New Evaluation Era of Incremental Sliding Windows Queries Over Data Streams

Sunil Kumar Sah , Sheo Shankar Prasad

Abstract— Two research efforts have been conducted to realize sliding-window queries in data stream management systems, namely, query reevaluation and incremental evaluation. In the query reevaluation method, two consecutive windows are processed independently of 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.

Index Terms— Query Reevaluation, Incremental Evaluation, Input Triggered, Tuple Expiration, Negative Tuples.

INTRODUCTION:-

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.

In the query reevaluation 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.

- *Sunil Kumar Sah, UDCA, Computer Centre, T.M.Bhagalpur University, Bhagalpur, India,09504611107, E-mail: sunilsahil@rediffmail.com.*
- *Sheo Shankar Prasad, UDCA, Computer Centre,, T.M.Bhagalpur University, Bhagalpur, India,9162380697. E-mail: sheobgp@gmail.com)*

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.

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.

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. 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. 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 and W-EXPIRE). 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 and 2) reducing the number of 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 are 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, IT A and NTA, for various queries. This comparison helps identify the appropriate situations in which 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.

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, S1 to Sn. Each input data stream Sj is assigned a window of size wj. At any time instance T, the answer to the sliding-window query is equal 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 Si contains the tuples arriving between times T-wi and T. The same notions of semantics for continuous sliding-window queries are used in other systems. 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, S1 to Sn, are time-stamped 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 wi associated with stream Si represents the lifetime of a tuple t from Si. 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 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. 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 is equal to the tuple arrival time, and Ets, which is equal to ts þ wi. 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 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.

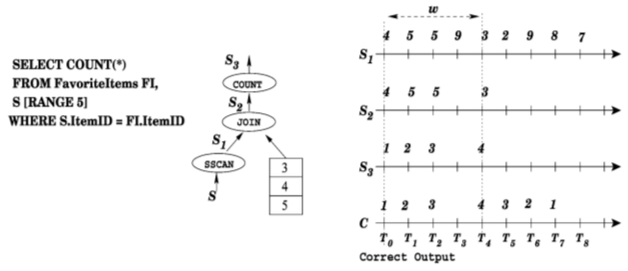
3.1 The Input-Triggered Approach (ITA)

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 is equal 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 which are a result of 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, t1, may co-exist in the window with another tuple, t2, but t2 may get delayed at a lower operator in the pipeline. 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 Q1 "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 Q1) will not receive any notification about the current time and, hence, will not expire old tuples.

Fig. 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 Favorite Items. 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 until an input tuple passes the join and triggers the scheduling 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.

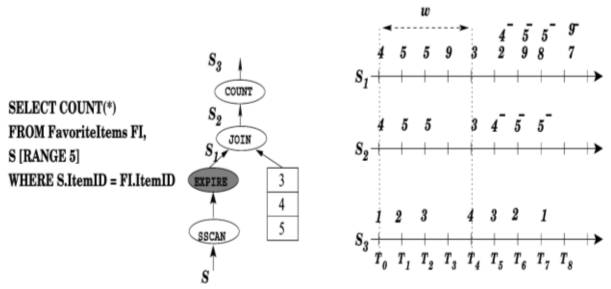
Figure 1.



(a) **(b)**

Input-triggered evaluation (a) Query Q1 with the query pipeline. (b) Execution timeline.

Figure 2.



(a) **(b)**

Negative tuples evaluation (a) Query Q1 with query pipeline. (b) Execution timeline.

3.2 *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. 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 a 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.

3.3 Handling Delays Using Negative Tuples

Fig. 2b gives the execution of NTA for the example in Fig. 2a (the negative tuples implementation of the query in Fig. 1a). At time T5, the tuple with value 4 expires and appears in S1 as a negative tuple with value 4. The tuple 4^- joins with the tuple 4 in the *Favorite Items* table. At time T_5, COUNT receives the negative tuple 4^-. Thus, COUNT outputs a new count of 3. Similarly, at time T6, 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 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.

3.4 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). 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. Conclusions

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.

5 *ACKNOWLEDGEMENT*

I would like to thank the staff of ISI Kolkatta, for their valuable support.

6. *REFERENCES*

1. Arasu and J. Widom, "Resource Sharing in Continuous Sliding- Window Aggregates," Proc. Int'l Conf. Very Large Data Bases (VLDB), 2004.

2. Ayad and J.F. Naughton, "Static Optimization of Conjunctive Queries with Sliding Windows over Infinite Streams," Proc. ACM SIGMOD Conf., 2004.

3. DATAR, M., GIONIS, A., INDYK, P., AND MOTWANI, R. 2002. Maintaining stream statistics over sliding windows. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 635-644.

4. Li, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. A. 2005. Semantics and evaluation techniques for window aggregates in data streams. *In Proceedings of the ACM SIGMOD International Conference on Management of Data,* 311-322.