Update-Pattern-Aware Modeling and Processing of Continuous Queries*

Lukasz Golab M. Tamer Özsu
School of Computer Science
University of Waterloo, Canada
{lgolab,tozsu}@uwaterloo.ca

ABSTRACT

A defining characteristic of continuous queries over on-line data streams, possibly bounded by sliding windows, is the potentially infinite and time-evolving nature of their inputs and outputs. New items continually arrive on the input streams and new results are continually produced. Additionally, inputs expire by falling out of range of their sliding windows and results expire when they cease to satisfy the query. This impacts continuous query processing in two ways. First, data stream systems allow tables to be queried alongside data streams, but in terms of query semantics, it is not clear how updates of tables are different from insertions and deletions caused by the movement of the sliding windows. Second, many interesting queries need to store state, which must be kept up-to-date as time goes on. Therefore, query processing efficiency depends highly on the amount of overhead involved in state maintenance.

In this paper, we show that the above issues can be solved by understanding the update patterns of continuous queries and exploiting them during query processing. We propose a classification that defines four types of update characteristics. Using our classification, we present a definition of continuous query semantics that clearly states the role of relations. We then propose the notion of update-pattern-aware query processing, where physical implementations of query operators, including the data structures used for storing intermediate state, vary depending on the update patterns of their inputs and outputs. When tested on IP traffic logs, our update-pattern-aware query plans routinely outperform the existing techniques by an order of magnitude.

1. INTRODUCTION

On-line stream processing has recently become a popular data management problem, driven by emerging applications such as sensor networks, Internet traffic analysis, Web-based financial tickers, and transaction log analysis. A number of recent projects aim at designing a data stream management system (DSMS) capable of executing *continuous queries* that run over a period of time and incrementally produce new answers as new data arrive.

Due to the potentially infinite nature of data streams, many queries cannot be computed in finite memory [2]. A general solution for bounding the memory requirements of continuous queries is to define *sliding windows* on the incoming streams. At any time, a *time-based* sliding window of size T retains only those items which have arrived in the last T time units, whereas a *count-based* window of size N retains the N most recent items. There are other ways of bounding the memory requirements of continuous queries—*punctuations* [22] and *k-constraints* [7] are two examples—but they involve making assumptions about the nature of the data arriving on the stream.

An important characteristic of continuous queries is that their inputs and outputs evolve over time. New answers are produced in response to the arrival of new data and older data expire as the windows slide forward. Furthermore, some previously reported answers may cease to satisfy the query at some point. We define an *update pattern* of a continuous query as the order in which its results are produced and deleted over time. The purpose of this paper is to introduce the notion of update-pattern-aware modeling and processing of continuous queries. In particular, we classify continuous queries based on their update patterns and use this classification to a) provide a precise definition of the semantics of continuous queries over streams, windows, and relations, and b) introduce update-pattern-aware query processing strategies that outperform the existing approaches.

Previous work on update patterns of queries over streams has only distinguished between monotonic and non-monotonic queries, with the conclusion that only monotonic queries are feasible over infinite streams [16, 21]. Let $Q(\tau)$ be the answer of a continuous query Q at time τ. Q is monotonic if $Q(\tau) \subseteq Q(\tau+\epsilon)$ for all τ and all $\epsilon \geq 0$. The update patterns of monotonic queries are trivial to define because no result is ever deleted from the answer set. Hence, this classification is not sufficiently precise as it fails to classify non-monotonic queries according to the pattern of deletions from their answer set. This is a significant omission as most non-trivial continuous queries are non-monotonic because they require windowing when dealing with infinite streams. To see this, consider the following generally accepted definition of continuous query semantics [11, 15].

*This research is partially supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), Sun Microsystems of Canada, and Communications and Information Technology Ontario (CITO).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

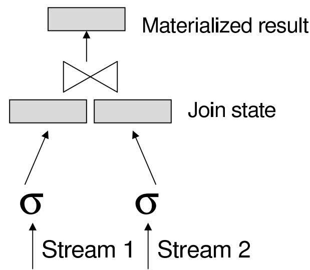

Figure 1: Example of a continuous query plan computing a join of two streams.

DEFINITION 1. *At any time τ, $Q(\tau)$ must be equal to the output of a corresponding one-time relational query whose inputs are the current states of the streams, sliding windows, and relations referenced in Q.*

For now, assume that incoming tuples are processed immediately and in order of arrival. According to Definition 1, all queries over sliding windows are non-monotonic because their results expire as the windows slide forward.

To motivate the need for a more detailed classification of the update patterns of continuous queries, note that Definition 1 is imprecise because it does not distinguish between the time-evolving state of the data streams versus relational tables. Existing research either disallows relations in continuous query plans (e.g., [4]), implicitly assumes that relations are static, at least throughout the lifetime of the query (e.g., [1, 11, 17]), or allows arbitrary updates of relations [18]. If updates of tables are allowed, one would intuitively think that they are semantically different from updates of input streams caused by the movement of the sliding windows.

Another issue that depends on the knowledge of update patterns is the maintenance of operator state. Consider the plan in Figure 1 for computing a join of two streams and materializing the result. New tuples are processed by the corresponding selection operators and surviving tuples are inserted into the state of the join operator. For each arrival in one of the two state buffers, the join operator probes the other buffer and produces new results, which are then inserted into the materialized result. If the input streams are bounded by sliding windows, then the intermediate join state must be kept up-to-date to ensure that old tuples do not produce any new join results. Similarly, the materialized output must conform to Definition 1, which is to say that expired tuples must continually be deleted as the windows slide forward. Clearly, updating (sub)results by sequentially scanning the state buffers is exceedingly inefficient, therefore we would like to know the order in which tuples arrive and expire so that suitable data structures may be designed to minimize the state maintenance overhead.

The three contributions of this work consist of the classification of update patterns of continuous queries, and two applications of this classification: definition of precise query semantics, and introduction of update-pattern-aware query processing techniques. In particular:

- We present a classification of update patterns of continuous queries that forms the basis of update-pattern-aware query processing. Our classification divides non-monotonic queries into three types to highlight the differences in their expiration patterns.
- Using this classification, we give a precise definition of continuous query semantics that considers the update patterns of relations and streams separately.
- We develop update-pattern-aware query plans, where each branch in the plan is annotated with the update patterns of the input flowing on it, and physical operator implementations vary according to the nature of their inputs. In particular, operators use update-pattern-aware data structures for intermediate state maintenance. When tested on IP traffic logs, our update-pattern-aware query plans significantly outperform the existing data stream processing techniques.

In the remainder of this paper, Section 2 discusses related work in continuous query processing, Section 3 presents our classification of update patterns of continuous queries, Section 4 uses the classification to define continuous query semantics, Section 5 develops update-pattern-aware query processing strategies, Section 6 presents our experimental results, and Section 7 concludes the paper with suggestions for future work.

2. PREVIOUS WORK IN CONTINUOUS QUERY PROCESSING

We begin by reviewing related work in continuous query processing. First, we make the following assumptions.

- A data stream is an append-only sequence of relational tuples with the same schema.
- All sliding windows are assumed to be time-based. We will comment on count-based windows in Section 7.
- Upon arrival at the system, each tuple is assigned a nondecreasing timestamp ts. This allows us to ignore communication delays and out-of-order arrival, which were addressed in [1, 20].
- Each new tuple is processed immediately by all the operators in the query before the next tuple is processed. Consequently, results are produced in timestamp order. More advanced scheduling strategies have been proposed in [5, 9, 13] and are orthogonal to this work.

2.1 Continuous Query Operators

The main difference between standard relational operators and continuous query operators is that the latter must incrementally produce new answers and may need to deal with expiring tuples from sliding windows. The operator implementations are presented in this section; more details may be found in, e.g., [10, 11, 14, 15, 23].

Projection, selection, and union are unary operators that process new tuples on-the-fly, either by discarding unwanted attributes (projection), dropping tuples that do not satisfy the selection condition, or propagating the inputs up the query plan (union). These operators are stateless and do not have to be modified to work over sliding windows. Note that only non-blocking merge union is allowed to ensure that output is produced in arrival order [16].

Join and intersection are binary operators that store both of their inputs. Each new arrival is inserted into its state buffer and triggers the probing of the other input's state buffer to find matching results. New results are then

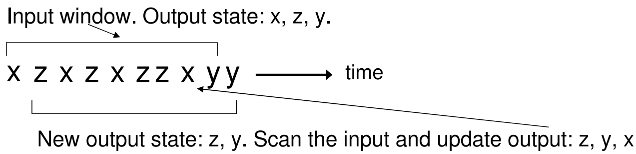

Figure 2: Behaviour of the duplicate elimination operator over a sliding window.

appended to the output stream. The state of both inputs must be maintained so that expired tuples are not used during the probing step to produce any new results. However, expiration can be done periodically (lazily), as long as expired tuples can be identified and skipped during processing. The tradeoff is that memory usage increases because we are temporarily storing expired tuples.

Duplicate elimination over a sliding window stores both its input and its current output. At all times, the output must contain exactly one tuple with each distinct value v present in the input. When a new tuple arrives, it is inserted into the input buffer, and matched against the stored output. If the new tuple is not a duplicate, it is added to the output state and appended to the output stream. When a result tuple with value v expires from the output state, we take immediate action as follows. The input buffer is scanned to determine if there are any other tuples with value v that have not expired (as in the window join, the input buffer can be maintained lazily). If so, one of these tuples, say the youngest, is inserted into the output state and appended to the output stream. An example is illustrated in Figure 2. When the result tuple with value x expires from the output, it is replaced with another tuple having value x that has not yet expired.

Group-by incrementally updates the value of a given aggregate for each group; we do not consider aggregation as a separate operator as it can be represented as group-by with a single group. For each new input, we add it to the state buffer, determine which group it belongs to, and return an updated result for this group. The new result is understood to replace a previously reported result for this group. Also, for each tuple that expires from the input state, we decrement the aggregate value of the appropriate group and return a new result for this group on the output stream. The input must be maintained eagerly so that the returned aggregate values are up-to-date.

Negation stores its left and right inputs (call them W_1 and W_2, respectively) along with multiplicities of all the distinct values occurring in each input. Let v_1 and v_2 be the number of tuples with value v in W_1 and W_2, respectively. For each distinct value v present in W_1, the output of the negation operator consists of v_3 tuples from W_1 such that

$$v_3 = \begin{cases} v_1 - v_2 & \text{if } v_1 > v_2 \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

A new arrival on W_1 with value v is inserted into its state buffer and the corresponding counter (v_1) is incremented. If $v_1 > v_2$, the new tuple is appended to the output stream. Expiration from W_1 is handled (eagerly) by removing the old tuple from the W_1 state and decrementing v_1.

An arrival on W_2 with value v is inserted into its state buffer and increments v_2. If $v_2 \leq v_1$, one result tuple with value v (say the oldest) must be deleted from the answer set to satisfy the negation condition in Equation 1. These explicit deletions are usually implemented in the form of *negative tuples* [3, 11] generated by the negation operator and appended to the output stream. A negative tuple with a particular set of attributes signals that the corresponding result tuple is no longer part of the result. Finally, if a tuple with value v expires from W_2, we decrement v_2 and if $v_1 \geq v_2$, we probe W_1 and append one tuple from W_1 with value v (say the youngest) to the output stream.

2.2 Determining when Results Expire

All stateful operators over sliding windows must remove old tuples from their state buffers. Expiration from an individual (time-based) sliding window is simple: a tuple expires if its timestamp falls out of the range of the window. Expiration from intermediate results can be determined as follows. When a new tuple with timestamp ts arrives in a window, we attach to it another timestamp, exp, that denotes the expiration time of this tuple. exp is derived by adding one window size to ts. If this tuple joins with a tuple from another window, whose timestamps are ts' and exp', the expiration timestamp of the result tuple is the minimum of exp and exp'. That is, a composite result tuple expires if at least one of its constituent tuples expires from its windows (recall Definition 1). As discussed above, only the negation operator can force some result tuples to expire earlier than their exp timestamps by generating negative tuples.

2.3 Query Execution Strategies

As the input windows slide forward, continuous query operators process two types of events: arrivals of new tuples and expirations of old tuples. New tuples are inserted into the operator state and processed as appropriate, whereas the handling of expirations depends on the operator, and involves removing the expired tuple from the state and possibly generating new results. Similarly, a materialized view of the result is maintained by inserting new results into the view and expiring stale results. Note that it may not be sufficient to perform state expiration only when new tuples arrive, unless the arrival rates are high. For example, suppose that we materialize the result of a sliding window aggregate. It may be the case that no new tuples arrive on the input for some time, but the aggregate value changes as a result of expiration from the input. In general, duplicate elimination, group-by, and negation may produce new output in response to expirations, therefore these operators must maintain their state eagerly. There are two techniques for maintaining correct query results: the negative tuple approach and the direct approach [11, 12].

2.3.1 Negative Tuple Approach

In the negative tuple approach, each window referenced in the query is materialized and explicitly generates a negative tuple for every expiration (in addition to pushing newly arrived tuples into the plan). This generalizes the purpose of negative tuples, which are now used to signal all expirations explicitly (instead of being produced by the negation operator only if a result tuple expires because it no longer satisfies the negation condition). Negative tuples propagate through the query plan and are processed by operators in a similar way as regular tuples, but they also cause operators to remove corresponding "real" tuples from their state. This is illustrated in Figure 3, showing how the join query from Fig-

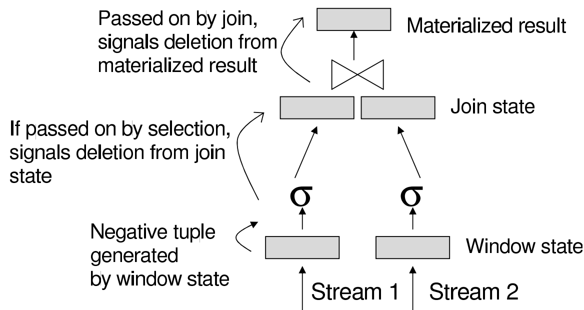

Figure 3: Processing negative tuples generated by expirations from a window over Stream 1.

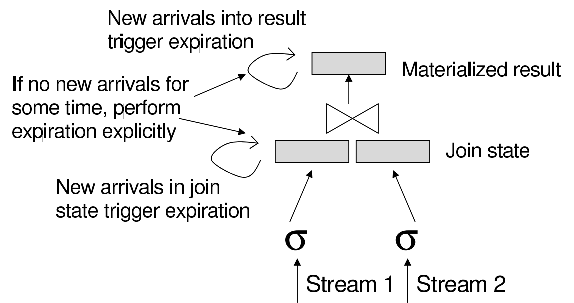

Figure 4: Query execution using the direct approach (only deletions from Stream 1 window are shown).

ure 1 processes a particular negative tuple generated by an expiration from the window over Stream 1 (expirations from the other window are treated similarly and are not labeled for clarity). Observe that the negative tuple is processed by all the operators in the pipeline, so that the materialized result may eventually receive a number of negative tuples, corresponding to all the join results in which the original negative tuple participated.

The negative tuple approach can be implemented efficiently if the operator state is sorted by key so that expired tuples can be looked up quickly in response to negative tuples. The downside is that twice as many tuples must be processed by the query because every tuple eventually expires from its window and generates a corresponding negative tuple. Furthermore, the input windows must be stored so that we know when to generate negative tuples.

2.3.2 Direct Approach

Negation-free sliding window queries have the property that expiration times of (intermediate and final) results can be determined via *exp* timestamps, as explained in Section 2.2. Hence, operators can access their state directly and find expired tuples without the need for negative tuples. The direct approach is illustrated in Figure 4 for the same query as in Figure 3; again, only deletions from the window over Stream 1 are illustrated. For every new arrival into one of the join state buffers, expiration is performed at the same time as the processing of the new tuple. However, if there are no arrivals for some time (this interval may be specified by the user as the maximum delay in reporting new answers), each operator that stores state, including the final materialized result, initiates expiration from its state buffer.

A technical issue with the direct approach is that newly arrived tuples may not be processed immediately, therefore the state of intermediate results may be delayed with respect to the inputs. For example, in Figure 4, tuples with timestamps of up to 100 may have arrived on the input streams, but the join operator may have only processed tuples with timestamps up to 98. A solution to guarantee correct results maintains a local clock at each operator, corresponding to the timestamp of the tuple most recently processed by its parent [11]. This way, the local clock of the join operator in Figure 4 is 98 and it will not expire tuples out of its state prematurely by assuming that the current time is 100.

2.3.3 Discussion

The direct approach does not incur the overhead of processing negative tuples and does not have to store the base windows (labeled as window state in Figure 3), but it may be slower than the negative tuple approach for queries over multiple windows [11]. This is because straightforward implementations of state buffers may require a sequential scan during insertions or deletions. For example, if the state buffer is sorted by tuple arrival time, then insertions are simple, but deletions require a sequential scan of the buffer. On the other hand, sorting the buffer by expiration time simplifies deletions, but insertions may require a sequential scan to ensure that the new tuple is ordered correctly, unless the insertion order is the same as the expiration order. To solve this problem, in Section 3, we will define the order in which various continuous queries expire their results. In Section 5, we will propose an update-pattern-aware query execution technique that outperforms both the negative tuple and the direct approaches.

3. UPDATE PATTERNS OF CONTINUOUS QUERIES

3.1 Classification

Recall that continuous queries may be broadly divided into two groups—monotonic and non-monotonic—with all queries over sliding windows being non-monotonic. We now present a more precise classification that sub-divides non-monotonic queries into three types that exhibit progressively more complex update patterns: weakest, weak, and strict.

- *Monotonic* queries produce an append-only output stream and therefore do not incur deletions from their answer set. Only stateless operators over infinite streams (projection, selection, union, and distributive aggregates) can give rise to monotonic queries.
- *Weakest non-monotonic* queries do not store state and do not reorder incoming tuples during processing; tuples are either dropped or appended to the output stream immediately. As a consequence, results expire in the same order in which they were generated, i.e., first-in-first-out (FIFO). Projection and selection over a single sliding window are weakest non-monotonic, as is a merge-union of two windows.
- *Weak non-monotonic* queries may not expire results in FIFO order, but have the property that the expiration

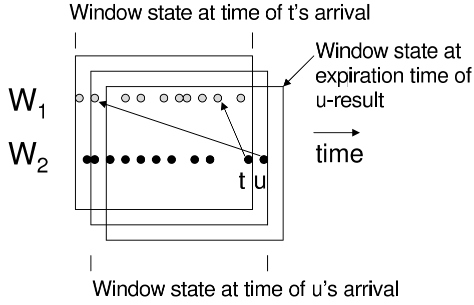

Figure 5: Illustration of the update patterns of a sliding window join.

time of each result tuple can be determined without generating negative tuples on the output stream. Examples include join, duplicate elimination, and group-by.

- *Strict non-monotonic* queries have the property that at least some of their results expire at unpredictable times and require the inputs to generate negative tuples to explicitly signal these expirations. Negation over two windows is one example.

All but the strict non-monotonic queries produce results that can be materialized and maintained without the need to generate negative tuples. This is why only negation-free queries are compatible with the direct approach to state maintenance defined in Section 2. Note that sliding window queries with strict non-monotonic patterns must also be non-monotonic over infinite streams (e.g., negation). The reason is that windowing alone does not produce premature expirations (individual windows are weakest non-monotonic because their results expire in FIFO order). Moreover, if an operator is monotonic over streams, but possibly impractical due to the need to store unbounded state, then it must be weak non-monotonic over windows (e.g., join). The reason is that storing state and referring to it during processing creates possibilities for reordering incoming tuples, as will be explained below.

3.2 Discussion

In Figure 5, we illustrate the behaviour of a sliding window join over two windows with the same size, W_1 and W_2. W_1-tuples are represented as light dots, whereas W_2-tuples are drawn as dark dots. When tuple t arrives on W_2, suppose that it joins with the W_1-tuple indicated by the arrow. Similarly, when tuple u arrives next, it joins with the indicated W_1-tuple. The three overlapping rectangles symbolize the state of the input windows at three times: t's arrival time, u's arrival time, and the time when the join result involving u expires from the result (when at least one of the base tuples that make up the result has expired from its window). The update patterns are not FIFO because the u-result has expired before the t-result, yet the t-result was generated first.

To understand why duplicate elimination is weak non-monotonic, recall the example in Figure 2. When the old tuple with value x expires, we append a newer tuple with the same value to the output stream. However, two tuples with value y arrived after the newer x-tuple, yet one of them was appended to the output stream before the x-tuple. Thus, the expiration order is not the same as the order of insertion into the result stream because some tuples may be appended to the output stream not immediately after they arrive.

As described in Section 2, group-by produces a new result for a given group whenever the aggregate value of this group changes. Clearly, some groups may be updated more often than others, therefore the expiration order of result tuples is not FIFO. However, we know when each result tuple expires (namely when a new result for the same group is produced on the output stream) without the need for negative tuples.

Negation is strict non-monotonic because, as discussed in Section 2, it produces negative tuples in some situations to indicate that previously reported results no longer satisfy the query. These expirations are not caused by the movement of the sliding windows, but rather by the semantics of the negation operator. Other tuples expire as predicted by the *exp* timestamps and therefore do not require negative tuples to be generated.

4. UPDATE-PATTERN-AWARE SEMANTICS OF CONTINUOUS QUERIES

4.1 Defining the Meaning of Relations in Continuous Queries

The update pattern classification from Section 3.1 may be used to define continuous query semantics in more detail, including the precise role of relations. In the traditional sense, a relation (or table) is an unordered multiset of tuples with the same schema that supports arbitrary insertions, deletions, and updates. Unless we assume that relations in the context of continuous queries are static, the possibility of arbitrary updates means that relations appear more difficult to process than sliding windows over infinite streams.

To see this, recall the join query in Figure 1, and assume that one of the inputs is a relation and the other is a sliding window. According to Definition 1, an insertion into the table requires a window scan to produce any new join results. Similarly, a deletion from the table requires a window scan to undo previously reported results containing the deleted tuple. That is, negative tuples must be produced on the output stream so that the result corresponds to the current state of the relation. As a result, a join of a table with a sliding window is strict non-monotonic, whereas a join of two sliding windows is only weak non-monotonic.

According to the update pattern classification, any solution that considers updates of relations to be "easier" than insertions and expirations from sliding windows must treat a join of a table with a sliding window as a weakest non-monotonic operator (for simplicity, assume that only the join operator can consume a relation as one of its inputs; we denote such a join by $\bowtie^R$). Given this constraint, our solution is to define a *non-retroactive relation (NRR)* as a table that allows arbitrary updates, but make the following distinction between the semantics of table updates versus the semantics of streams and sliding windows: updates of *NRR*s do not affect previously arrived stream tuples. Consequently, a join of a sliding window and a *NRR*, denoted $\bowtie^{NRR}$, does not need to scan the other state buffer when processing an update of the *NRR*; only arrivals on the streaming input trigger the probing of the *NRR* and generation of new results. Thus, the streaming (or windowed) input does not

have to be stored, and furthermore, $\bowtie^{NRR}$ is monotonic if the second input is a stream and weakest non-monotonic if it is a window.

Aside from being simpler to implement, our definition of *NRR*s is intuitive based on the nature of some of the data stored by DSMSs in relations, namely metadata. For example, an on-line financial ticker may store a table with mappings between stock symbols and company names. In this case, when a financial ticker updates its table of stock symbols and company names by deleting a row corresponding to a company that is no longer traded, all the previously returned stock quotes for this company need not be deleted. Similarly, adding a new stock symbol for a new company should not involve attempting to join this stock symbol with any previously arrived stream tuples, because there are no prior stock quotes for this new company. Formally, we require that an update of a *NRR* at time τ should only affect stream tuples that arrive after time τ.

At this point, we note that the difference between streams, *NRR*s, and relations is purely semantic. It is possible to store metadata (or any other type of tabular data) as a stream if we require insertions to be retroactive to previously arrived tuples, or as a traditional relation if arbitrary insertions, deletions, and updates are to affect previously arrived stream tuples. However, as explained above, the update patterns and implementations of operators that allow retroactive updates are more complicated.

4.2 Continuous Query Semantics

We propose the following definition of continuous query semantics. We define a *base stream* as a data stream generated by a (possibly external) source and a *derived stream* as one produced by an operator or a query. Any base stream can be bounded by a sliding window, whose size may be different for each stream. A continuous query Q references one or more base streams (possibly bounded by sliding windows), zero or more *NRR*s, and zero or more relations, runs over a period of time, and produces the following output.

Definition 2. *Let $Q(\tau)$ be the answer set of Q at time τ and let $\{\mathrm{NRR}_1(\tau), \mathrm{NRR}_2(\tau), \ldots, \mathrm{NRR}_k(\tau)\}$ be the state of each of the k NRRs referenced in Q at time τ. Let $t.ts$ be the generation time of a result tuple t. If Q does not reference any NRRs, i.e., $k = 0$, then $Q(\tau)$ must be equivalent to the output of a corresponding relational query Q' whose inputs are the current states of the streams, sliding windows, and relations referenced in Q. If $k > 0$, then in addition to the above, each result tuple t in $Q(\tau)$ must reflect the following state of the NRRs referenced in Q: $\{\mathrm{NRR}_1(t.ts), \mathrm{NRR}_2(t.ts), \ldots, \mathrm{NRR}_k(t.ts)\}$.*

The output of monotonic queries is an append-only stream, whereas the output of non-monotonic queries (weakest, weak, or strict) is a materialized view that reflects all the "real" (insertions) and negative (deletions) tuples that have been produced on the output stream.

5. UPDATE-PATTERN AWARE QUERY PROCESSING

In this section, we present the design of an update-pattern-aware query processor. As discussed in Section 2, our goal is to decrease processing times and reduce the state maintenance overhead. The two existing query execution techniques only satisfy one of these requirements: the negative tuple approach performs state maintenance efficiently, but performs twice as much processing, whereas the direct approach does not incur processing overhead, but performs state maintenance inefficiently. Our technique satisfies both goals by exploiting the update patterns of query operators.

5.1 Assumptions

All query plans, including all operator state, are assumed to fit in main memory. We concentrate on processing individual queries, though operator state may be shared across similar queries, as in [3]. We represent logical and physical query plans as operator trees. The allowed logical operators were listed in Section 2.1. Additionally, our physical operator algebra includes $\bowtie^R$ and $\bowtie^{NRR}$ from Section 4.1, and an improved implementation of duplicate elimination that we will present in Section 5.3.1. In the remainder of this section, we ignore monotonic queries such as selections over infinite streams, as they do not expire results and therefore do not require update pattern awareness (see, e.g., [6] for optimization techniques for monotonic continuous queries, such as filter reordering).

5.2 Update Pattern Propagation in Continuous Query Plans

The first step towards update pattern awareness is to define the update patterns of continuous queries based on the update characteristics of individual operators. We use our classification from Section 3 to label all the edges in a (physical) query plan with the update patterns generated by the corresponding sub-queries. We abbreviate weakest non-monotonic patterns as *WKS*, weak non-monotonic as *WK*, and strict non-monotonic as *STR*. Recall that *WKS* patterns are the simplest, followed by *WK* and *STR*. To annotate all the edges in the plan with update pattern information, we begin by labeling all the edges originating at the leaf nodes (i.e., sliding windows) with *WKS* and apply the following five rules as appropriate.

1. The output of unary weakest non-monotonic operators and $\bowtie^{NRR}$ is the same as the input.

2. The output of binary weakest non-monotonic operators is *STR* if at least one if their inputs is *STR*, *WK* if the inputs are either *WKS* or *WK*, and *WKS* if both inputs are *WKS*.

3. The output of weak non-monotonic operators except group-by is *STR* if at least one of their inputs is *STR*. Otherwise, the output is *WK*.

4. The output of group-by is always *WK*, regardless of the update patterns of its input.

5. The output of strict non-monotonic operators and $\bowtie^R$ is always *STR*, regardless of the input types.

Rule 1 follows from the fact that weakest non-monotonic operators do not interfere with the order of incoming tuples. Rule 2 applies to merge-union, which also does not reorder incoming tuples, therefore the output patterns correspond to whichever input patterns are more complex. Rule 3 states that *WK* operators produce output patterns at least as complex as *WK*, and possibly *STR* if the inputs contain premature expirations. Rule 4 illustrates the peculiar behaviour of group-by. Even if its input contains negative tuples, its

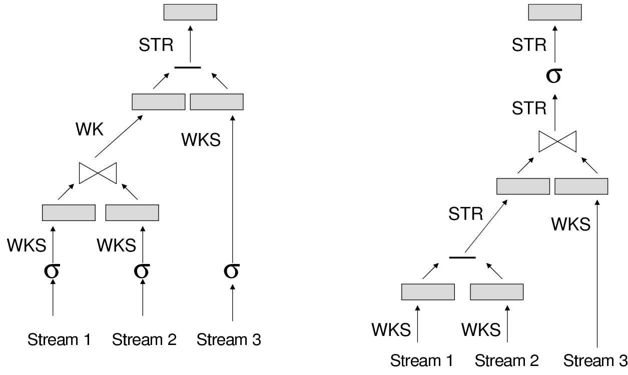

Figure 6: Example of two query plans (for the same query) annotated with update patterns.

output does not. This is due to the assumption that newly generated aggregate values replace old values without explicitly producing negative tuples. Finally, Rule 5 follows from the fact that STR queries and relations whose updates are retroactive may produce unpredictable expirations.

An example of an annotated query plan containing selections, joins, and negation is shown in Figure 6; we will explain the purpose of this query in more detail in Section 6.1. Two equivalent rewritings of the query are depicted. Observe that the two rewritings result in different update patterns on some of the edges.

5.3 Update-Pattern-Aware Physical Optimizations

Given a query plan annotated with update patterns, we propose two techniques to be employed by the update-pattern-aware query processor: using different physical implementations of a given operator depending on the input update patterns, and using update-pattern-aware data structures for maintaining state buffers and storing the final result.

5.3.1 Operator Implementation

Recall from Section 2.1 that the implementation of duplicate elimination proposed in the literature stores both its input and its output [11]. We use this implementation only if the input exhibits STR update patterns. For WKS and WK input patterns, we define a more efficient implementation, denoted by δ^*. The idea is to avoid storing the entire input if we are not expecting premature expirations caused by negative tuples. Instead, for each tuple in the output state, it suffices to additionally store the youngest tuple with the same distinct value (if any); we call this additional state *auxiliary output state*. When a new tuple arrives and does not match any tuples in the stored output, we add the new tuple to the output, as before. However, if the new tuple is a duplicate, it means that it is the youngest tuple with its particular distinct value, and is added to the auxiliary output state. When an output tuple expires, we simply return on the output stream the corresponding youngest tuple stored in the auxiliary state without accessing (and storing) the input. Thus, instead of storing both the input and the output, the space requirement of δ^* is at most twice the size of the output. Since duplicate elimination never produces an output whose size is larger than the input, δ^* is more space-efficient than the existing implementation. Moreover, the time overhead of inserting and expiring tuples is lower.

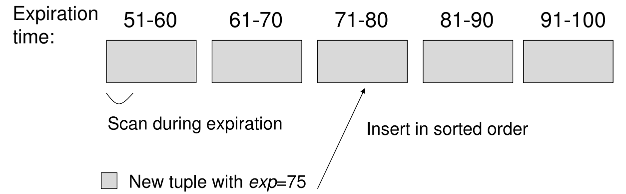

Figure 7: Illustration of our data structure for storing the results of weak non-monotonic subqueries.

5.3.2 Data Structures for Storing Operator State and Final Query Results

The second update-pattern-aware optimization involves using suitable data structures for maintaining state buffers and storing final results. First, we deal with the simple cases: maintaining results whose inputs exhibit WKS patterns and maintaining final group-by results. In the former case, results expire in order of generation, so we can implement the state buffer as a list, with insertions appended to the end of the list and deletions occurring from the beginning. In the latter case, the result consists of aggregate values for each group and can be stored as an array, indexed by group label.

Next, we deal with maintaining operator state with WK input patterns, where the insertion order is different from the expiration order. As discussed in Section 2.3.3, sorting the state buffer by insertion time causes inefficient deletions, whereas sorting by expiration time means that insertions require a sequential scan of the state buffer. Our solution partitions the state buffer by expiration time. Individual partitions can then be sorted by expiration time for operators that must expire results eagerly, or by insertion time for operators with lazy expiration[1]. An example is illustrated in Figure 7 for five partitions (with each partition sorted by expiration time), assuming that the current time is 50 and the window size is 50. Our data structure may be thought of as a circular array of partitions, therefore at time 60, the leftmost partition will contain tuples with expiration times of 101 through 110. Clearly, adding more partitions improves insertion and deletion times (there is less state to scan), but increases the space requirements as each partition is stored as a separate structure. The partitioned state buffer is used, for example, as the materialized view of the join query in Figure 1 and as the left input to the negation operator on the left of Figure 6.

Maintaining results of (sub)queries with STR patterns is difficult because some result tuples may expire at unpredictable times. If premature expirations are rare, we re-use the structure from Figure 7 and periodically incur the cost of scanning all the partitions to perform a deletion triggered by the arrival of a negative tuple. Otherwise, if we are expecting the majority of deletions to occur via negative tuples, then we employ the negative tuple approach. That is, we generate negative tuples for every expiration and sort the state by the negation attribute. The intuition is that if most of the results expire prematurely, then we may as well expire all the results via negative tuples and use a data structure

[1]Our data structure is similar to the calendar queue [8] if we think of expirations as events that are scheduled according to their expiration times.

that makes it easy to do so.

The choice between the two techniques for maintaining results with STR update patterns depends on the frequency of premature expiration, which, in turn, depends on the distribution of the attribute values in the two inputs to the negation operator. For example, if the two inputs have different sets of values of the negation attribute, then premature expirations never happen. To see this, recall the operator description in Section 2.1 and note that negative tuples are produced only if both inputs contain at least one tuple each with a common attribute value.

5.4 Putting it All Together: Update-Pattern-Aware Query Processing

5.4.1 Plan Generation and Cost Model

Each plan is annotated with update pattern information, as in Section 5.2. Whenever duplicate elimination appears in the plan with weakest or weak non-monotonic input, it is replaced with δ^*, as explained in Section 5.3.1. Moreover, each operator that stores state chooses an appropriate data structure for the state buffer, depending on the update patterns of its inputs, as discussed in Section 5.3.2. If a partitioned data structure is used, the number of partitions is initially set to a user-defined default value. Similarly, in the negative tuple approach, the state buffer is a hash table on the key attribute with a user-defined number of buckets.

Each candidate plan is associated with a per-unit-time cost, similar to [10, 14]. The cost includes inserting new tuples into the state, processing them, expiring old tuples, and processing negative tuples, if any. The per-unit-time cost of insertions and expirations from a state buffer depends on its implementation, the input arrival rate, and the frequency of expirations (recall that some operators are permitted to maintain state lazily, but others must expire old tuples eagerly and possibly produce new results). Now, for each operator, we define λ_1 and λ_2 to be its input rates (if the operator is unary, then $\lambda_2 = 0$), λ_o to be its output rate, N_1 and N_2 to be the expected sizes of its inputs (again, $N_2 = 0$ for unary operators), and N_o to be the expected output size. We assume that these quantities may be approximated on the basis of stream arrival rates, attribute value distributions, and operator selectivities.

Selection, projection, and union process each tuple in constant time, therefore their cost is $\sum_i \lambda_i$. Join and intersection cost $\lambda_1 N_1 + \lambda_2 N_2$ per unit time. The cost of δ^* is roughly $\lambda_1 \frac{N_o}{2}$ as every new tuple scans the output, which is sorted by expiration time. Using the negative tuple approach doubles the cost of all of these operators. Letting C be the cost of re-computing an aggregate[2], the cost of group-by is $2\lambda_1 C$ regardless of whether negative tuples are used or not (every tuple changes the value of an aggregate twice: once when it arrives and once when it expires). Finally the cost of negation is at least $2\lambda_1 \log d_1 + 2\lambda_2 \log d_2$, where d_1 and d_2 are the numbers of distinct values in the two inputs, respectively. This assumes that the frequency counts stored by the negation operator are sorted by value and can be binary-searched (recall Section 2.1). Additionally, negation incurs the cost of probing the state of input 1 and generating negative tuples in case of premature expirations.

[2]This depends on the number of groups and the complexity of the aggregate. For instance, simple aggregates such as SUM may be re-computed in constant time simply by adding/subtracting the value of the new/old tuple to the current value of the aggregate.

5.4.2 Query Optimization

Though continuous query plans may be reordered in a way similar to traditional relational plans (e.g., selection push-down and join enumeration), we enforce one constraint: the input to $\bowtie^R$ and $\bowtie^{NRR}$ cannot be strict non-monotonic, therefore it is not possible to push these through a negation. This is because a join involving a relation or a NRR is incapable of processing negative tuples—the "real" tuple that corresponds to the negative tuple may have been deleted or updated, therefore it may be impossible to reproduce the join results involving the negative tuple.

We employ the following two update-pattern-aware optimization heuristics: update pattern simplification and duplicate elimination pull-up. The first rule pushes down operators with simple (weakest non-monotonic) update patterns and pulls up those with more complicated update patterns (particularly strict non-monotonic, i.e., negation). This is done to minimize the number of operators affected by negative tuples (especially joins and duplicate elimination, whose processing costs increase if they have to process negative tuples), and more generally, to reduce the update pattern complexity in the largest possible sub-tree of the plan; see, e.g., Figure 6. Other benefits of update pattern simplification include being able to use δ^* more often and a greater flexibility in reordering $\bowtie^R$ and $\bowtie^{NRR}$. The second rule pushes duplicate elimination below (before) a join so that the output of δ^* can be shared as the input to the join.

These two rules are, for the most part, consistent with well-known relational optimization rules. For example, pushing down weakest non-monotonic operators is analogous to predicate push-down and pushing duplicate elimination below joins is also sensible as duplicate elimination always decreases the cardinality of intermediate results. These similarities are advantageous because they suggest that update pattern awareness can be easily incorporated into relational cost models and optimizers. However, one difference is that relational optimizers typically push down the negation operator if the negation condition is a simple predicate or if it reduces the cardinality of intermediate results. However, in our scenario, it may, in some cases, be cheaper to pull up the negation operator in order to decrease the burden of processing negative tuples.

Once an optimal plan is found, several parameters may be adjusted to determine the amount of memory required by the query. These include the lazy maintenance interval (if the state is maintained lazily, the expiration cost is cheaper, but the expected memory consumption increases) and the number of partitions in the state buffer implementations.

5.4.3 Query Processing

Having found an optimal update-pattern-aware plan as described above, our query execution strategy is as follows. If the query is negation-free, then we use the direct approach, which enjoys reduced processing overhead due to the absence of negative tuples and matches the expiration efficiency of the negative tuple approach due to update-pattern-aware data structures. If the query contains a negation operator, then we have two choices, as outlined in Section 5.3.2. If the expected number of premature expirations is small, then we use the direct approach and employ the

partitioned data structure from Figure 7 to store intermediate and final results. For example, in the plan on the left of Figure 6, the final result is stored using the partitioned data structure. Otherwise, if premature expirations are expected to be frequent, then all the operators below negation use the direct approach without generating negative tuples, but all the operators above negation use the negative tuple approach. For example, applying this approach to the plan on the left of Figure 6 means that the join operator employs the direct approach and does not generate negative tuples, but the negation operator generates a negative tuple for every expiration. Thus, the final result is a hash table on the negation attribute. Using the negative approach is recommended only with negation pull-up. Otherwise, in the context of the plan on the right of Figure 6, the join operator is forced to process a large number of negative tuples. Of course, if the join operator generates a large number of results, but the negation predicate reduces the cardinality of intermediate results, then the update-pattern-aware optimizer is likely to choose the negative tuple approach with negation push-down, depending on the sizes of the intermediate results.

6. EXPERIMENTS

6.1 Overview

We implemented the update-pattern-aware query processor using Sun Microsystems JDK 1.4.1. For comparison, the negative tuple and direct approaches from [11, 12] were also implemented, and are referred to as *NT* and *DIRECT*, respectively. Our technique is abbreviated as *UPA*. Sliding windows and state buffers are implemented as linked lists, or circular arrays of linked lists in case of partitioned data structures. Testing was performed on a Windows XP machine with a Pentium IV 1.8 Ghz processor and 512 Mb of RAM. Query inputs consist of network traffic data obtained from the Internet Traffic Archive at `http://ita.ee.lbl.gov`. We use a trace that contains wide-area TCP connections between the Lawrence Berkeley Laboratory and the rest of the world [19]. Each tuple in the trace consists of the following fields: system-assigned timestamp *ts*, session duration, protocol type, payload size, source IP address, and destination IP address. Furthermore, negative tuples contain a special flag and result tuples have an additional timestamp, *exp*, used to determine the expiration time (recall Section 2.2). Although the trace may be thought of as a single stream, we break it up into several logical streams based on the destination IP addresses. This simulates different outgoing links and allows us to test join queries that combine similar packets from each link.

Five types of queries are tested, the first four of which are illustrated in Figure 8 and the last is as shown in Figure 6. Query 1 joins tuples from two outgoing links on the source IP address, with the selection predicate being either *protocol=ftp* or *protocol=telnet*. The former is a selective predicate (the result size is approximately equal to the size of the inputs), whereas the latter produces ten times as many results (telnet is a more popular protocol type in the trace). Query 1 tests the performance of our partitioned data structure. Query 2 selects the distinct source IP addresses (or the distinct source-destination IP pairs) on an outgoing link and is used to test our improved δ^* operator as well as the partitioned data structure. Query 3 performs a negation of two outgoing links on the source IP address and tests the two possible choices for storing the results of strict non-monotonic queries: using a partitioned data structure or using the negative approach. Query 4 selects the distinct source IP addresses on two outgoing links and performs a join on the source IP address. It is used to test the combined advantage of using partitioned data structures for storing intermediate and final results, and the efficiency of our improved duplicate elimination operator. Finally, Query 5 performs a negation of two outgoing links on the source IP address and joins a third link on the source IP address having *protocol* $= ftp$. That is, Query 5 is essentially a composition of queries 1 and 3. We will test both rewritings of Query 5 illustrated in Figure 6 in order to show that negation pull-up may be efficient in some situations.

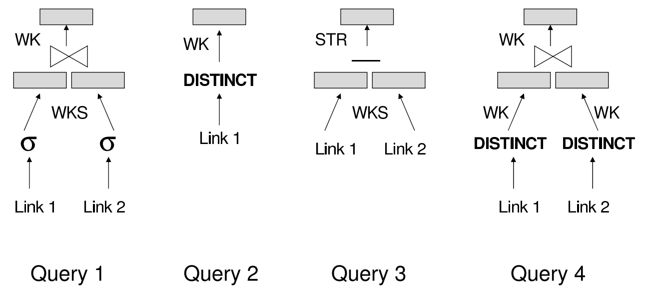

Figure 8: Illustration of the first four query plans used in our experiments.

As mentioned in Section 2, each incoming tuple is fully processed before the next tuple is scheduled for processing. As a result, we fix the stream arrival rates, and ignore queuing delays and processing latencies caused by bursts of tuples arriving at the same time; again, this has been discussed in the context of data stream scheduling [5, 9, 13] and is orthogonal to our work.

There are four experimental parameters: sliding window size, lazy expiration interval (for operators that maintain state lazily), eager expiration interval (for operators such as grouping, duplicate elimination, and negation, which must react to expirations immediately), and the number of partitions in the state buffers. Depending on the query, the window size varies anywhere from 100 Kilobytes to over 10 Megabytes. In terms of time, this corresponds to a range of 2000 to 200000 time units, with an average of one tuple arriving on each link during one time unit. This range allows us to comment on the performance trends of various techniques as the data size grows. For simplicity, the lazy expiration interval is set to five percent of the window size. Increasing this interval gives slightly better performance and is not discussed further. Furthermore, due to the fixed stream arrival rates, we set the eager expiration interval to equal the tuple inter-arrival time. In *NT*, this means that each new arrival into one of the input windows triggers a window scan to determine if any negative tuples must be generated. In *DIRECT* and *UPA*, each new arrival causes a probe of the state of each operator that must immediately react to expirations. Finally, the number of state buffer partitions is set to 10, unless otherwise noted. The reported performance figures correspond to the average overall query execution times (including processing, tuple insertion, and expiration) per 1000 tuples processed.

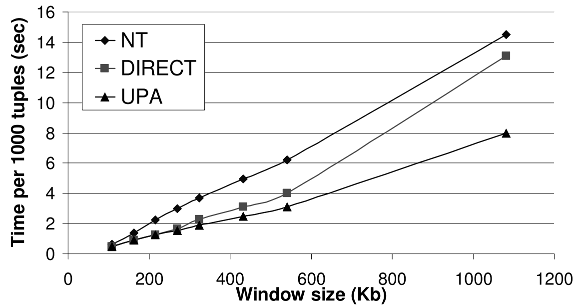

Figure 9: Processing times of Query 1 using $protocol = ftp$ **as the selection predicate.**

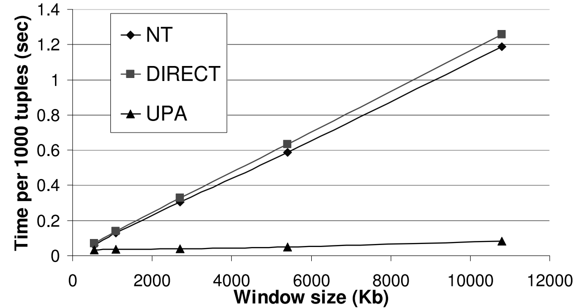

Figure 11: Processing times of Query 2 with duplicate elimination on the source IP address.

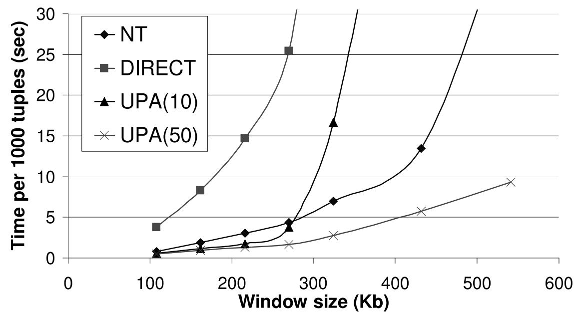

Figure 10: Processing times of Query 1 using *protocol=telnet* as the selection predicate.

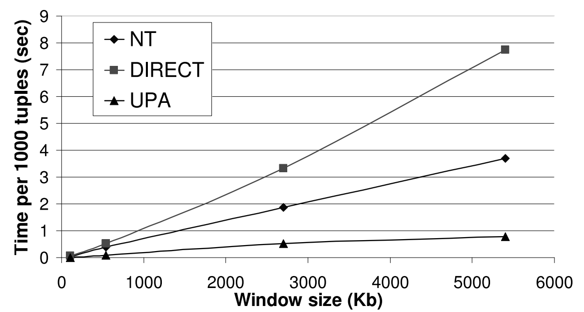

Figure 12: Processing times of Query 2 with duplicate elimination on source and destination IP address pairs.

6.2 Query 1

We begin by testing two variants of Query 1. Figure 9 illustrates the performance of the first variant, which uses *protocol=ftp* as the selection predicate (recall that this is the more selective predicate that produces small result sets). As the window size grows, our approach is nearly twice as fast as the other two. *DIRECT* outperforms *NT* because the result size is relatively small and therefore the cost of scanning the entire result set during updates is not as great as the overhead of negative tuples. However, the performance of *DIRECT* degrades as the window size grows. The second variant of Query 1 is analyzed in Figure 10 and uses *protocol=telnet* as the selection predicate. The result size is approximately ten times as large as in the first variant. In this case, *DIRECT* is by far the slowest because it is very expensive to scan the large result. Our approach with ten partitions (denoted by *UPA(10)*) initially performs well, but becomes very slow as the window size, and the result size, grows. However, increasing the number of partitions to fifty (denoted by *UPA(50)*) yields processing times that are up to one order of magnitude faster than *NT* for large window sizes.

6.3 Query 2

Figure 11 illustrates the processing times of duplicate elimination on the source IP address, whereas Figure 12 graphs the processing times of duplicate elimination on source and destination IP addresses. The former produces a small results set (roughly 2000 distinct IP addresses); the result set of the latter is approximately ten times as large. Combining our efficient implementation of δ^* with update-pattern-aware data structures yields significant performance improvements. In Figure 11, our approach is one order of magnitude faster than the other two. In Figure 12, *UPA* is roughly twice as fast as *NT*. Furthermore, *DIRECT* performs poorly when the result size is large. The reason why *UPA* has a greater performance advantage when the result size is small is because the size of the auxiliary output state is also smaller, and therefore it is faster to maintain and probe (recall Section 5.3.1).

The average space requirements of Query 2 are graphed in Figure 13 (duplicate elimination on the source IP address) and Figure 14 (duplicate elimination on source and destination IP addresses). The former is very selective, therefore our approach is up to two orders of magnitude more space-efficient that *NT* and *DIRECT* (recall that the space requirements of δ^* are proportional to the output size, not the input size). The latter is less selective, but our approach is still significantly more space-efficient.

6.4 Query 3

Figure 15 shows the running time of negation on the source IP address for *NT* and *UPA*; recall that the direct approach is not compatible with negation. Our approach of using a partitioned data structure to store the result slightly outperforms *NT* for window sizes of up to roughly 500 Kilobytes. This is because the result size is small and the penalty for

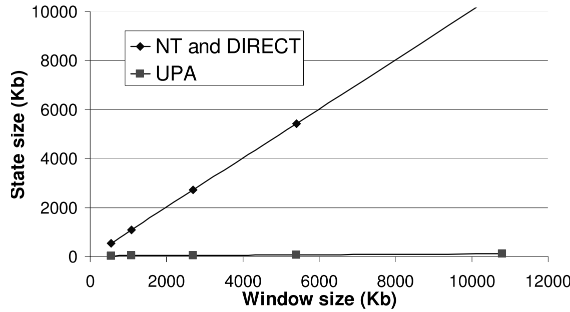

Figure 13: Average space consumption of Query 2 with duplicate elimination on the source IP address.

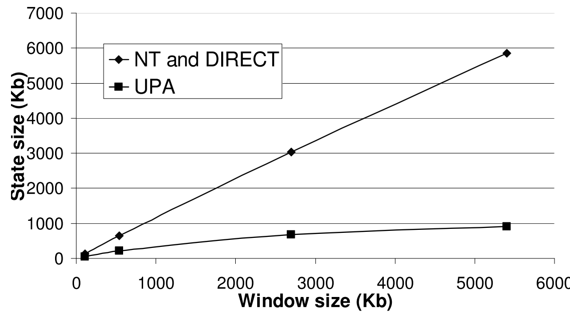

Figure 14: Average space consumption of Query 2 with duplicate elimination on source and destination IP address pairs.

scanning the entire result buffer when expiring a negative tuple is lower than the overhead of generating negative tuples. However, as the window size (and the result size) grows, our approach begins to perform worse because the cost of expiring negative tuples becomes high. We counted the number of premature expirations in Query 3 and found it to be approximately 40 percent of the result set. This is a fairly high proportion, which explains why our approach was competitive only for small window sizes. Nevertheless, these results prove that in some cases, it may be beneficial to use our partitioned data structure instead of the negative approach.

6.5 Query 4

Processing times of Query 4 are shown in Figure 16. As expected, the results are similar to Queries 1 and 2 in that our approach yields a significant performance improvement. Again, the direct approach is the slowest, especially as the window size grows and the result size grows accordingly. In addition to faster processing times, our approach of evaluating Query 4 is more space-efficient because the outputs of the distinct operators are re-used as the inputs to the join, as illustrated in Figure 8. Similar to Figure 13, our approach is up to two orders of magnitude more space-efficient than *NT* and *DIRECT* (exact results are omitted due to space constraints).

6.6 Query 5

In the final test, we execute the two plans for Query 5 il-

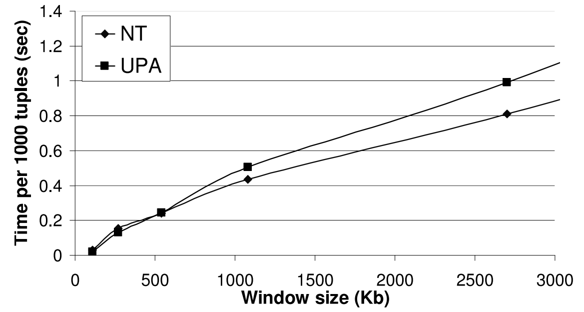

Figure 15: Processing times of Query 3.

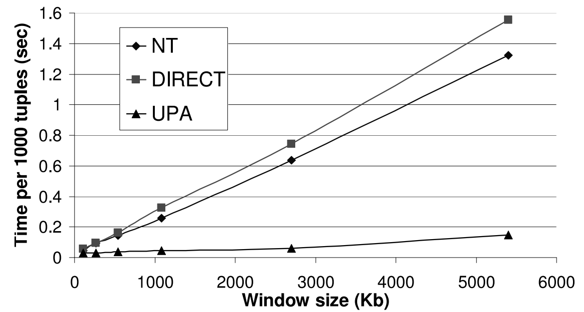

Figure 16: Processing times of Query 4.

lustrated in Figure 6. As discussed in the context of Query 3, negation on the source IP address produces a relatively large number of premature expirations and works best with the negative tuple approach. Recall from Section 5.4.3 that in these cases, we use the negative approach for all the operators above negation in the query plan. That is, the plan on the left of Figure 6 is executed by maintaining the intermediate state directly and maintaining the final result via negative tuples. In the plan on the right, negation is pushed down, therefore our approach is equivalent to the negative tuple approach throughout the plan. Figure 17 shows the processing times of Query 5 for the three possible approaches: negative tuples, negative tuples with the join pulled up (denoted *NT(join up)*), and our approach with the join pushed down, where the join does not generate negative tuples.

First, note that *NT(join up)* outperforms *NT* because the negation operator is more selective than the join and therefore the former plan costs less. However, our approach performs best for sufficiently large window sizes because of the decrease in the number of negative tuples that have to be processed. *NT(join up)* is optimal for small window sizes because the number of negative tuples generated is small and the processing overhead is not as large as the penalty of using a sub-optimal ordering. In general, our approach may not always be advisable. For example, if the join in Query 5 produced a large number of results, then pushing the join below negation would be highly sub-optimal, despite the savings in negative tuple processing. On the other hand, if an ordering with the join pushed down is optimal to begin with, then our approach can make the plan even more

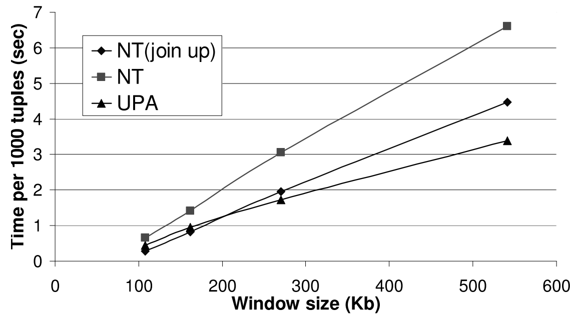

Figure 17: Processing times of Query 5.

efficient by eliminating the overhead of processing negative tuples below the negation operator.

7. CONCLUSIONS AND FUTURE WORK

This paper introduced the notion of update pattern awareness in the context of continuous queries over relations, streams, and sliding windows. We presented a classification of update patterns of continuous queries and applied it to solve two problems: 1) defining precise semantics of continuous queries with a clearly defined role of relations and their update patterns, and 2) efficient query execution over sliding windows. Our experimental results showed significant improvements in query processing times and space consumption achieved by update pattern awareness.

We intend to pursue the following two directions in future work. First, we want to extend our update pattern analysis to queries over count-based windows. The main difference is that expiration times of such queries depend on the arrival rates of the inputs and therefore cannot be predicted. A straightforward solution is to consider all count-based window queries as strict non-monotonic and employ the negative tuple approach in their evaluation. However, there may be special cases where more detailed update pattern awareness is possible. Moreover, we intend to further explore continuous query optimization using update pattern knowledge. One of several possible questions is whether update pattern awareness can also be used to improve the adaptivity of sliding window query plans to changing stream conditions.

8. REFERENCES

[1] D. Abadi et al. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Trans. Database Sys.*, 29(1):162–194, 2004.

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 14(1), 2005, to appear.

[4] A. Ayad and J. Naughton. Static optimization of conjunctive queries with sliding windows over unbounded streaming information sources. *SIGMOD 2004*, pages 419–430.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, 2004.

[6] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. *SIGMOD 2004*, pages 407–418.

[7] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Sys.*, 29(3):545–580, 2004.

[8] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[9] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. *VLDB 2003*, pages 838–849.

[10] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. *VLDB 2003*, pages 500–511.

[11] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. Elmagarmid. Efficient execution of sliding window queries over data streams. Technical Report CSD TR 03-035, Purdue University, 2003.

[12] M. Hammad et al. Nile: a query processing engine for data streams. *ICDE 2004*, page 851.

[13] Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. *BNCOD 2004*, pages 16–30.

[14] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. *ICDE 2003*, pages 341–352.

[15] J. Krämer and B. Seeger. A temporal foundation for continuous queries over data streams. *COMAD 2005*, pages 70–82.

[16] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. *VLDB 2004*, pages 492–503.

[17] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. *ICDE 2002*, pages 555–566.

[18] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. *CIDR 2003*, pages 245–256.

[19] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, 1995.

[20] U. Srivastava and J. Widom. Flexible time management in data stream systems. *PODS 2004*, pages 263–274.

[21] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *SIGMOD 1992*, pages 321–330.

[22] P. Tucker, D. Maier, T. Sheard, and L. Faragas. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowledge and Data Eng.*, 15(3):555–568, 2003.

[23] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-join queries over streaming information sources. *VLDB 2003*, pages 285–296.