

# Supporting Predicate-Window Queries in Data Stream Management Systems

Thanaa M. Ghanem

ghanemtm@cs.purdue.edu

Department of Computer Science, Purdue University, West Lafayette, IN 47907-1398

## Abstract

*The window query model is widely used in data stream management systems where the focus of a continuous query is limited to a set of the most recent tuples. In this dissertation, we show that an interesting and important class of continuous queries can not be answered by the existing sliding-window query models. Thus, we introduce a new model for continuous 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 windows are characterized by the following (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. The goal of this dissertation is to develop an efficient framework to realize predicate windows inside data stream management systems. The predicate-window query framework enables the system to efficiently support a wide variety of streaming applications through an expressive query language and efficient query evaluation mechanisms (i.e., query execution and query optimization). As a test bed for our research, the predicate-window framework is being developed inside Nile; a prototype data stream management system developed at Purdue University.*

## 1. Introduction

The emergence of data streaming applications calls for new query semantics and processing techniques to cope with the unbounded nature of streams. Examples of streaming applications include: building monitoring, electronic trading, and sensor applications. Unlike the traditional queries, queries over data streams have the following distinguishing characteristics: (1) Queries are continuous in nature and require continuous evaluation. (2) A large number of streams and continuous queries are running concurrently in the system. Several research efforts have developed semantics and languages for continuous queries, e.g., [3, 6, 7, 8, 11, 19]. Most of these efforts have recognized the need for *windows* to express queries over the unbounded streams. Windows are used in various ways, for ex-

ample, windows may be assigned to streams [3, 8] or to operators (e.g., window join and window aggregates) [7, 19].

In this dissertation, we propose the predicate-window query framework as a general and efficient framework for continuous queries over data streams. The predicate-window query model distinguishes itself from the existing window query models by the following. (1) Predicate-window queries can express an interesting and important class of queries that is not supported by the existing window query models. (2) Predicate-window semantics employs minimal extensions over the traditional SQL semantics and is independent from the runtime stream environment. (3) The predicate-window query model is a generalization of the existing window models since all types of windows (e.g., window per stream, or window per operator) can be expressed as predicate windows.

In [17], we show that an important class of continuous queries can not be expressed using any of the existing continuous query languages. This important class includes, for example, continuous queries in which the input streams do not represent append-only relations. 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, report the modifications in the answer every two minutes*”. The schema of the input stream in  $Q_1$  contains two attributes: sensor identifier and temperature. The sensor identifier attribute is considered as a key and an input stream tuple is an update over the previous tuple with the same key value. In the same time, the answer of  $Q_1$  is modified every two minutes to include only the sensors that qualify the predicate “*temperature greater than 90*”. The modifications in the answer consist of *insertions* of sensors that report temperatures greater than 90 and/or *deletions* for previously qualified sensors that disqualify the predicate due to a change in the temperature. Notice that sensors are inserted and deleted from  $Q_1$ ’s answer in an out-of-order manner.

In the previous query,  $Q_1$ , the input and output streams are viewed as sequences of modify operations (e.g., insert, update, and delete operations) over the underlying sensors’

temperatures. Streams of modify operations are common in streaming environments in which objects continuously send updates to their values (e.g., building monitoring and tracking moving objects). In [17], we show that the Query,  $Q_1$ , can not be expressed using any of the window query models that restrict the stream definition to the representation of an append-only relation [4, 7]. Moreover, with the wide spread of streaming applications, streams may have several denotations other than the append-only and update streams [16]. For example, a stream may denote the concatenation of serializations of a relation.

The “predicate-window” query semantics define the stream as a sequence of modify operations over a specified relation. A predicate-window query does not use specific constructs to define windows. Instead, a window predicate is defined as a regular SQL predicate in the *where* clause of the query. Moreover, window predicates can be expressed over any attribute (ordered or non-ordered) in the stream tuple. Basically, a predicate-window query is semantically equivalent to a *materialized view* over relations that are modified by *streams* of modify operations. The query is expressed by a regular SQL expression and the query answer is refreshed whenever either any of the input streams is modified or the query definition is updated. The modification in the answer is represented as another stream of modify operations in a way similar to incremental maintenance of materialized views. Although similar in many aspects, the predicate-window query semantics has to deal with some issues that are not handled in the traditional materialized views. For example, a predicate-window query definition may include a refresh period to indicate how often the query answer should be refreshed (e.g., *refresh the answer every two minutes*). Notice that, the refresh period is equivalent to the *slide* parameter in the sliding-window query model [3, 15].

In this dissertation, we propose a framework to support predicate-window queries inside data stream management systems through the following four contributions:

1. Defining concise and expressive semantics for predicate-window queries.
2. Developing an algebraic framework for the proposed semantics. The algebraic framework gives a systematic and principled structure for efficient evaluation of continuous queries.
3. Designing efficient query optimization and shared query execution algorithms that are based on the algebraic framework.
4. Developing efficient execution plans and physical operator implementations to realize predicate windows inside Nile [1]; a prototype data stream management system developed at Purdue University.

## 2. The PhD Contributions

In this section, we briefly present the four parts of the proposed framework to support predicate-window queries over data streams.

### 2.1. Predicate-window Semantics

We develop the predicate-window semantics with the following goals in mind:

- **Expressiveness.** The predicate-window semantics can express a wide variety of continuous queries over the various stream denotations (e.g., append-only relations, or modifications over a relation) and the various window semantics (e.g., window per stream, window per operator, or window over an unordered attribute).
- **Minimal extensions to SQL semantics.** The predicate-window semantics is clear, well defined, and easy to be understood where it employs minimal extensions over the traditional SQL semantics.

The predicate-window query model employs the following semantics.

**Stream semantics:** A stream,  $S$ , is modelled as a relation,  $\mathfrak{R}(S)$ , with the arriving stream tuples as *modifications* (insert, update, or delete) over  $\mathfrak{R}(S)$ . The timestamp attribute is treated as any other attribute in the tuple. This model is general and can represent any stream semantics. Assume, for example, a stream  $S_i$  is defined as a bag of elements following a specified schema [3]. In our framework,  $S_i$  is modelled as a relation,  $\mathfrak{R}(S_i)$ , with the same schema as  $S_i$ . When a tuple is added to the bag of  $S_i$ , the tuple is modelled as an insert operation over  $\mathfrak{R}(S_i)$ . A query over a stream,  $S$ , references the relational view,  $\mathfrak{R}(S)$ , of  $S$ . According to the underlying streaming application,  $\mathfrak{R}(S)$  can be modified either by only inserting new tuples or by general modify tuples (i.e., insert, update, and delete). Any modification in  $\mathfrak{R}(S)$  is propagated to produce the corresponding set of modifications in the query answer in a way similar to incremental maintenance of materialized views [13].

**Query semantics:** A predicate-window query is defined by an SQL expression over relations. Basically, the input relations are time-varying relations that represent relational views of input streams (i.e.,  $\mathfrak{R}(S)$ ). At any point in time,  $T$ , the query answer reflects the contents of the underlying relations at time  $T$ . The query answer is updated whenever any of the underlying relations is modified (e.g., by the arrival of new stream tuples) or the query definition is updated.

**Example:** As an example, a time-based sliding window over stream  $S$  can be expressed by the following predicate-

window query,  $Q_2$ :

```
SELECT *
FROM  $\mathcal{R}(S)$  R
WHERE  $Now - 5 < R.TS < Now$ 
```

Where  $TS$  is the tuple timestamp,  $5$  is the window size and  $Now$  is the time at which the query is issued. The answer of  $Q_2$  is continuously changing to include the newly arrived tuples that qualify the window predicate and delete the tuples that disqualify the predicate. In the previous query,  $Q_2$ , the input relation,  $\mathcal{R}(S)$ , is modified by the arrival of  $S$ 's tuples. A new  $S$ 's tuple is inserted in the query answer if it qualifies the predicate " $Now - 5 < R.TS < Now$ ". In the same time, the query definition is updated by changing the selection predicate boundaries (i.e.,  $Now$ ). As the value of  $Now$  is continuously changing, delete operations are produced in the output for tuples that expire from the window. Notice that, although the input stream,  $S$ , represents an append-only relation, the output stream does not represent append-only relation since it includes both *insert* and *delete* operations.

**Stream-specific issues:** Although similar to traditional SQL views, we plan to consider the following stream-specific requirements in the complete predicate-window query model:

- **Refresh periods.** The query definition can express arbitrary refresh periods to achieve coarser refresh of the query answer (e.g., *report the modifications in the answer every two minutes*).
- **Timestamps.** Since the inputs and outputs are streams, special attention is paid to the timestamps. The input timestamps are mapped to an attribute in the stream's corresponding relation. In the same time, timestamps are attached to the output stream tuples so that the output is interpreted in the same way as the input stream(s). The output streams can, then, be used as input in another predicate-window query.

## 2.2. Predicate-window Algebra

Algebraic laws are very important as they form the basis for efficient query transformation, query optimization, and shared query execution. In the existing continuous query languages, a window is either defined as a separate operator that is always placed at the bottom of the query pipeline [3, 18], or a window is defined inside an operator (e.g., window join in [7]). A language that is defined over one window semantics can not be easily used to express queries using the other window semantics. Moreover, there are no algebraic transformation rules in the existing continuous query literature to define the commutability and associativity of the window operators with other operators in the pipeline.

In our framework, we aim to develop a complete algebraic framework (i.e., algebraic laws and transformation rules) for the predicate-window query model. Defining the window as a regular predicate in the *where* clause, opens the room for optimizing window queries by following the rich SQL query optimization literature. Basically, the algebraic predicate-window framework is an extension over the relational algebra to capture time-varying relations. We define the algebraic framework in terms of:

- *New operators* that map streams to relations and vice versa. The relational view of a stream is a time-varying relation. Notice that, the mapping from a stream to a relation is done by simply applying the stream's tuples (i.e., insert, update or delete) to the relation in an increasing order of timestamps. On the other hand, mapping a relation to a stream is done by producing a tuple in the stream whenever the relation is modified by either inserting, updating, or deleting a tuple.
- *Extended relational operators* that capture time-varying relations. Inputs and outputs to the relational operators are time-varying relation. When an operator is applied to time-varying relation, the operator considers two parts: (1) Tuples or data in the relation. (2) Time points at which the relation is consistent with the underlying stream(s).
- *Algebraic laws and transformation rules* to define mappings and relationships among the new and extended operators. Transformation rules and equivalences over time-varying relations result in not only the same data, but also the same data at the same time points. The algebraic laws are used to test the equivalence and containment of time-varying relations.

Based on the algebra, we aim to develop cost models to be used by the query optimizers to enumerate the query plans and produce efficient execution plans. Cost models for predicate-window queries consider the optimization goals of the underlying streaming applications. Examples for optimization goals include, increasing the output rate of the query, minimizing response time for a tuple, and reducing the resource usage (e.g., memory and CPU).

## 2.3. Query Execution Plans

We develop an incremental and pipelined execution model to efficiently evaluate predicate-window queries. Although the predicate-window semantics considers only the relational view of an input stream, execution plans consider the physical stream characteristics (i.e., the sequence of modify operations). Basically, the predicate-window query pipelines are implemented using the negative tuples approach [18] where streams of positive and negative tuples are flowing in the query pipeline. The physical operators in

the pipeline are differential operators where the various operators' implementations employ the differential equations that are used in the incremental maintenance of materialized views [13].

If the characteristics of the input stream and the query is known in advance, the query pipeline can be tuned in order to optimize the resource usage (e.g., memory and CPU). For example, sliding windows can be considered as a special class of predicate windows in which tuples enter and expire from the window in a First-In-First-Out manner. In [18], we benefit from this known behavior of sliding windows and propose the *piggybacking* technique to reduce the CPU cost and pipeline bandwidth for sliding-window queries. The main idea in the piggybacking approach is that, since tuples enter and expire from the window in a FIFO manner, there is no need to process an explicit deletion tuple whenever a tuple expires from the window. Instead, the *piggybacking* technique avoids the processing of expired tuples whenever possible by depending on the timestamps of the inserted tuples to identify the expired tuples.

We aim at further extending our execution model by considering the runtime stream characteristics (e.g., out-of-order arrival, high rate). Examples for extensions include: (1) Extending the plan with stream specific operators (e.g., samplers and load shedders). (2) Using efficient in-memory indexing for the operators' states. (3) Implementing efficient caching algorithms to reduce memory usage by the query (e.g., adaptive caching [5]).

## 2.4. Shared Query Execution

The streaming environment is characterized by the large number of concurrent continuous queries, hence sharing the query execution is a primary task for query optimizers. The current efforts for shared query execution focus on sharing the execution at the operator level (e.g., shared aggregates [2], shared join [10, 14], shared predicate index [9, 10]).

In our framework, we aim to develop shared query execution algorithm that is based on query composition. By query composition we mean, using the output of one continuous query as input in another continuous query in a way similar to views in traditional databases. Query composition is enabled in the predicate-window query model because both the inputs and outputs of a query are interpreted in the same way (as streams of modify operations). Unlike existing efforts in shared execution, our algorithm is based on a whole query pipeline and not only on a single operator. Moreover, we consider, whenever possible, sharing the execution among queries that are similar in the query expression but different in the refresh periods.

Based on the algebraic model, sharing the execution can be achieved by common subexpression extractions. Com-

mon subexpressions should be common in both the window predicates and the refresh periods. Common subexpressions can be used in one of two ways as follows:

- **Query composition:** For example, given the expression of a continuous query  $Q_2$ , we look for another continuous query  $Q_1$  that matches a subexpression of  $Q_2$ . If such  $Q_1$  exists, then  $Q_2$  expression is re-written in terms of  $Q_1$  and hence the execution of  $Q_1$  is shared between the two queries. The problem of finding a matching query is similar to the “*view exploitation*” problem [12] in traditional databases.
- **Sub-query design:** For example, given the expressions of two or more continuous queries, we can use the common subexpressions to design a set of sub-queries that can be shared by the input queries such that the total execution cost of all the queries is minimized.

## 3. Related Work

**Continuous query semantics and languages:** Many research efforts have developed semantics and query languages for continuous queries over data streams, e.g., [3, 6, 7, 8, 11, 19]. Our semantics distinguishes itself from other languages by the following. (1) Modelling streams as a sequence of modify operations over a specified relation in contrast to a sequence of tuples that represents an append-only relation. This general stream modelling enables the semantics to express continuous queries that can not be expressed by the other window models. (2) Minimal extensions to SQL semantics. (3) General for all types of windows. Also, our framework is the first to address the following: (1) Providing an algebraic framework for window queries that explains the associativity and commutability of windows with the other operators in the pipeline. (2) Using query composition to achieve shared execution of continuous queries.

There are two SQL-based continuous query models that are most close to ours: CQL [4] and ATLaS [19]. Both languages restricts the stream definition to the representation of an append-only relation. Similar to our operators, CQL has three classes of operators: stream-to-relation, relation-to-relation, and relation-to-stream. However, our instantiation of operators in each class is different than that of CQL. Moreover, CQL is different from our predicate-window language in the following: (1) CQL can not express queries over streams of modify operations. (2) CQL assigns windows to streams and hence can not express queries where windows are assigned to operators (e.g., window join where two tuples are joined only if the two tuples are 5 time units apart). (3) CQL defines the window as a stream-to-relation operator but there are no transformation rules to

show how the window operator interacts with other operators in the pipeline. ATLaS is another SQL-based language for continuous query that is designed mainly for data mining and time-series queries. To guarantee that the output of the query represents an append-only relation, ATLaS restricts the set of queries that produce output streams to include append-only operators (e.g., selection and projections). On the other hand, since sliding-window queries produce non append-only output, ATLaS models the output of a sliding-window query as a concrete view. In the same time, only ad hoc queries are allowed on the concrete views [20]. Moreover, ATLaS focuses on aggregate queries and set-based queries are not discussed thoroughly in the published literature.

**Materialized views:** The physical operator implementation of predicate-window query pipelines follows the differential equations from the incremental maintenance of materialized views [13]. Moreover, our approach for shared query using query composition is based on query optimization using views as discussed in [12]. Basically, we extended the materialized view algorithms to work with streams instead of relations.

## 4. Conclusions

In this dissertation, we propose the predicate-window query model that empowers the data stream management system through an expressive query language and efficient querying mechanisms. Unlike the existing continuous query languages, the predicate-window framework defines the stream as a sequence of modify operations over a specified schema. This general stream definition allows predicate-windows to be used to express continuous queries over a wider variety of streaming applications. The predicate-window query semantics employs minimal extensions to the traditional SQL semantics hence opens the room for making use of the rich SQL algebra and optimization literature. We introduce a complete framework to efficiently support predicate-window queries in a data stream management system. The framework consists mainly of four components: Concise predicate-window semantics and syntax, predicate-window algebra, incremental pipelined execution plans, and shared query execution using query composition.

## References

- [1] <http://www.cs.purdue.edu/Nile/>.
- [2] A. Arasu and Jennifer Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*, 2004.
- [3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDBJ*, to appear.
- [4] Arvind Arasu and Jennifer Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record*, 33(3):6–12, 2004.
- [5] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Raveev Motwani. Adaptive Caching for Continuous Queries. In *ICDE*, 2005.
- [6] Philippe Bonnet, Johannes E. Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *MDM*, 2001.
- [7] Donald Carney. et al. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*, 2002.
- [8] Sirish Chandrasekaran. et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.
- [9] Sirish Chandrasekaran and Michael J. Franklin. PSoup: A System for Streaming Queries over Streaming Data. *VLDBJ*, 12(2):140–156, 2003.
- [10] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.
- [11] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.
- [12] Jonathan Goldstein and Per-Åke Larson. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *SIGMOD*, 2001.
- [13] Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD*, 1995.
- [14] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed E. Elmagarmid. Scheduling for Shared Window Joins over Data Streams. In *VLDB*, 2003.
- [15] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *SIGMOD*, 2005.
- [16] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of Data Streams and Operators. In *ICDT*, 2005.
- [17] Thanaa M.Ghanem, Walid G. Aref, and Ahmed K. Elmagarmid. Exploiting Predicate-window Semantics over Data Streams. *SIGMOD Record*, to appear.
- [18] Thanaa M.Ghanem, Moustafa A. Hammad, Mohamed F.Mokbel, Walid G. Aref, and Ahmed K. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. In *Purdue University Technical Report, CSD TR 04-040*, January 2006.
- [19] Haixun Wang, Carlo Zaniolo, and Chang R. Luo. ATLaS: a Small but Complete SQL Extension for Data Mining and Data Streams (demo). In *VLDB*, 2003.
- [20] WEB Information System Laboratory, UCLA, CS Department. An introduction to the Expressive Stream Language (ESL). <http://wis.cs.ucla.edu>.