join messages, and piggybacking 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 the negative tuples or the input-triggered approaches.

9. REFERENCES


Figure 14: The Piggybacking Approach: Performance.
Figure 15: The Piggybacking Approach: Overhead.


