Purdue University
Purdue e-Pubs

Computer Science Technical Reports | Department of Computer Science

2005

Exploiting Predicate-window Semantics over Data Streams

Thanaa M. Ghanem

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

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Report Number:
05-007

Ghanem, Thanaa M.; Aref, Walid G.; and Elmagarmid, Ahmed K., "Exploiting Predicate-window Semantics over Data Streams" (2005). *Computer Science Technical Reports.* Paper 1622.
http://docs.lib.purdue.edu/cstech/1622

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

EXPLOITING PREDICATE-WINDOW SEMANTICS OVER DATA STREAMS

Thanaa M. Ghanem
Walid G. Aref
Ahmed K. Elmagarmid

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #05-007
April 2005

Exploiting Predicate-window Semantics over Data Streams

Thanaa M. Ghanem Walid G. Aref Ahmed K. Elmagarmid

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

April 1, 2005

Abstract

The *continuous sliding-window* query model is used widely in data stream management systems where the focus of a continuous query is limited to a set of the most *recent* tuples. In this paper, we show that an interesting and important class of queries over data streams cannot be answered using the *sliding-window* query model. Thus, we introduce a new model for continuous window queries, termed the *predicate-window* query model that limits the focus of a continuous query to the stream tuples that qualify a certain predicate. *Predicate-window* queries have some distinguishing characteristics, e.g., (1) The window predicate can be defined over any attribute in the stream tuple (ordered or unordered). (2) Stream tuples qualify and disqualify the window predicate in an out-of-order manner. In this paper, we discuss the applicability of the predicate-window query model. We will show how the existing *sliding-window* query models fail to answer some of the *predicate-window* queries. Finally, we discuss the challenges in supporting the *predicate-window* query model in data stream management systems.

1 Introduction

The emergence of data streaming applications calls for new query processing techniques to cope with the high rate and unbounded nature of data streams. Queries over data streams are characterized by the following: (1) Most of the queries in the streaming environment are continuous. Continuous queries need continuous reevaluation as new tuples arrive, and (2) Usually, queries are interested only in a specific part (*window-of-interest*) of the received data. The sliding-window query model [1] is introduced to answer continuous queries that are interested only on the most recent stream tuples. There are two common types of sliding-windows: Time-based sliding window (e.g., tuples in the last hour) and tuple-based sliding window (e.g., the last 100 tuples). Window-aware operators (e.g., window-join [3, 5, 6] and window-aggregates [7]) 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 new tuples incoming into the window.

1.1 Motivation

Continuous *sliding-window* queries over data streams have been introduced to limit the focus of a continuous query to a specific part (*window-of-interest*) of the incoming stream tuples. The *window-of-interest* in the sliding-window query model includes the most-recent input tuples. In a sliding-window query over n input streams, S_1 to S_n, a window of size w_i is defined over the input stream S_i. The sliding-window w_i can be defined over any ordered attribute *attr* in the stream tuple (e.g., a timestamp or a sequence number). As the window slides, the query answer is updated to reflect both the new tuples entering the

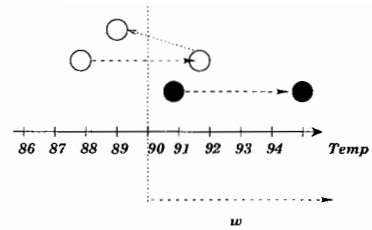

Figure 1: Example 1

sliding-window and the old tuples expiring from the sliding-window. Tuples enter and expire from the sliding-window in a First-In-First-Expire (FIFE) fashion.

An interesting and important class of queries is not supported by the sliding-window query model. Consider a continuous query that is interested only in the input tuples that qualify a certain predicate p, where p is defined over an unordered attribute. For example, consider a temperature monitoring application in which a large number of sensors are spatially distributed, and each sensor sends continuously its current temperature. A common query in this environment is: Q_1 *"Continuously, report the sensor identifiers for sensors that have temperature greater than 90"*. At any time point T', the window-of-interest for Query Q_1 includes only the sensors that qualify the predicate "*temperature greater than 90*". If a sensor S reports a temperature greater than 90, then S should be considered in Q_1's window. Whenever S reports another temperature that disqualifies the predicate "*temperature greater than 90*", S expires from Q_1's window. Notice that sensors enter and expire from Q_1's window in an out-of-order manner. A sensor expires (and hence is deleted) from Q_1's window only when the sensor reports another temperature that disqualifies the window predicate.

To utilize the sliding-window query model, the query semantics reads as follows: Q_2: *"Continuously, report sensor identifers for sensors that have temperature greater than 90 in the last T time units"*, where T is the size of the sliding-window. The Query Q_2 is semantically different from Query Q_1. The main difference between the two queries is that the window-of-interest in Q_1 includes "*sensors having temperature greater than 90*" while the window-of-interest in Q_2 includes "*sensors that have reported temperature greater than 90 in the last T time units*".

Example 1: This example illustrates the difference between Q_1 and Q_2 (with T=5). Consider the temperature monitoring application where the input stream has the schema <SensorID, Temperature, TimeStamp>. Assume that the following input tuples have arrived <2,88,1> <2,92,2> <3,91,3> <1,95,4> <2,89,5> <3,95,6>. Q_1 and Q_2 produce the following output: (1) When tuple <2,92,2> arrives, Sensor 2 is reported in the answer. Similarly, when tuple <3,91,3> arrives, Sensor 3 is reported in the query answer. Later, when tuple <2,89,5> arrives, Sensor 2 expires (is deleted) from the answer. On the other hand, when tuple <3,95,6> arrives, Sensor 3 is not deleted from the query answer since Sensor 3 still qualifies the window predicate only with a different temperature. Figure 1 gives the behavior of Sensors 2 (white circles) and Sensor 3 (black circles) in query Q_1. (2) In Query Q_2, when tuples <2,92,2> and <3,91,3> arrive, Sensors 2 and 3 are reported in the answer. Later, when tuple <2,89,5> arrives, the answer will not be affected since temperature 89 disqualifies the predicate. When tuple <3,95,6> arrives, Sensor 3 will be reported again in the query answer. To summarize, at time 6, the answer to Q1 is Sensors 3 and 1, because these are the sensors with temperature greater than 90 at time 6. In contrast, the answer to Q2 is Sensors 2, 3 and 1. Sensor 2 appears in the answer of Q2, because Sensor 2 reports a temperature greater than 90 once in its history in the past 5 time units. Notice that Sensor 2 will expire from Q_2 at time 7 (when tuple <2,92,2> is 5 time units old).

1.2 The Negative Tuples Approach

In the rest of this paper we assume that the pipelined query execution model with the negative tuples approach [1, 2] is used to process *window* queries over data streams. The pipelined query execution model for data streams is a modification of the one used in traditional database management systems [2] where all query operators are connected via first-in-first-out queues. In the negative tuples approach, a special operator, termed EXPIRE, is added at the bottom of the query pipeline; one EXPIRE operator per data stream. EXPIRE buffers the input stream tuples, and outputs a negative tuple whenever a tuple expires from the window. The negative tuple is processed by the various operators in the query pipeline to negate the effect (if any) of the corresponding positive tuple. The output of the continuous query is a continuous stream of positive and negative tuples. A negative tuple is interpreted as a deletion of a previously produced positive tuple.

2 The Predicate-window Query Model

The predicate-window query model is a generalization over the sliding-window query model where the former supports a larger class of continuous queries over data streams. The window-of-interest for the predicate-window includes the input stream tuples that satisfy a given predicate.

Assumptions: In the predicate window query model, we have the following assumptions:

- Each input stream tuple t has a *correlation* attribute $t.CORAttr$. The input stream tuples with the same value of the correlation attribute are correlated together as follows. If a later tuple t_n carries the same values of the correlation attribute as that of t, then t_n is considered an update over t. In Example 1, the correlation attribute is $SensorID$. Therefore, tuple <2,89,5> is an update over tuple <2,91,2>.
- There is no regular pattern for updates. In Example 1, some sensors may send their readings every fixed time interval and some other sensors send their readings whenever a change in temperature is detected.

2.1 Continuous Predicate-window Query Semantics

A predicate-window query Q is defined over n data streams S_1 to S_n and n window predicates P_1 to P_n where the window predicate P_i is defined over the tuples in stream S_i. At any point in time T, the answer to Q equals the answer to a snap-shot query Q', where Q' is issued at time T and the inputs to Q' are the tuples in stream S_i that qualify the predicate P_i at time T.

Assume that an input tuple t_i from stream S_i has the following schema: $t_i < CORAttr, PAttrs, Attrs >$, where $CORAttr$ is the correlation attribute, $PAttrs$ are the attributes over which the predicate P_i is defined and $Attrs$ are the other attributes. A tuple t_i qualifies the predicate P_i at time T, iff: (1) t_i arrives in the stream at point in time before T, (2) $t_i.PAttrs$ qualifies P_i and (3) There is no stream tuple t'_i that arrives after t_i and $t'_i.ID = t_i.ID$.

2.2 Syntax and Operators

We represent the predicate-window by adding a new construct, termed PWINDOW, to SQL. The syntax for PWINDOW is as follows:

PWINDOW <*predicate*> *ON* <*CORAttr*>

where <*predicate*> is the predicate that qualifies (and disqualifies) tuples into (and out of) the window and <*CORAttr*> is one or more attributes that correlate incoming stream tuples.

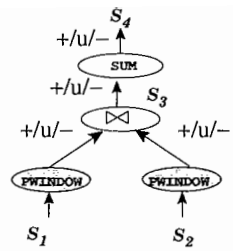

Figure 2: The PWINDOW operator

Example 1 revisited: The following is the SQL syntax for the query Q_1 in Example 1:
SELECT S.SensorID
FROM Sensors S
[*PWINDOW S.Temperature > 90 ON S.SensorID*]

A new operator PWINDOW needs to be incorporated in the stream query engine. The PWINDOW operator is a generalization of the EXPIRE operator. PWINDOW is placed at the bottom of the query pipeline (Figure 2). PWINDOW encapsulates the window predicate (or multiple predicates) and applies it on every incoming stream tuple. PWINDOW is responsible for notifying the query pipeline by any changes in the window contents. PWINDOW is a statefull operator that needs to keep all tuples currently in the window in its state H. PWINDOW produces three different types of tuples:

1. **Positive Tuple (t^+):** When a new incoming stream tuple t qualifies the window predicate and $t.CORAttr$ is not in H, PWINDOW inserts t in H and output a positive tuple for t.
2. **Update Tuple (t^u):** When a new incoming stream tuple t qualifies the window predicate and $t.CORAttr$ is already in H, PWINDOW updates the attributes of t in H and produces an update tuple for t as output.
3. **Negative Tuple (t^-):** When a new incoming stream tuple t does not qualify the window predicate and $t.CORAttr$ is in H, PWINDOW deletes t from H and produces a negative tuple for t as output.

Different operators in the query pipeline will be furnished by methods to process the different types of tuples. The output of the query is a stream of positive, update, and negative tuples. An update tuple is interpreted as a replacement for the previously produced positive tuple with the same tuple identifier. The negative tuple is interpreted as a deletion of the previous positive (or update) tuple with the same tuple identifier.

3 A Comparison with the Existing Window Approaches

In this section, we show how the existing sliding-window query approaches fail to answer some of the predicate-window queries. We use query Q_1 (from Example 1) as a running example.

3.1 WHERE Clause

The main difference between a predicate in PWINDOW and a predicate in the *where-clause* is that a disqualified tuple in the PWINDOW predicate may result in a negative tuple as an output while a disqualified tuple in the *where-clause* predicate does not result in any output tuples. We illustrate the

difference between the *window* predicate and the *where* predicate by the following example. Consider a SQL query that is similar to Q_1 but the *window* predicate is expressed in the *where* predicate as follows:
SELECT S.SensorID
FROM Sensors S
WHERE S.Temperature > 90
If this query is continuously running as the stream tuples arrive, at time 2, when tuple <2,92,2> arrives, Sensor 2 is reported in the query answer. Later, when tuple <2,89,5> arrives, and since the temperature 89 disqualifies the *where* predicate, tuple <2,89,5> does not affect the query answer. The output from the SQL query with the *where* predicate is different from the expected output of Q_1. In Q_1, although tuple <2,89,5> disqualifies the window predicate, tuple <2,89,5> results in an output negative tuple to expire Sensor 2 from the query answer.

The *where* predicate cannot be used to express predicate-window queries. When a tuple t qualifies the *where* predicate and is reported in the query answer, t will remain in the query answer for ever. In the predicate-window query model, when a tuple t qualifies the window predicate and is reported in the query answer, later, t may be deleted from the query answer if t receives an update so that t does not qualify the window predicate any more.

3.2 Sliding-windows

The sliding-window query model fails to answer some of the predicate-window queries (as shown in Example 1). The sliding-window query model is characterized by the following: (1) A window with size w is defined over an ordered attribute in the stream schema (e.g., a timestamp or a sequence number) and (2) Tuples enter and expire from the sliding-window in a First-In-First-Expire (FIFE) fashion. Some of the predicate-window queries do not follow the characteristics of the sliding-window query model. For example, consider query Q_1. The window predicate is defined over the unordered attribute temperature. There is no window size for the sliding-window that can emulate the behavior of Q_1. Moreover, in Q_1, tuples enter and expire from the predicate-window in an out-of-order manner. A tuple expires from the predicate window whenever the tuple receives an update that disqualifies the window predicate. Due to the different characteristics, some of the predicate-window queries cannot be answered using the sliding-window query model.

3.3 Partitioned Sliding-windows

Partitioned sliding-window queries have been introduced and used by several data stream management systems [1, 7]. A partitioned sliding-window partitions the input stream into sub-streams and the sliding-window is applied on each sub-stream independently. The windows of the various sub-streams are merged to produce the final query answer. The CQL clause for the partitioned-window is as follows:
PARTITIONBY <PARAttr>
ROWS <w>
WHERE <predicate>
where *<PARAttr>* is the partitioning attribute, *<w>* is the sub-stream sliding-window size, and *<predicate>* is an optional window filter.

The "*partition by <PARAttr>*" in the partitioned-window clause is similar to the "*ON <CORAttr>*" in the PWINDOW clause. The two clauses aim to group input stream tuples having the same value of some attribute. Although having some similarities, we show that partitioned sliding-windows fail to answer some predicate-window queries.

A partitioned sliding-window query may have two classes of predicates as follows: (1) Partitioned-window predicates (*where <predicate>* in the PARTITION BY clause) and (2) Query predicates (the outer *where* clause in the query). The difference between the partitioned-window predicate and the query

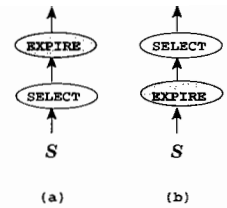

Figure 3: Partitioned sliding-window

predicate is as follows. The partitioned-window predicate qualifies (and disqualifies) tuples into (and out-of) the window for each sub-stream. In this case, the window size is calculated over the qualified tuples only. For example, if the window size is 3, then at any time point, the last 3 qualified tuples will be inside the window of the corresponding sub-stream. On the other hand, the query predicate qualifies (and disqualifies) tuples into (and out-of) the query answer. In this case, the window size is calculated over both qualified and disqualified tuples. For example, if the window size is 3, the last 3 tuples will be inside the window of the corresponding sub-stream. From these last 3 tuples, only the qualified tuples will be used in the query answer.

In the following, we show that both the partitioned-window query with window predicates and the partitioned-window query with query predicates are semantically different from the window predicate in the predicate-window query model.

3.3.1 Partitioned-window predicates

A partitioned-window clause partitions the stream into sub-streams. A partitioned-window predicate qualifies (and disqualifies) tuples into (and out-of) each sub-stream. Assume that a partitioned sliding-window Q_p query that is similar to Q_1 but with the window predicate "*temperature > 90*" is used as the partitioned-window filter. The CQL syntax for Q_p is as follows: *SELECT S.SensorID*
FROM Sensors S
[*Partition By S.SensorID*
Rows 1
WHERE S.Temperature > 90]

The semantics of the query Q_p is as follows: *"For each sensor, continuously report the last reading with temperature > 90"*. The query pipeline for Q_p is shown in Figure 3a, where the window filter (the *select* operator) is applied before the window size (the *EXPIRE* operator). Let Q_p be a continuously running while the stream tuples arrive. At time 2, when tuple <2,92,2> arrives, Sensor 2 is reported in the query answer. Later, when tuple <2,89,5> arrives, since 89 disqualifies the selection filter, tuple <2,89,5> is filtered out and does not contribute to the window for Sensor 2 sub-stream. Tuple <2,92,2> expires from the window only when Sensor 2 reports another reading with temperature greater than 90. Notice that the output of Q_p is different from the output of the predicate-window query Q_1. In Q_1, when tuple <2,89,5> arrives, Sensor 2 expires from the query answer.

The window filter in Q_p is different from the window predicate in Q_1 in the following: the window filter in Q_p qualifies (and disqualifies) tuples into (and out of) the sub-streams. On the other hand, the window predicate in Q_1, qualifies (and disqualifies) sub-streams into (and out of) the query answer.

3.3.2 Query predicates

The other type of predicates in the partitioned-window query is the query predicate. The query predicate in a partitioned-window query qualifies tuples into the query answer. The window for each sub-stream may include both qualified and disqualified tuples. Consider a partitioned-window query Q_p' similar to Q_1 but with the window predicate used as a query predicate as follows:
SELECT S.SensorID
FROM Sensors S
[*Partition By S.SensorID*
ROWS 1]
WHERE S.Temperature > 90

The semantics for Q_p' is as follows: "*Continuously report the readings with temperature greater than 90 considering only the last reading for each sensor*". The pipeline for query Q_p' is given in Figure 3b where the window size (the *EXPIRE* operator) is applied before the query filter (the *select* operator). Assume that query Q_p' is continuously running when the stream tuples arrive. At time 3, tuple <3,91,3> arrives to the EXPIRE operator. Since it is the most recent reading for Sensor 3, tuple <3,91,3> will be forwarded to the *select* operator. Since 91 qualifies the selection predicate, Sensor 3 is produced in the query answer. Later, at time 6, tuple <3,95,6> arrives. Upon receiving <3,95,6>, since only the last reading for each sub-stream resides inside the window, the EXPIRE operator will emit two tuples: a negative tuple -<3,91,3> and a positive tuple <3,95,6>. Both tuples will be passed to the *select* operator. Both -<3,91,3> and <3,95,6> appear in the query answer. Notice that the semantics of the reception of -<3,91,3> then <3,95,6> is that Sensor 3 is deleted from the answer then Sensor 3 is reported again in the answer.

Q_p' query answer (including the deletion then insertion of Sensor 3) is semantically different from the predicate-window query Q_1. The semantics of the predicate-window query requires that at any point in time, the query answer includes all sensors satisfying the window predicate. Q_1 semantics does not hold in the time interval between the deletion and insertion of Sensor 3 in Q_p' window. The length of the time period for the semantically wrong answer is non-deterministic since tuples may encounter delays in the query pipeline. The problem can be worse if an aggregate operation (e.g., COUNT) is performed over the output tuples. Assume that another query is interested in the COUNT of sensors having temperature greater than 90. Assume that before tuple -<3,91,3>, the COUNT value was 10. Upon receiving -<3,91,3>, the COUNT operator will update its answer to 9. When tuple <3,95,6> is processed by the COUNT operator, a new count with value 10 will be produced. Notice that the count of value 9 should not appear in the query answer.

The previous examples shows that in the partitioned-sliding-window, the independent application of the partitioned-window clause and the where-clause is semantically different from the predicate-window query. The reason is that the independent evaluation of the window and the query predicates cannot capture the case of a tuple still being inside the window but only with a different value.

3.4 The NOW window

The keyword NOW defines a window over a data stream [1]. The NOW window means that at any point in time, say T, the answer of the query should include only the tuples that have a timestamp T. The NOW window is semantically different from the predicate-window query. Consider a query Q_n that is similar to Q_1 with the NOW window. The CQL syntax for Q_n is as follows:
SELECT S.SensorID
FROM Sensors S [*NOW*]

WHERE S.Temperature > 90
The semantics for Q_n is as follows: "*Report the sensors that have reported temperature greater than 90 NOW*". Assume that Q_n is continuously running when the input stream tuples arrive. At time 2, the query answer will include only Sensor 2 (because of the arrival of tuple <2,92,2>). Similarly, at time 3, the window will include only Sensor 3.

Query Q_n's answer is different from Q_1's answer. At any time point T, the NOW window includes only tuples that arrive at time T. On the other hand, at any time T, the predicate-window may include tuples that have arrived before T.

3.5 Punctuations

A punctuation is an artificial tuple, carrying a predicate p, that is inserted in the data stream to mark the end of a subsequence [9]. A punctuation tuple with predicate p arriving at time T means that no more tuples qualifying p will arrive later (after time T) in the input stream. The punctuation predicate does not carry any information about the input stream tuples that have arrived before time T and already have been used in generating the query answer. The tuples used in generating the query answer before the arrival of a punctuation p may include both tuples qualifying p and tuples disqualifying p.

The punctuation predicate cannot be used to represent the window predicate in the predicate-window query model. The reason is that before the arrival of a punctuation p, tuples disqualifying p may be included in the window-of-interest of a query. On the other hand, a window predicate, say wp, ensures that, at any time point, only tuples qualifying wp are included in the window-of-interest of the query. Due to the different semantics, punctuations fail to evaluate predicate-window queries.

4 Types of Predicate-window Queries

The window predicate can take several other forms other than the selection predicate in Query Q_1. In this section, we discuss the various types of predicate-window queries.

4.1 Select predicate-window

In the *select* predicate-window type, the window predicate is a selection predicate that is defined over one attribute in the input stream. The selection predicate compares the incoming stream tuple against a constant (e.g., Temperature > 90).

4.2 Join predicate-window

The join predicate-window is a generalization of the select predicate-window. The join window predicate is defined over an attribute in the input stream tuple and compares the incoming stream tuple against a set of constants stored in a relational table.

Example: Consider the following scenario: Persons wearing RFID's are moving inside a building. Each RFID continuously reports the RoomID of the current location. Consider the following query: *"Continuously report the identifiers of persons located in one of the AlertRooms"*. The pre-defined set of alert rooms is stored in a relational table $AlertValues$. The window predicate in this query is a join predicate between the incoming stream and the $AlertValues$ table.

4.3 Dynamic predicate-window

In the *select* and the *join* predicate-windows, the *window predicate* is fixed and the updates cause tuples to qualify into (or disqualify from) the window. The *dynamic* predicate-window is another type of predicate-windows in which tuples expire from the window because the window predicate is continuously changing (e.g., current time).

Example: A sliding-window query is a dynamic predicate-window in which the window predicate is defined over the timestamp attribute. Consider the following sliding-window query: "*Continuously report the sensor identifiers for sensors that have reported a reading in the last T time units*". The same query can be rephrased as "*Continuously report the sensor identifiers for tuples that have a timestamp greater than* $NOW - T$". The SQL representation for this query is:
SELECT S.SensorID
FROM Sensors S
PWINDOW $S.TimeStamp > NOW - T$ *ON S.TimeStamp*

4.4 IN/OUT predicate-window

In the previous sections, we introduced the predicate-window query model with one predicate defined in the PWINDOW clause. In this section, we introduce an extended predicate-window query model, namely the IN/OUT predicate-window model. The main idea in the IN/OUT predicate-window model is to distinguish between two predicates: (1) ***IN window-predicate***: tuples that qualify the IN window-predicate will be considered by the query. (2) ***OUT window-predicate***: when a tuple currently in the predicate-window qualifies the OUT window-predicate, then that tuple will expire from the window. The IN and OUT window predicates are different and are independent. The two predicates should not have any overlap (no stream tuple can satisfy both the IN and OUT predicates at the same time). In the predicate-window query model, the OUT window-predicate (implicitly) is the *complement* of the IN window predicate.

5 Challenges in Realizing Predicate-windows in Data Stream Management Systems

In this section, we discuss the challenges in realizing the predicate-window query model in data stream management systems.

5.1 Incremental Maintenance of the Query Answer

As discussed in Section 2.2, the PWINDOW operator is responsible for tracking changes in the window and emitting tuples accordingly (positive, update, or negative tuples). The output tuples from PWINDOW flow in the query pipeline and are processed by the various operators. The results of processing these tuples by the various operators are used to update the query answer incrementally. For each relational operator and for each tuple type, the following should be specified: (1) The actions to be taken by the operator to process the input tuple, (2) the changes in the operator's state (if any) due to the processed tuple, and (3) the output of the operator.

The incremental maintenance of continuous predicate-window queries is different from the traditional incremental query maintenance. The incremental evaluation of continuous queries in traditional databases has been addressed in Tapestry [8] and the maintenance of materialized views [4]. Tapestry addresses *append-only* queries in which an output tuple will remain in the query answer forever. Unlike Tapestry, the output tuple of a predicate-window query may be updated or deleted. On the other hand, materialized

(a) (b)

Figure 4: Object Update Pattern

views deal with data resident on disk and the query answer is materialized. In materialized views, changes to the base tables are reflected into the materialized view via incremental maintenance algorithms [4]. Unlike materialized views, both the input to and output of the predicate-window query is a stream of tuples.

Long-living tuples: Unlike sliding-windows, a tuple entering the predicate-window may remain in the window for long periods of time. We call the tuples that do not expire from the predicate-window as "*long-living-tuples*". The number of tuples inside a predicate-window can grow unboundedly due to long-living tuples. Limiting the number of tuples inside a predicate-window is an interesting area of research.

5.2 Predicate Selectivity

For the window predicate, two different selectivities can be distinguished: positive selectivity and negative selectivity. The positive selectivity is defined the same as the traditional selectivity definition. Positive and update tuples will contribute to the positive selectivity of the window predicate. Negative tuples emitted from the window predicate will contribute to the negative selectivity. The negative selectivity can be defined as the selectivity of the OUT predicate in the predicate-window query. The OUT predicate can be implicit as the complement of the window predicate or explicit as in the IN/OUT predicate-window query model.

Positive and negative selectivities are illustrated by Figure 4. Given query Q_1 as in Example 1, the OUT predicate in this query is (implicitly) the complement of the window predicate "temperature greater than 90". Figure 4 gives the input of two different sensors to the PWINDOW operator. The circles in the figure represent the input to the PWINDOW operator. The *white* circles represent positive or update output tuples, the *black* circles represent negative output tuples, and the *dashed* circles represent filtered out inputs. The two PWINDOW operators have the same number of input tuples (11 tuples) and the same number of positive/update tuples (5 tuples) but a different number of negative tuples (black circles). The negative selectivity of the query depends on the update pattern of the input tuples. Estimating the selectivity of the window predicate is critical for query optimization. Estimating the positive and negative selectivities of the window predicate is another interesting area for future research.

5.3 Shared Execution of Predicate-window Queries

Applications over data streams always involve a large number of concurrent continuous queries over the same data. Queries must be handled collectively by exploiting similarities and sharing resources such as computation, memory, and disk bandwidth among the queries. The PWINDOW operator is a new operator introduced by the predicate-window query model. Sharing the PWINDOW operator among several predicate-window queries can greatly improve the performance of the query processing engine. Efficient techniques for sharing the PWINDOW is an interesting area for future research.

6 Conclusion

In this paper, we proposed the predicate-window query model as a general model for window queries over data streams. Examples are discussed to illustrate how the existing sliding-window query approaches fail to answer some of the predicate-window queries. Moreover, the predicate-window query model can emulate the behavior of the sliding-window query model. We discussed several challenges and open research issues that need to be thought of for efficient realization of the predicate-window query model inside a data stream management system.

7 Acknowledgment

This work was supported in part by the National Science Foundation under Grants IIS-0093116, IIS-0209120, and 0010044-CCR.

References

[1] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University, October 2003.

[2] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Query Processing using Negative Tuples in Stream Query Engines. Technical Report 04-040, Purdue University, November 2004.

[3] L. Golab and M. T. Ozsu. Processing Sliding Window multi-joins in Continuous queries over Data Streams. In *VLDB*, 2003.

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

[5] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. In *SSDBM*, 2003.

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

[7] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *SIGMOD*, 2005.

[8] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, pages 321–330, 1992.

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