Semantics and Implementation of Continuous Sliding Window Queries over Data Streams

JÜRGEN KRÄMER and BERNHARD SEEGER
University of Marburg

In recent years the processing of continuous queries over potentially infinite data streams has attracted a lot of research attention. We observed that the majority of work addresses individual stream operations and system-related issues rather than the development of a general-purpose basis for stream processing systems. Furthermore, example continuous queries are often formulated in some declarative query language without specifying the underlying semantics precisely enough. To overcome these deficiencies, this article presents a consistent and powerful operator algebra for data streams which ensures that continuous queries have well-defined, deterministic results. In analogy to traditional database systems, we distinguish between a logical and a physical operator algebra. While the logical algebra specifies the semantics of the individual operators in a descriptive but concrete way over temporal multisets, the physical algebra provides efficient implementations in the form of stream-to-stream operators. By adapting and enhancing research from temporal databases to meet the challenging requirements in streaming applications, we are able to carry over the conventional transformation rules from relational databases to stream processing. For this reason, our approach not only makes it possible to express continuous queries with a sound semantics, but also provides a solid foundation for query optimization, one of the major research topics in the stream community. Since this article seamlessly explains the steps from query formulation to query execution, it outlines the innovative features and operational functionality implemented in our state-of-the-art stream processing infrastructure.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; H.1.0 [**Models and Principles**]: General

General Terms: Algorithms

Additional Key Words and Phrases: Semantics, data streams, continuous queries, query optimization

ACM Reference Format:

Krämer, J. and Seeger, B. 2009. Semantics and implementation of continuous sliding window queries over data streams. ACM Trans. Database Syst. 34, 1, Article 4 (April 2009), 49 pages. DOI = 10.1145/1508857.1508861 http://doi.acm.org/10.1145/1508857.1508861

This work was supported by the German Research Foundation (DFG) under grant SE 553/4-3.

Authors' addresses: J. Krämer and B. Seeger, Department of Mathematics and Computer Science, University of Marburg, Germany; email: {kraemerj,seeger}@informatik.uni-marburg.de.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 0362-5915/2009/04-ART4 $5.00

DOI 10.1145/1508857.1508861 http://doi.acm.org/10.1145/1508857.1508861

1. INTRODUCTION

Continuous queries over unbounded data streams have emerged as an important query type in a variety of applications, for example, financial analysis, network and traffic monitoring, sensor networks, and complex event processing [Babcock et al. 2002; Golab and Özsu 2003; Abadi et al. 2003; Cranor et al. 2003; Wang et al. 2003; Demers et al. 2007]. Traditional database management systems are not designed to provide efficient support for the continuous queries posed in these data-intensive applications [Babcock et al. 2002]. For this reason, novel techniques and systems dedicated to the challenging requirements in stream processing have been developed. There has been a great deal of work on system-related topics such as adaptive resource management [Babu et al. 2005; Tatbul et al. 2003], scheduling [Babcock et al. 2003; Carney et al. 2003], query optimization [Viglas and Naughton 2002; Zhu et al. 2004], and on individual stream operations, such as user-defined aggregates [Wang et al. 2003], windowed stream joins [Kang et al. 2003; Golab and Öszu 2003a], and windowed aggregation [Yang and Widom 2001; Li et al. 2005]. However, an operative data stream management system needs to unify all this functionality. This constitutes a serious problem because most of the approaches rely on different semantics, which makes it hard to merge them. The problem gets even worse if the underlying semantics is not specified properly but only motivated through illustrative examples, for instance, in some informal, declarative query language. Moreover, such queries tend to be simple and the semantics of more complex queries often remains unclear.

To develop a complete Data Stream Management System (DSMS), it is crucial to identify and define a basic set of operators to formulate continuous queries. The resulting stream operator algebra must have a precisely defined and reasonable semantics so that at any point in time, the query result is clear and unambiguous. The algebra must be expressive enough to support a wide range of streaming applications, for example, the ones sketched in SQR [2003]. Therefore, the algebra should support windowing constructs and stream analogs of the relational operators. As DSMSs are designed to run thousands of continuous queries concurrently, a precise semantics is also required to enable subquery sharing in order to improve system scalability, that is, common subplans are shared and thus computed only once [Chen et al. 2000]. Furthermore, it has to be clarified whether query optimization is possible and to what extent. The relevance of this research topic is also confirmed in the survey Babcock et al. [2002]:

> [P]erhaps the most interesting open question is that of defining extensions of relational operators to handle stream constructs, and to study the resulting "stream algebra" and other properties of these extensions. Such a foundation is surely key to developing a general-purpose well-understood query processor for data streams.

Besides the semantic aspects, it is equally important for realizing a DSMS to have an efficient implementation that is consistent with the semantics. To date, little work has been published that combines semantic findings with execution details in a transparent manner. Therefore, this article specifies our

general-purpose stream algebra not only from the logical but also from the physical perspective. In addition, it reveals how to adapt the processing steps for one-time queries, common in conventional DBMSs, to continuous queries.

The notion of time plays an important role in the majority of streaming applications [Babcock et al. 2002; Golab and Özsu 2003; SQR 2003]. Usually, stream tuples are tagged with a timestamp. Any stream algebra should consequently incorporate operators that exploit the additionally available temporal information. Extending the relational algebra towards a temporal algebra over data streams is nontrivial for a number of reasons, including the following.

—*Timestamps.* It is not apparent how timestamps are assigned to the operator results. Assume we want to compute a join over two streams whose elements are integer-timestamp pairs. Is a join result tagged with the minimum or maximum timestamp of the qualifying elements, or both? What happens in the case of cascading joins? While some approaches prefer to choose a single timestamp, such as Babcock et al. [2003] and Ghanem et al. [2007], others suggest keeping all timestamps to preserve the full information [Golab and Öszu 2003a; Zhu et al. 2004]. To the best of our knowledge, currently there exists no consensus.

—*Resource Limitations.* It is not always possible to compute exact answers for continuous queries because streams may be unbounded, whereas system resources are limited [Babcock et al. 2002; Golab and Özsu 2003]. Since high-quality approximate answers are acceptable in lieu of exact answers in the majority of applications, we employ *sliding windows* as approximation technique. By imposing this technique on data streams, the range of a continuous queries is restricted to finite sliding windows capturing the most recent data from the streams, rather than the entire past history. For this reason, it is essential to analyze the use and impact of these windowing constructs on logical and physical query plans. Be aware that things can become complex quickly, especially if continuous queries include subqueries that are possibly windowed.

—*Language Extensions.* In DBMSs it is common to formulate queries via SQL rather than directly composing query plans at operator level. However, the novel constructs specific to stream processing necessitate extensions to SQL. An appropriate syntax with an intuitive meaning plus the language grammar modifications have to be identified. Eventually, queries should be compact and easy to write.

While all operators of our logical algebra can handle infinite streams, those operators of the physical algebra keeping state information such as the join and aggregation usually cannot produce exact answers for unbounded input streams with a finite amount of memory [Arasu et al. 2002]. According to our *continuous sliding window query semantics*, we are able to provide practical implementations to restrict resource usage. We prefer a query semantics that considers sliding windows over data streams for two reasons. First, the most recent data is emphasized, which is viewed to be more relevant than the older data in the majority of real-world applications. Second, the query answer is

exact with regard to the window specifications. Our approach differs in this quality from other approximation techniques maintaining compact *synopses* as summarizations of the entire operator state [Babcock et al. 2002; Golab and Özsu 2003]. Unfortunately, it is often not possible to give reliable guarantees for the results of approximate queries, in particular for sampling-based methods, which prevents query optimization [Chaudhuri et al. 1999]. Approaches based on synopses are altogether complementary to the work presented here.

As only a few papers published so far address the design and implementation issues of a general-purpose algebra for continuous query processing, we focus on continuous sliding window queries, their semantics, implementation, and optimization potential in this work. We next summarize our contributions.

- We define a concrete logical operator algebra for data streams with precise query semantics (Section 4). The semantics are derived from the well-known semantics of the extended relational algebra [Dayal et al. 1982; Garcia-Molina et al. 2000] and its temporal enhancement [Slivinskas et al. 2001]. To the best of our knowledge, such a semantic representation over multisets does not exist, neither in the stream community nor in the field of temporal databases.
- We discuss the basic implementation concepts and algorithms of our physical operator algebra (Section 7). Our novel stream-to-stream operators are designed for push-based, incremental data processing. Our approach is the first that uses time intervals to represent the validity of stream elements according to the window specifications in a query. We show how time intervals are initialized and adjusted by the individual operators to produce correct query results. Although it was not possible to apply research results from temporal databases to stream processing directly, our approach profits from the findings in Slivinskas et al. [2001].
- We illustrate query formulation (Section 3) and reveal plan generation (Section 5) in our stream processing infrastructure PIPES [Krämer and Seeger 2004]. Altogether, this article surveys our practical approach and makes the successive tasks from query formulation to query execution seamlessly transparent, an aspect missing in related papers. Moreover, we discuss the expressive power of our approach and compare it to the *Continuous Query Language (CQL)*.
- We adapted the temporal concept of snapshot-equivalence for the definition of equivalences at data stream and query plan level. These equivalences do not only validate the concordance between our physical and logical operator algebra, but also establish a solid basis for logical and physical query optimization, as we were able to carry over most transformation rules from conventional and temporal databases to stream processing (Sections 6 and 8).
- A discussion, along with experimental studies, confirms the superiority of our unique time-interval approach over the positive-negative approach [Arasu et al. 2006; Ghanem et al. 2007] for time-based sliding window queries, the predominant type of continuous queries (Appendix C).

2. PRELIMINARIES

This section defines a formal model of data streams. We first introduce the terms *time* and *tuple*, then we formalize our different types of streams, namely, *raw streams*, *logical streams*, and *physical streams*. After that, we define equivalence relations for streams and stream transformations. Section 2.7 briefly outlines the processes involved in optimizing and evaluating a continuous query. Eventually, we introduce our running example in Section 2.8.

2.1 Time

Let $\mathbb{T} = (T, \leq)$ be a discrete time domain with a total order $\leq$. A *time instant* is any value from T. We use $\mathbb{T}$ to model the notion of *application time*, not system time. Note that our semantics and the corresponding implementation only require any discrete, ordered domain. For the sake of simplicity, let T be the non-negative integers $\{0, 1, 2, 3, \dots\}$.

2.2 Tuples

We use the term *tuple* to describe the *data portion* of a stream element. In a mathematical sense, a tuple is a finite sequence of objects, each of a specified type. Let $\mathcal{T}$ be the composite type of a tuple. Let $\Omega_{\mathcal{T}}$ be the set of all possible tuples of type $\mathcal{T}$. This general definition allows a *tuple* to be a relational tuple, an event, a record of sensor data, etc. We do not want to restrict our stream algebra to the relational model, as done in Arasu et al. [2006], because our operations are parameterized with functions and predicates that can be tailored to arbitrary types. For the case of the relational model, the type would be a relational schema and the tuples would be relational tuples over this schema. As we want to show that our approach is expressive enough to cover CQL, we point out how our operations can be adapted to the relational model whenever necessary.

2.3 Stream Representations

Throughout the article, we use different representations for data streams. *Raw streams* denote input streams registered at the system. Hence, these streams provide the system with data. In our internal representation, we use different stream formats. We distinguish between a logical and a physical algebra in analogy to traditional DBMSs. The logical operator algebra relies on *logical streams*, whereas its physical counterpart is based on *physical streams*. Figure 1 gives an overview of the different stream representations and the formats of stream elements. Figure 2 shows the conceptual schema of our running example drawn from the NEXMark benchmark.

2.3.1 *Raw Streams.* *Raw streams* constitute external streams provided to our DSMS. In the majority of real-world streams, tuples are tagged with a timestamp [Golab and Özsu 2003; Arasu et al. 2006; Abadi et al. 2003; SQR 2003].

Definition 2.1 (*Raw Stream*). A *raw stream* S^r of type $\mathcal{T}$ is a potentially infinite *sequence of elements* (e, t), where $e \in \Omega_{\mathcal{T}}$ is a *tuple* of type $\mathcal{T}$ and $t \in$

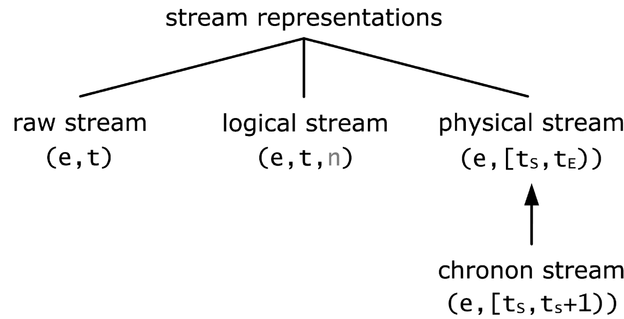

Fig. 1. Overview of stream representations.

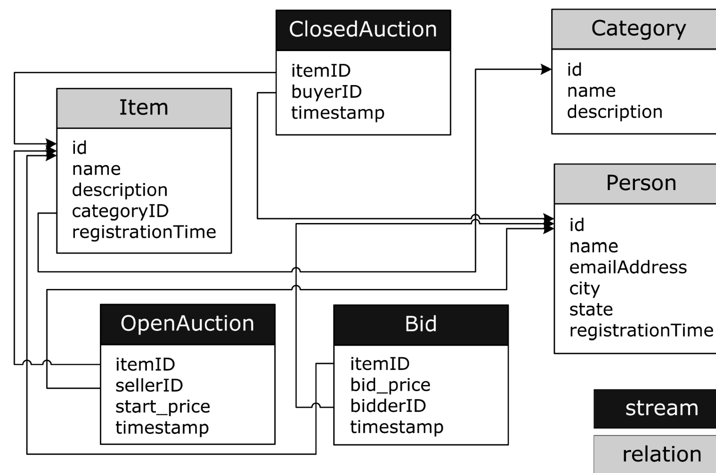

Fig. 2. Conceptual schema of NEXMark data.

T is the associated *timestamp*. A raw stream is nondecreasingly ordered by timestamps. In the following, let $\mathbb{S}^r_{\mathcal{T}}$ be the set of all raw streams of type $\mathcal{T}$.

A stream element (e, t) can be viewed as an event, that is, as an instantaneous fact capturing information that occurred at time instant t [Jensen et al. 1994]. A measurement obtained from a sensor is an example for such an event. Note that the timestamp is not part of the tuple, and hence does not contribute to the stream type. There can be zero or multiple tuples with the same timestamp in a raw stream. We only require the number of tuples having the same timestamp to be finite. This weak condition generally holds for real-world data streams. For cases in which raw streams do not satisfy our ordering requirement, that is, some elements may arrive out of order by timestamp, we also provide techniques to cope with bounded disorder. See Section 7.2.3 and Krämer [2007] for further details.

—*Base Streams and Derived Streams.* Raw streams can be converted to logical and physical streams using the transformations given in Section 2.5. We use the terms *base streams or source streams* for the resultant streams

because these streams represent the actual inputs of the logical and physical plans, respectively. In contrast, we term the output streams produced by logical and physical operators *derived streams*.

—*System Timestamps.* If the tuples arriving at the system are not equipped with a timestamp, the system assigns a timestamp to each tuple, by default using the system's local clock. While this process generates a raw stream with system timestamps that can be processed like a regular raw stream with application timestamps, the user should be aware that application time and system time are not necessarily synchronized.

2.3.2 *Logical Streams.* A *logical stream* is the order-agnostic multiset representation of a raw or physical stream used in our logical algebra. It shows the validity of tuples at time-instant level. Logical and physical streams are sensitive to duplicates, that is, tuples valid at the same point in time, because: (i) raw streams may already deliver duplicates, and (ii) some operators like projection or join may produce duplicates during runtime, even if all elements in the input streams are unique.

Definition 2.2 (*Logical Stream*). A *logical stream* S^l of type $\mathcal{T}$ is a potentially infinite *multiset (bag)* of elements (e, t, n), where $e \in \Omega_{\mathcal{T}}$ is a *tuple* of type $\mathcal{T}$, $t \in T$ is the associated *timestamp*, and $n \in \mathbb{N}$, $n > 0$, denotes the *multiplicity* of the tuple. Let $\mathbb{S}^l_{\mathcal{T}}$ be the set of all logical streams of type $\mathcal{T}$.

A stream element has the following meaning: Tuple e is valid at time instant t and occurs n times at t. A logical stream S^l satisfies the following condition.

$$\forall (e, t, n), (\hat{e}, \hat{t}, \hat{n}) \in S^l. \, (e = \hat{e} \wedge t = \hat{t}) \Rightarrow (n = \hat{n})$$

The condition prevents a logical stream from containing multiple elements with identical tuple and timestamp components.

—*Relational Multisets.* We enhanced the well-known multiset notation for the extended relational algebra proposed by Dayal et al. [1982] for logical streams by incorporating the notion of time. We use this novel representation to convey our query semantics in an elegant yet concrete way. The snapshot perspective facilitates the understanding of the temporal properties of our stream operators.

—*Abstraction.* Many of the physical aspects like infinite stream sizes, blocking operators, out-of order processing, and temporal stream synchronization that are indispensable for the implementation of physical stream operators can be ignored at the logical level. Similar to the logical view of operators in relational DBMS, such an abstraction is valuable for: (i) presenting the meaning of a continuous query in a clear and precise fashion and (ii) discovering and reasoning about equivalences as a rationale for query optimization, without dealing with specific implementation details.

2.3.3 *Physical Streams.* Operators in the physical algebra process so-called *physical streams*. Instead of sending positive and negative tuples through

a query pipeline as done in Arasu et al. [2006] and Ghanem et al. [2007], we propose a novel approach that assigns a time interval to every tuple representing its validity. To the best of our knowledge, we are the first in the stream community to pursue such an approach [Krämer and Seeger 2005]. Conceptually, a physical stream can be viewed as a more compact representation of its logical counterpart that: (i) coalesces identical tuples with consecutive timestamps into a single tuple with a time interval and (ii) brings the resulting stream elements into ascending order by start timestamps.

Definition 2.3 (*Physical Stream*). A *physical stream* S^p of type $\mathcal{T}$ is a potentially infinite, ordered *multiset of elements* $(e, [t_S, t_E))$, where $e \in \Omega_{\mathcal{T}}$ is a *tuple* of type $\mathcal{T}$, and $[t_S, t_E)$ is a half-open *time interval* with $t_S, t_E \in T$. A physical stream is nondecreasingly ordered by start timestamps. Let $\mathbb{S}^p_{\mathcal{T}}$ be the set of all physical streams of type $\mathcal{T}$.

The meaning is as follows: Tuple e is valid during the time period given by $[t_S, t_E)$, where t_S denotes the start and t_E the end timestamp. According to our definition, a physical stream may contain duplicate elements, and the ordering of elements is significant. Note that we do not enforce any order for stream elements with identical start timestamps.

Definition 2.4 (*Chronon Stream*). A *chronon stream* of type $\mathcal{T}$ is a specific physical stream of the same type whose time intervals are *chronons*, namely, nondecomposable time intervals of fixed, minimal duration determined by the time domain $\mathbb{T}$.[1]

Throughout this article, the terms *time instant* or simply *instant* denote single points in time, whereas the terms *chronon* and *time unit* denote the time period between consecutive points in time (at finest time granularity).

Example 1. Figure 3 gives an example of two physical streams that map to the same logical stream. The physical stream at the top is a chronon stream. The x-axis shows the time line. The letters indicate tuples. The numbers in the drawing of the logical stream denote the multiplicity of the corresponding tuples shown on the y-axis.

2.4 Value-Equivalence

Value-equivalence is a property that characterizes two stream elements independent of the stream representation. Informally, we denote two stream elements to be *value-equivalent* if their tuples are equal.

Definition 2.5 (*Value Equivalence*). Let $S^x_1 \in \mathbb{S}^x_{\mathcal{T}}, S^x_2 \in \mathbb{S}^x_{\mathcal{T}}$, where $x \in \{r, p, l\}$, be two streams whose tuples have a common supertype $\mathcal{T}$. Let s_1 and s_2 be two stream elements from S^x_1 and S^x_2, respectively, with tuples e_1 and e_2. We denote both elements to be *value-equivalent iff* $e_1 = e_2$.

We use the term *value-equivalence* instead of *tuple-equivalence* to be compatible with the consensus glossary of temporal database concepts [Jensen et al. 1994].

[1] see Jensen et al. [1994] for the definition of the term *chronon*.

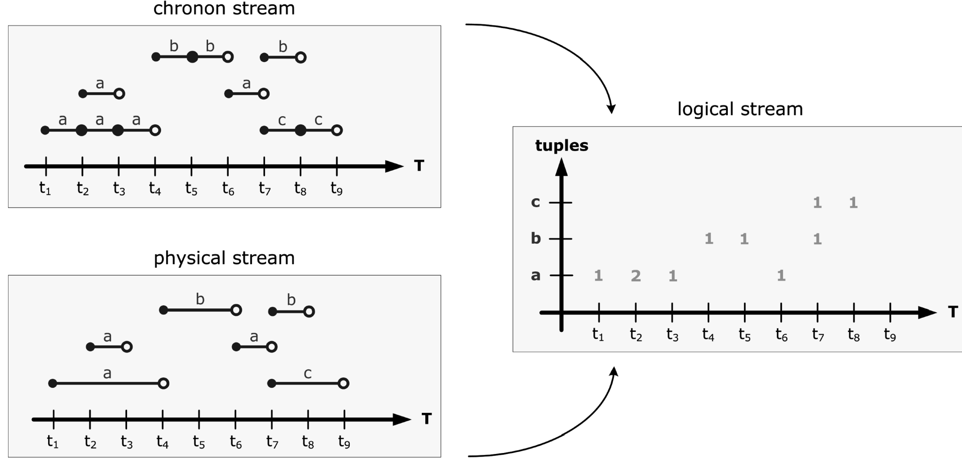

Fig. 3. An example for logical and physical streams.

2.5 Stream Transformations

To make streams available in either the logical or physical algebra, raw streams have to be converted into the respective stream model first. In addition to these input stream transformations, we specify the conversion from a physical stream into its logical analog. The latter transformation is required to prove the semantic equivalence of query answers returned by logical and physical plans.

2.5.1 *Raw Stream To Physical Stream.* Let S^r be a raw stream of type $\mathcal{T}$. The function $\varphi^{r \mapsto p} : \mathbb{S}^r_{\mathcal{T}} \to \mathbb{S}^p_{\mathcal{T}}$ takes a raw stream as argument and returns a physical stream of the same type. The conversion is achieved by mapping every stream element of the raw stream into a physical stream element as follows.

$$(e, t) \mapsto (e, [t, t+1))$$

The time instant t assigned to a tuple e is converted into a chronon. Therefore, the resultant physical stream is a *chronon* stream. Since a raw stream is ordered by timestamps, the resultant physical stream is properly ordered by start timestamps. The number of elements in a physical stream having identical start timestamps is finite because only a finite number of elements in a raw stream are allowed to have the same timestamp.

2.5.2 *Raw Stream to Logical Stream.* The function $\varphi^{r \mapsto l} : \mathbb{S}^r_{\mathcal{T}} \to \mathbb{S}^l_{\mathcal{T}}$ transforms a raw stream S^r into its logical counterpart.

$$\varphi^{r \mapsto l}(S^r) := \{(e, t, n) \in \Omega_{\mathcal{T}} \times T \times \mathbb{N} \mid n = |\{(e, t) \in S^r\}| \ \wedge \ n > 0\} \tag{1}$$

The logical stream subsumes value-equivalent elements of the raw stream. Hence, the multiplicity n corresponds to the number of duplicate tuples e occurring in S^r at time instant t.

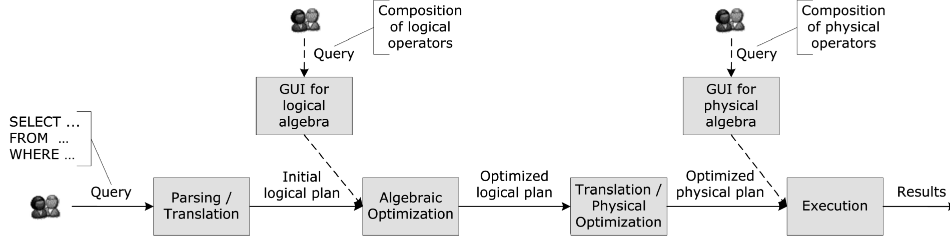

Fig. 4. Steps from query formulation to query execution.

2.6 Physical Stream To Logical Stream

The function $\varphi^{p \mapsto l} : \mathbb{S}^p_{\mathcal{T}} \rightarrow \mathbb{S}^l_{\mathcal{T}}$ transforms a physical stream S^p into its logical counterpart.

$$\varphi^{p \mapsto l}(S^p) := \{(e, t, n) \in \Omega_{\mathcal{T}} \times T \times \mathbb{N} \mid n = |\{(e, [t_S, t_E)) \in S^p \mid t \in [t_S, t_E)\}| \wedge n > 0\} \quad (2)$$

Recall that a logical stream describes the validity of a tuple e at time-instant level. For this reason, the time intervals assigned to tuples in the physical stream need to be split into single time instants. The multiplicity n of a tuple e at a time instant t corresponds to the number of elements in the physical stream whose tuple is equal to e and whose time interval intersects with t.

Remark 2.6. Often when there is no ambiguity we omit the superscript of a stream variable indicating the stream representation. For instance, throughout our logical and physical algebra, we label streams with S instead of S^l and S^p, respectively.

2.7 Query Processing Steps

The structure of this article follows the processes involved in optimizing and evaluating a query. Figure 4 shows this series of actions, which resembles the well-approved steps being applied in traditional database management systems. Nonetheless, notice the following differences. (1) Due to the long-running queries, the optimizer also takes subplans of running queries into account during plan enumeration to benefit from subquery sharing. (2) When the physical plan is registered at the execution engine, the execution engine integrates the plan into the system's global query graph, which consists of all operative query plans, and initiates data processing. (3) Alternatively, we permit the user to pose queries directly by constructing operator plans from either the logical or physical algebra via a graphical user interface.

2.8 Running Example

Examples throughout this work are drawn from the *NEXMark benchmark* for data stream systems developed in the Niagara project [Chen et al. 2000]. The benchmark provides schema and query specifications from an online auctions system such as eBay (see Figure 2). At any point in time, hundreds of auctions for individual items are open in such a system. The following actions occur

continuously over time: (i) New people register, (ii) new items are submitted for auction, and (iii) bids arrive for items.

We use the schema and continuous queries published at the Stream Query Repository [SQR 2003] in the following; full details can be found in the original specification of the benchmark [Tucker et al. 2002]. The schema consists of three relations (i.e., `Category`, `Item`, and `Person`) and three streams (`OpenAuction`, `ClosedAuction`, and `Bid`). Figure 2 shows the abbreviated conceptual schema of NEXMark. It illustrates the different types of tuples, their structure, and dependencies. The NEXMark benchmark generator provides the data in a data-centric XML format. In our implementation, we utilized a standard data binding framework for converting the tuples from XML format into objects. As this work deals with continuous queries over data streams, we refer the interested reader to Krämer [2007] to obtain information about how we incorporate relations into query processing. In our terminology, all streams in the NEXMark benchmark are raw streams since tuples are associated with timestamps. For example, the `Bid` stream provides tuples of the form $(itemID, bid_price, bidderID)$ tagged with a timestamp.

Remark 2.7. Some applications may already provide physical streams as inputs to the system, that is, they associate a tuple with a time period rather than a timestamp [Madden et al. 2002; Demers et al. 2007]. In this case, we omit the transformation from the raw to the physical stream and connect the input streams directly with the physical operator plans. The logical source streams are obtained from converting every physical input stream into a logical one. Even in the NEXMark scenario, it would be possible and meaningful to use a physical stream as input if a single auction stream is used instead of two separate streams indicating auction starts and endings (`OpenAuction` and `ClosedAuction`). As a result, the time interval associated to an auction would correspond to the auction duration. For compliance reasons, however, we stick firmly to the suggested conceptual schema of the NEXMark benchmark.

3. QUERY FORMULATION

Query formulation is a minor aspect in this article. Instead, we put emphasis on our query semantics and the efficient implementation of stream-to-stream operators. Our DSMS prototype called PIPES [Krämer and Seeger 2004] supports two different ways for formulating a continuous query: (i) via a *graphical interface* or (ii) via a *query language*. The graphical interface allows users to build query plans from our operator algebra manually, by connecting the individual operators in an appropriate manner. In this article, we want to present our declarative query language, which is a slight extension of standard SQL with a subset of the window constructs definable in SQL:2003. A thorough discussion of similarities and differences between CQL [Arasu et al. 2006] and our language is given in Krämer [2007]. Appendix D proves that our language is at least as expressive as CQL.

3.1 Registration of Source Streams

Recall that a source stream is obtained from mapping a raw stream into our internal stream representation. To register a source stream in the system, it is important to declare its name and schema first. This can be done with the CREATE STREAM clause (see Example 3.1). Registering a source stream adds an entry to the input stream catalog of the system. In analogy to the CREATE TABLE command in SQL, the name declaration is followed by the schema definition. The SOURCE clause is a new feature of our query language and consists of two parts. The first part is a user-defined function that establishes the connection to the actual underlying data source, for example, a raw stream obtained through a network connection. The second part indicates the attribute according to which the stream is ordered. It starts with the keywords ORDERED BY and expects a single attribute name.

Example 2. Let us consider the following statement for the registration of a raw stream.

```
CREATE STREAM OpenAuction(itemID INT, sellerID INT,
                          start_price REAL, timestamp TIMESTAMP)
SOURCE establishConnection('port34100', 'converter')
ORDERED BY timestamp;
```

The user-defined function *establishConnection* takes a port number and a converter as arguments. The converter transforms the sequence of bytes arriving at port 34100 into a raw stream.

Remark 3.1 (*Timestamp Attribute*). Be aware that, although the timestamp attribute is explicitly defined, it is not part of the stream schema. As a consequence, it cannot be referenced in queries. However, it is required in the stream definition to identify input streams with *explicit timestamps*. As some streams might provide multiple timestamp attributes, the ORDERED BY clause determines the attribute according to which the stream is ordered. If no ORDERED BY clause is specified, the system assigns an *implicit timestamp* upon arrival of an element using the system's internal clock.

3.2 Continuous Queries

The specification of a continuous query in our language resembles the formulation of one-time queries in native SQL using the common SELECT, FROM, WHERE, and GROUP BY clauses. While being as close to SQL as possible, our query language supports most of the powerful query specification functionality provided by standard SQL (SQL:1992), for instance, control over duplicates, nested subqueries, aggregates, quantifiers, and disjunctions [Dayal 1987].

3.2.1 *Window Specification in FROM clause.* As windowing constructs play an important role in stream processing, we slightly enhanced the FROM clause with a WINDOW specification. The BNF grammar for this specification is shown in Figure 5. To define a window over a stream, the window specification has to follow the stream reference in the FROM clause. A window specification

```
<from clause> ::= FROM  <stream reference> [<window specification>]
                  [{, <stream reference> [<window specification>]}...]

<window specification>    ::= WINDOW([<window partition clause>]
                              <window frame clause> [<slide clause>])
<window partition clause> ::= PARTITION BY <column list>
<window frame clause>     ::= <window frame units> <window frame start>
<window frame units>      ::= ROWS | RANGE
<window frame start>      ::= <value specification> | UNBOUNDED
<value specification>     ::= <positive integer> [<units>]
<slide clause>            ::= SLIDE <value specification>
```

Fig. 5. BNF grammar excerpt for extended FROM clause and window specification.

restricts the scope of a query to a finite, sliding subset of elements from the underlying stream. Rather than defining an entirely new syntax for windows from scratch, we reused the WINDOW specification of SQL:2003. Windows in SQL:2003 are restricted to built-in OLAP functions and thus are only permitted in the SELECT clause. Conversely, we aimed to let windows slide directly over any stream, base, and derived streams, for example, to express windowed stream joins. For this reason, we needed to extend the FROM clause. A detailed comparison with the window expression in SQL:2003 reveals two further changes. First, we solely consider preceding windows, as only these window types make sense in stream processing. The keyword PRECEDING then becomes superfluous and, therefore, we dropped it. Second, we added an optional SLIDE clause to specify the window advance, which, for instance, is required to express tumbling windows [Patroumpas and Sellis 2006].

3.2.2 *Default Window.* The BNF grammar for the extended FROM clause allows the user to omit the window specification. In this case, the default window is a time-based window of size 1, which covers all elements of the referenced input stream valid at the current time instant. We call this special type *now*-window. The following sections will show that now-windows can be omitted in the query formulation and also in query plans because evaluating now-windows produces the identity.

Example 3. Let us consider some examples of continuous queries formulated in our enriched SQL language.

Currency Conversion Query. Convert the prices of incoming bids from U.S. dollars into euros.

```
SELECT itemID, DolToEuro(bid_price), bidderID
FROM   Bid;
```

DolToEuro is a user-defined function to convert a dollar price to euros. The function is invoked on every incoming bid. The query does not specify a windowing construct, hence a now-window is applied by default.

Selection Query. Select all bids on a specified set of 5 items.

```
SELECT Bid.*
FROM   Bid
WHERE  itemID = 1007 OR itemID = 1020
       OR itemID = 2001 OR itemID = 2019
       OR itemID = 1087;
```

The filter condition is expressed in the `WHERE` clause listing the item identifiers of interest.

Short Auctions Query. Report all auctions that closed within 5 hours of their opening.

```
SELECT OpenAuction.*
FROM   OpenAuction O WINDOW(RANGE 5 HOURS),
       ClosedAuction C
WHERE  O.itemID = C.itemID;
```

This query is an example for a *windowed stream join*. It takes two streams as input, namely the `OpenAuction` and `ClosedAuction` stream. A time-based window of range 5 hours is defined over the `OpenAuction` stream, and a now-window is applied to the `ClosedAuction` stream by default. The join condition checks equality on item identifiers.

3.3 Registration of Derived Streams

The result of a continuous query is a data stream. Depending on the operator algebra used, this derived stream is either a logical or physical stream. Besides the functionality to register a source stream, it also possible to make derived streams available as inputs for other continuous queries. Like the `CREATE VIEW` mechanism in SQL, a derived stream can be defined by

```
CREATE STREAM <stream name> AS
              <query expression>.
```

Example 4. *Closing Price Query.* Report the closing price and seller of each auction.

```
CREATE STREAM CurrentPrice AS
  SELECT P.itemID, P.price, O.sellerID AS sellerID
  FROM ((SELECT itemID, bid_price AS price
         FROM Bid WINDOW(RANGE 2 DAYS))
         UNION ALL
        (SELECT itemID, start_price AS price
         FROM  OpenAuction WINDOW(RANGE 2 DAYS))) P,
         ClosedAuction C,
         OpenAuction O WINDOW(RANGE 2 DAYS)
  WHERE P.itemID = C.itemID AND C.itemID = O.itemID;

CREATE STREAM ClosingPriceStream AS
  SELECT   itemID, sellerID, MAX(P.price) AS price
```

```
FROM     CurrentPrice P,
GROUP BY P.itemID, P.sellerID;
```

The NEXMark benchmark generator sets the maximum auction duration to two days; hence, we use this timespan in the windowing constructs. Furthermore, it is assumed that the closing price in an auction is the price of the maximum bid, or the starting price of the auction in cases there were no bids [SQR 2003]. We choose this continuous query to demonstrate the following language features: the use of *derived streams*, *time-based sliding window joins*, and *windowed grouping with aggregation*. Furthermore, the query is relatively complex and offers optimization potential.

Remark 3.2. If we modeled the online auction application with a single auction stream using time intervals to express an auction's duration, the closing price would be much easier to compute (see also remark in Section 2.8). This approach makes sense since the auction duration is usually known in advance. In this case, it would be satisfactory to define a now-window on the `Bid` stream.

3.4 Nested Queries

It is obvious that our query language supports the formulation of nested queries because derived streams are allowed as inputs for continuous queries. The declaration and use of derived streams makes the language more appealing, as it is easier and more intuitive to express complex queries. However, the query result would be unchanged if all names of derived streams were replaced by the corresponding query specification.

Besides the simple type of subqueries standing for derived streams, our query language also supports the more complicated types of subqueries in SQL, like nested queries with aggregates or quantifiers [Dayal 1987].

Example 5. *Highest Bid Query.* Return the highest bid(s) in the last 10 minutes.

```
SELECT itemID, bid_price
FROM   Bid WINDOW(RANGE 10 MINUTES),
WHERE  bid_price = (SELECT MAX(bid_price)
                    FROM BID WINDOW(RANGE 10 MINUTES));
```

We choose this query as an example for a *subquery with aggregation*. The nested query in the `WHERE` clause determines the maximum bid price over a sliding window of ten minutes. The outer query applies the same window (identical `FROM` clause) and checks whether the price associated to a bid is equal to the maximum.

Hot Item Query. Select the item(s) with the most bids in the past hour.

```
SELECT itemID
FROM  (SELECT    B1.itemID AS itemID, COUNT(*) AS num
       FROM      Bid WINDOW(RANGE 60 MINUTES) B1
       GROUP BY  B1.itemID)
```

Table I. Examples of Logical Streams

(a) Stream S_1

T	Multiset
1	$\langle c \rangle$
2	$\langle a, a, a \rangle$
3	$\langle a, a, a, b \rangle$
4	$\langle a, a, a, b, c \rangle$
5	$\langle b, b \rangle$
6	$\langle b, b \rangle$

(b) Stream S_2

T	Multiset
1	$\langle \rangle$
2	$\langle b, b \rangle$
3	$\langle b, b \rangle$
4	$\langle a, b, c \rangle$
5	$\langle a, a, b \rangle$
6	$\langle a, c, c \rangle$

```
WHERE  num >= ALL(SELECT   COUNT(*)
                  FROM     Bid WINDOW(RANGE 60 MINUTES) B2
                  GROUP BY B2.itemID);
```

The example demonstrates a query with *universal quantification*. The nested query in the WHERE clause computes the number of bids received for each item over the last hour. The subquery in the FROM clause computes the same aggregate but assigns it to the corresponding item identifier. The outer query eventually checks whether the number of bids associated to an item identifier by the subquery in the FROM clause is equal to or greater than *all* values returned by the subquery in the WHERE clause.

4. LOGICAL OPERATOR ALGEBRA

This section presents some operators of our logical algebra. Due to space restrictions, we were not able to discuss the full operator set. For details, the interested reader is referred to Krämer [2007]. Section 4.1 discusses the *standard operators*, namely, those stream operators that have an analog in the extended relational algebra, whereas Section 4.2 defines the *window operators*. Rather than integrating windows into standard operators as done, for instance, in Kang et al. [2003] and Golab and Öszu [2003a], we separated the functionalities. This is an important step towards identifying a basic set of stream operators. It avoids the redundant specification of windowing constructs in the various operators and facilitates exchanging window types. In our case, the combination of window operators with standard operators creates their windowed analogs.

Example 6. The examples throughout this section illustrating the semantics of our operators are based on the two following logical streams.

$$S_1 := \{(c, 1, 1), (a, 2, 3), (a, 3, 3), (b, 3, 1), (a, 4, 3), (b, 4, 1), (c, 4, 1), (b, 5, 2), (b, 6, 2)\}$$

and

$$S_2 := \{(b, 2, 2), (b, 3, 2), (a, 4, 1), (b, 4, 1), (c, 4, 1), (a, 5, 2), (b, 5, 1), (a, 6, 1), (c, 6, 2)\}$$

Table I shows the two logical streams from the snapshot (time instant) perspective in tabular form. A table *row* indicates that the multiset of tuples listed in the second column is valid at the time instant specified in the first column. For ease of presentation, we denote a multiset by listing the tuples including duplicates in $\langle\rangle$ brackets, instead of using a set notation with tuple-multiplicity pairs.

Table II. Cartesian Product of Logical Streams S_1 and S_2

$S_1 \times S_2$

T	Multiset
1	$\langle\rangle$
2	$\langle a \circ b, a \circ b, a \circ b, a \circ b, a \circ b, a \circ b\rangle$
3	$\langle a \circ b, a \circ b, a \circ b, a \circ b,$ $a \circ b, a \circ b, b \circ b, b \circ b\rangle$
4	$\langle a \circ a, a \circ b, a \circ c, a \circ a, a \circ b,$ $a \circ c, a \circ a, a \circ b, a \circ c, b \circ a,$ $b \circ b, b \circ c, c \circ a, c \circ b, c \circ c\rangle$
5	$\langle b \circ a, b \circ a, b \circ b, b \circ a, b \circ a, b \circ b\rangle$
6	$\langle b \circ a, b \circ c, b \circ c, b \circ a, b \circ c, b \circ c\rangle$

4.1 Standard Operators

As examples for standard operators, we choose the Cartesian product ($\times$) and the scalar aggregation (α). See Krämer [2007] for further operators, including derived ones such as the join or grouping with aggregation.

4.1.1 *Cartesian Product.* The *Cartesian product* $\times : \mathbb{S}^l_{\mathcal{T}_1} \times \mathbb{S}^l_{\mathcal{T}_2} \rightarrow \mathbb{S}^l_{\mathcal{T}_3}$ of two logical streams combines elements of both input streams whose tuples are valid at the same time instant. Let $\mathcal{T}_3$ denote the output type. The auxiliary function $\circ : \Omega_{\mathcal{T}_1} \times \Omega_{\mathcal{T}_2} \rightarrow \Omega_{\mathcal{T}_3}$ creates an output tuple by concatenating the contributing tuples, denoted in infix notation. The product of their multiplicities determines the multiplicity of the output tuple.

$$\times(S_1, S_2) := \{(e_1 \circ e_2, t, n_1 \cdot n_2) \mid (e_1, t, n_1) \in S_1 \ \wedge \ (e_2, t, n_2) \in S_2\} \tag{3}$$

Example 7. Table II demonstrates the output stream of the Cartesian product over input streams S_1 and S_2 which, in turn, is the Cartesian product on the corresponding multisets at each time instant.

4.1.2 *Scalar Aggregation.* Let $\mathbb{F}_{agg}$ be the set of all aggregate functions over type $\mathcal{T}_1$. An *aggregate function* $f_{agg} \in \mathbb{F}_{agg}$ with $f_{agg} : \wp(\Omega_{\mathcal{T}_1} \times \mathbb{N}) \rightarrow \Omega_{\mathcal{T}_2}$ computes an *aggregate* of type $\mathcal{T}_2$ from a set of elements of the form (*tuple*, *multiplicity*). The aggregate function is specified as subscript. $\wp$ denotes the power set. The *aggregation* $\alpha : \mathbb{S}^l_{\mathcal{T}_1} \times \mathbb{F}_{agg} \rightarrow \mathbb{S}^l_{\mathcal{T}_2}$ evaluates the given aggregate function for every time instant on the nontemporal multiset of all tuples from the input stream being valid at this instant.

$$\begin{aligned} \alpha_{f_{agg}}(S) := \{(agg, t, 1) \mid \exists X \subseteq S.\ X \neq \emptyset \ \wedge \\ X = \{(e, n) \mid (e, t, n) \in S\} \ \wedge \ agg = f_{agg}(X)\} \end{aligned} \tag{4}$$

The aggregation implicitly eliminates duplicates for every time instant as it computes an aggregate value for all tuples valid at the same time instant weighted by the corresponding multiplicities. Note that the aggregate function can be a higher-order function. As a result, it is also possible to evaluate multiple aggregate functions over the input stream in parallel. The output type $\mathcal{T}_2$ describes the aggregates returned by the aggregate function. An aggregate

Table III. Scalar Aggregation

$\alpha_{\text{SUM}}(S_1)$

T	Multiset
1	$\langle c \rangle$
2	$\langle a + a + a \rangle$
3	$\langle a + a + a + b \rangle$
4	$\langle a + a + a + b + c \rangle$
5	$\langle b + b \rangle$
6	$\langle b + b \rangle$

consists of: (i) the aggregate value(s), and (ii) grouping information. The latter is important if a grouping is performed prior to the aggregation. An aggregate function should retain the portion of the tuples relevant to identify their group. For the relational case, this portion would correspond to the grouping attributes. Recall that the scalar aggregation treats its input stream as a single group.

Example 8. Table III illustrates the scalar aggregation for the sum. The tuples in the output stream are actually aggregate values, here denoted as sums. The aggregate value belonging to a time instant t is the sum of all tuples from S_1 being valid at t.

4.2 Window Operators

In this article, we focus on our time-based sliding window operator because this type of windows is the most common in continuous queries. Nevertheless, we also provide count-based and partitioned windows for compliance with the window specifications in SQL:2003 and CQL (see Krämer [2007]).

4.2.1 *Time-Based Sliding Window.* The *time-based sliding window* ω^{time} : $\mathbb{S}^l_T \times T \rightarrow \mathbb{S}^l_T$ takes a logical stream S and the window size as arguments. The *window size* $w \in T$, $w > 0$, represents a period of (application) time and is expressed as subscript.[2] The operator shifts a time interval of size w time units over its input stream to define the output stream.

$$\begin{aligned}\omega^{\text{time}}_w(S) := \{(e, \hat{t}, \hat{n}) \mid \exists X \subseteq S.\ X \neq \emptyset \wedge \\ X = \{(e,t,n) \in S \mid \max\{\hat{t} - w + 1, 0\} \leq t \leq \hat{t}\} \wedge \hat{n} = \sum_{(e,t,n)\in X} n\}\end{aligned} \tag{5}$$

At a time instant $\hat{t}$, the output stream contains all tuples of S whose timestamp intersects with the time interval $[\max\{\hat{t} - w + 1, 0\}, \hat{t}]$. In other words, a tuple appears in the output stream at $\hat{t}$ if it occurred in the input stream within the last w time instants $\leq \hat{t}$.

There are two special cases of windows supported by CQL: the *now-window* and the *unbounded window*. The now-window captures only the current time

[2]Our notation $w \in T$ indicates the number of time units captured by the window. The window size is given by the duration of the time interval $[0, w)$.

Table IV. Time-Based Sliding Window $\omega_2^{\text{time}}(S_1)$

T	Multiset
1	$\langle c \rangle$
2	$\langle a, a, a, c \rangle$
3	$\langle a, a, a, a, a, a, b \rangle$
4	$\langle a, a, a, a, a, a, b, b, c \rangle$
5	$\langle a, a, a, b, b, b, c \rangle$
6	$\langle b, b, b, b \rangle$
7	$\langle b, b \rangle$

instant of the input stream, hence, $w = 1$. We cover the semantics of the unbounded window, $w = \infty$, by defining that $\forall \hat{t} \in T : \max\{\hat{t} - \infty, 0\} = 0$. This means that all elements of S with timestamps $\leq \hat{t}$ belong to the window.

Example 9. Table IV specifies the logical output stream of the time-based sliding window operator applied to stream S_1 with a window size of 2 time units. A tuple occurs in the output stream at time instant t if it occurs in the input stream at time instants t or $t - 1$ for $t - 1 \geq 0$.

5. LOGICAL PLAN GENERATION

While query formulation with a graphical interface directly produces a query plan, queries expressed in a query language need to be compiled into a logical query plan. Parsing and translating a query from its textual representation into a logical plan closely resembles query plan construction in conventional database systems. In comparison with native SQL, we only extended the `FROM` clause with window specifications. Therefore, we need to clarify how the plan generator positions the window operators inside the plan.

5.1 Window Placement

Whenever the parser identifies a window expression following a stream reference, the corresponding window operator is installed in the query plan downstream of the node standing for the stream reference. There are two cases.

(1) If the stream reference represents a *source stream*, then the plan contains a leaf node as proxy for this stream reference. In this case, the plan generator installs the corresponding window operator downstream of the leaf node.
(2) If the stream reference represents a *derived stream* (subquery), then the query plan has been built partially and consists of the sources and operators computing the derived stream. In this case, the plan generator installs the corresponding window operator as topmost operator of the partial plan.

The rest of the plan generation process is equal to that in conventional DBMSs, that is, the plan generator constructs the plan downstream of the window operators as if these were not present.

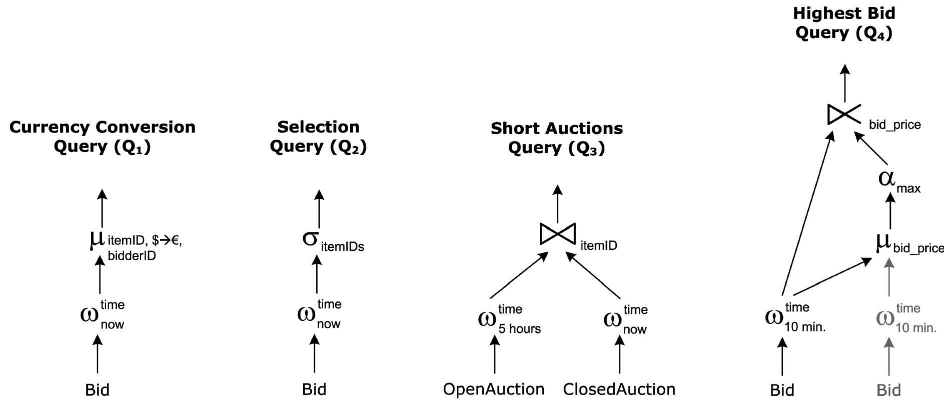

Fig. 6. Logical plans for NEXMark queries.

5.2 Default Windows

Window placement is only performed for windows explicitly specified in the query language statement. The user may have omitted the specification of default windows for the sake of simplicity. The corresponding now-windows do thus not occur in the logical query plan. Here, we have already applied an optimization, namely that now-windows do not have any impact on a logical stream (see also Section 6.3.2). Plan generation can hence ignore any now-windows that are explicitly defined in the query statement. Omitting default windows in logical plans eventually saves computational resources because the corresponding operators also disappear in the physical plans generated afterwards.

5.3 Examples

After the definition of our logical operators and the explanation of window operator placement, we want to show the logical plans for the example queries given in Section 3 (see Figures 6 and 7). The logical plans display the now-windows for the sake of better comprehension.[3]

The plans for the currency conversion (Q_1) and selection query (Q_2) are straightforward. A now-window is simply placed upstream of the map (μ) and filter (σ), respectively. The short auctions query (Q_3) uses time-based windows of different sizes. The window defined over the `OpenAuction` stream has a size of 5 hours, whereas the one over the `ClosedAuction` stream is a now-window. The subscript of the join symbol indicates that the join is defined on item identifiers. The projection to the attributes of the `OpenAuction` stream is performed by the function creating the join results. We omitted an extra symbol for that function. The plan for the highest bid query (Q_4) is more complicated, since it contains a subquery. Originally, each query is compiled into an operator plan that is a tree of operators. Because of subquery sharing, the operator plan diverges to a directed, acyclic operator graph. Subplans falling away due to subquery sharing appear in a lighter tone in our figures. In query Q_4, the time-based window

[3]Note that other approaches like the positive-negative approach require the now-windows [Arasu et al. 2006; Ghanem et al. 2007].

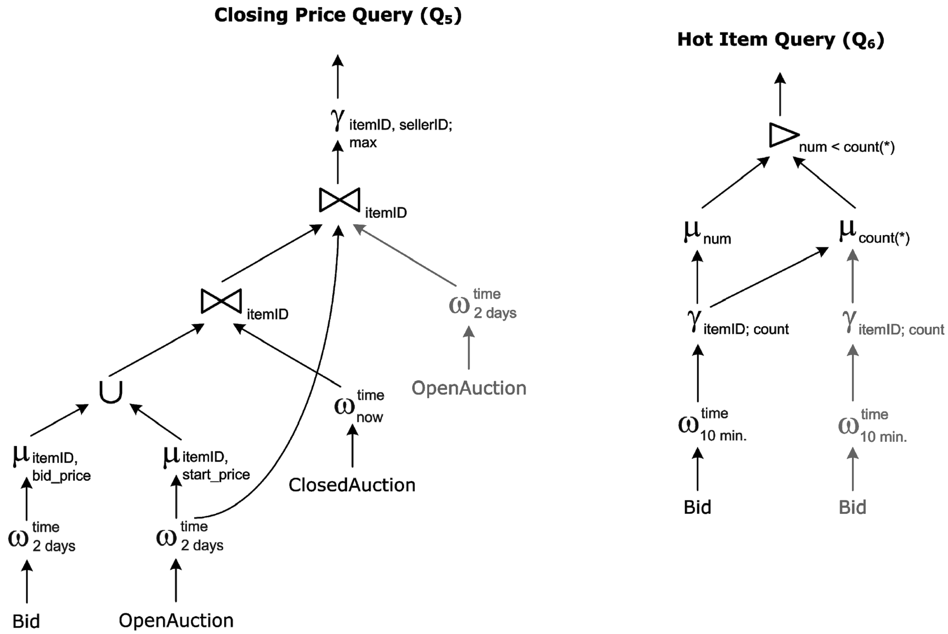

Fig. 7. Logical Plans for NEXMark queries (continued).

over the `Bid` stream can be shared since it is a commonality of the outer query and the nested query. The nested query performs a projection to the bid price attribute and computes the maximum afterwards. The outer query returns those elements in the window that match the maximum bid price computed by the nested query, that is, a semi-equijoin is placed as topmost operator. Hence, the subquery with aggregation is resolved using a semijoin ($\ltimes$) [Dayal 1987].

Figure 7 illustrates the logical plans for the more complex queries of the NEXMark scenario. In the closing price query (Q_5), we first compute the derived `CurrentPrice` stream. To obtain the current price of an auction, we merge the `Bid` and `OpenAuction` streams. Next, we compute an equijoin on item identifiers over the merged streams with the `ClosedAuction` stream. Since the time-based window of size two days captures all bids relevant for an item, we obtain a join result for each bid plus a join result for the initiation of the auction. The following equijoin with the `OpenAuction` stream associates the `sellerID` to each resultant tuple of the first join, which is a pair of item identifier and bid price. Recall that the seller identifier was projected out before the union to make the schema compatible. Finally, a grouping with aggregation (γ) applied to the `CurrentPrice` stream computes the maximum price for each item identifier, which corresponds to the closing price.

The logical plan for the hot item query (Q_6) resolves the nested query with universal quantification by means of an antijoin ($\triangleright$) [Dayal 1987]. First, the number of bids issued for the individual items within the recent ten minutes is computed by grouping on item identifiers combined with a count aggregate. The count attribute is renamed to *num* for the first input of the join. The `itemID`

attribute is projected out for the second input. The predicate of the antijoin verifies whether the *num* attribute of the first input is less than the count attribute of the second input. At every time instant, the antijoin outputs only those elements from the first input stream that do not satisfy the join predicate for any element in the second input stream. Consequently, at any time instant t, a result is only produced if a tuple exists in the first stream at t whose *num* value is equal to or greater than the *count* value of *all* tuples in the second stream at instant t.

6. ALGEBRAIC QUERY OPTIMIZATION

The foundation for any query optimization is a well-defined semantics. Our logical algebra not only precisely specifies the output of our stream operators but is also expressive enough to support state-of-the-art continuous queries. This section explains why our approach permits query optimization and clarifies to which extent. Section 6.1 introduces *snapshot-reducibility* [Jensen et al. 1994], an important property fulfilled by our standard operators. Section 6.2 defines equivalences for logical streams and operator plans. Section 6.3 reports on transformation rules holding in our logical algebra, while Section 6.4 discusses their applicability.

6.1 Snapshot-Reducibility

In order to define snapshot-reducibility, we first introduce the *timeslice operation* that allows us to extract snapshots from a logical stream.

Definition 6.1 (*Timeslice*). The *timeslice* operator $\tau : (\mathbb{S}^l_{\mathcal{T}} \times T) \rightarrow \wp(\Omega_{\mathcal{T}} \times \mathbb{N})$ takes a logical stream S^l and a time instant t as arguments, where t is expressed as subscript. It computes the snapshot of S^l at t, that is, the nontemporal multiset of all tuples being valid at time instant t.

$$\tau_t(S^l) := \{(e, n) \in \Omega_{\mathcal{T}} \times \mathbb{N} \mid (e, t, n) \in S^l\} \tag{6}$$

For the relational case, a snapshot can be considered as an *instantaneous relation* since it represents the bag of all tuples valid at a certain time instant (compare to Arasu et al. [2006]). Hence, the timeslice operator is a tool for obtaining an instantaneous relation from a logical stream.

Definition 6.2 (*Snapshot-Reducibility*). We denote a stream-to-stream operator op_S with inputs $S_1, \ldots, S_n$ as *snapshot-reducible iff* for any time instant $t \in T$, the snapshot at t of the results of op_S is equal to the results of applying its relational counterpart op_R to the snapshots of $S_1, \ldots, S_n$ at time instant t.

Figure 8 shows a commuting diagram that illustrates the previous definition. Snapshot-reducibility is a well-known concept in the temporal database community [Slivinskas et al. 2001; Jensen et al. 1994]. It guarantees that the semantics of a relational, nontemporal operator is preserved in its more complex, temporal counterpart. As time is an essential concept in stream processing as well, we decided to adopt the snapshot-reducibility property for our stream operators to the extent possible, with the aim to profit from the great deal of work done in temporal databases.

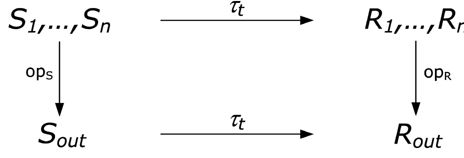

Fig. 8. Snapshot-reducibility.

LEMMA 6.3. *Let the tuples be relational tuples and operator functions and predicates adhere to the semantics of the extended relational algebra. It follows that the standard operators of our stream algebra are snapshot-reducible.*

PROOF. There exists a nontemporal analog in the extended relational algebra for every standard operator in our logical algebra. Moreover, we defined the semantics of standard operators such that it satisfies the snapshot-reducibility property. □

For example, the duplicate elimination over logical streams is snapshot-reducible to the duplicate elimination over relations. However, the class of standard operators represents only a subset of our logical algebra. This class considered alone would not be expressive enough for our purpose because windowing constructs are missing. Our window operators are not snapshot-reducible.

6.2 Equivalences

Let us first derive an equivalence relation for logical streams, and thereafter for logical query plans.

Definition 6.4 (*Logical Stream Equivalence*). We define two logical streams $S_1^l, S_2^l \in \mathbb{S}_T^l$ to be *equivalent iff* their snapshots are equal.

$$S_1^l \doteq S_2^l :\Leftrightarrow \forall t \in T.\ \tau_t(S_1^l) = \tau_t(S_2^l) \tag{7}$$

Definition 6.5 (*Logical Plan Equivalence*). Given two query plans Q_1 and Q_2 over the same set of logical input streams. Let each query plan represent a tree composed of operators from our logical operator algebra. Let S_1^l and S_2^l be the output streams of the plans, respectively. We denote both plans as *equivalent iff* their logical output streams are equivalent, that is, if for any time instant t the snapshots obtained from their output streams at t are equal.

$$Q_1 \doteq Q_2 :\Leftrightarrow S_1^l \doteq S_2^l \tag{8}$$

6.3 Transformation Rules

Due to defining our standard operations in semantic compliance with Slivinskas et al. [2001], the plethora of transformation rules listed in Slivinskas et al. [2001] carries over from temporal databases to stream processing. Here, semantic compliance means that our stream operators generate *snapshot-multiset-equivalent results* to the temporal operators presented in Slivinskas et al. [2001]. The transfer of this rich foundation for conventional and temporal query optimization enables us to algebraically optimize logical query plans over data streams.

6.3.1 *Conventional Rules.* Logical stream equivalence, which represents snapshot-equivalence over temporal multisets, causes the relational transformation rules to be applicable to our stream algebra.

Definition 6.6 (*Snapshot-Reducible Plans*). We define a *snapshot-reducible plan* as a query plan that involves exclusively snapshot-reducible operators.

LEMMA 6.7. *Given a snapshot-reducible plan, any optimizations with conventional transformation rules lead to an equivalent plan.*

PROOF. Let S_1^l and S_2^l be the output streams of the original and optimized plan, respectively. We have to show that for any time instant t, $\tau_t(S_1^l) = \tau_t(S_2^l)$. This condition follows directly from the snapshot-reducibility property of all operators involved and the correctness of the conventional transformation rules for the relational algebra. □

Lemma 6.7 states that all conventional transformation rules apply equally to plans built with standard operations of our stream algebra. This includes, for example, the well-known rules for join reordering, predicate pushdown, and subquery flattening [Garcia-Molina et al. 2000; Dayal 1987]. Transformed into our notation, examples for transformation rules are

$$(S_1^l \times S_2^l) \times S_3^l \doteq S_1^l \times (S_2^l \times S_3^l). \tag{9}$$

$$\sigma_p(S_1^l \cup S_2^l) \doteq \sigma_p(S_1^l) \cup \sigma_p(S_2^l). \tag{10}$$

$$\delta(S_1^l \times S_2^l) \doteq \delta(S_1^l) \times \delta(S_2^l). \tag{11}$$

The interested reader is referred to Slivinskas et al. [2001] for a complete list of applicable transformation rules.

6.3.2 *Window Rules.* We can add some novel rules for window operators to the previous transformation rules. Our first rule says that default windows can be omitted. Recall that a default window is a now-window, namely, a time-based window of size 1.

$$\omega_1^{\text{time}}(S^l) \doteq S^l \tag{12}$$

The correctness of this transformation rule directly derives from the definition of the time-based sliding window (see Section 4.2.1).

Furthermore, the time-based sliding window commutes with all stateless operators: filter, map, and union. For logical streams S^l, S_1^l, and S_2^l, we thus define the following transformation rules.

$$\sigma_p(\omega_w^{\text{time}}(S^l)) \doteq \omega_w^{\text{time}}(\sigma_p(S^l)) \tag{13}$$

$$\mu_f(\omega_w^{\text{time}}(S^l)) \doteq \omega_w^{\text{time}}(\mu_f(S^l)) \tag{14}$$

$$\cup(\omega_w^{\text{time}}(S_1^l), \omega_w^{\text{time}}(S_2^l)) \doteq \omega_w^{\text{time}}(\cup(S_1^l, S_2^l)) \tag{15}$$

We actually would have to prove the correctness of each individual rule, but the proofs are quite similar and straightforward. The reason for the commutativity is that filter, map, and union do not manipulate the validity of tuples. A look at the physical algebra might improve the understanding of this argumentation (see Section 7). The filter predicates and mapping functions are only invoked on

the tuple component of a stream element. From the perspective of the physical algebra, filter, map, and union satisfy the interval-preserving property formulated in Slivinskas et al. [2001].

A system can profit from the preceding window rules for the following reasons. If a queue is placed between the filter and the window operator, pushing the filter below the window (rule (13)) will reduce the queue size. In addition, this transformation saves processing costs for the window operator because the filtering is likely to decrease the input stream rate of the window operator. Altogether, the transformations can be useful for subquery sharing. For example, if a time-based window is applied after a union, and a new query is posed which could share the window functionality over one input stream, it is preferable to apply rule (15) and push the window operator down the union. See Krämer [2007] for transformation rules referring to count-based and partitioned windows.

6.4 Applicability of Transformation Rules

Assume an arbitrary logical query plan to be given in the form of an operator tree. We first divide the plan into *snapshot-reducible* and *nonsnapshot-reducible* subplans. The use of these properties enables us to decide where and which transformation rules can be applied properly.

6.4.1 *Snapshot-Reducible Subplans.* Conventional transformation rules can only be applied to snapshot-reducible subplans. Hence, these have to be identified first. This can be achieved by a bottom-up traversal. A new subplan starts: (i) at the first snapshot-reducible operator downstream of a source or (ii) at the first snapshot-reducible operator downstream of a nonsnapshot-reducible operator, for example, a window operator. As long as snapshot-reducible operators are passed, these are added to the current subplan. Maximizing the subplans increases the number of transformation rules applicable.

A query optimizer should consider the property of whether a subplan is snapshot-reducible or not in its plan enumeration process to apply the transformation rules correctly. Such a property can be handled similarly to the property whether duplicates are relevant or not, applied in Slivinskas et al. [2001] for optimization purposes.

Example 10. Figure 9 shows a possible algebraic optimization for the closing price query. The grey highlighted areas denote snapshot-reducible subplans. In this example, there exists only a single snapshot-reducible subplan that comprises all operators downstream of the window operators. We choose an optimization that is not obvious, namely, we push the grouping with aggregation below the join. This is possible because the attribute `itemID` is used for joining and grouping. The secondary grouping attribute `sellerID` specified for the grouping in the left plan just retains the seller information but does not affect the query answer in terms of additional results. The reason is an invariant of the auction scenario, namely, that every item can be sold only once.

Join reordering would be a further transformation rule applicable to the left plan. In this case, the output of the union would be joined with the

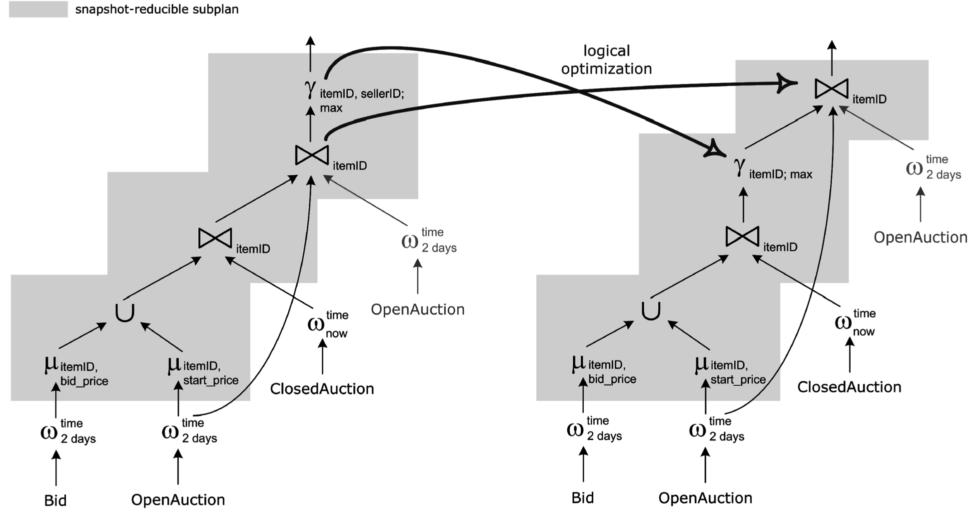

Fig. 9. Algebraic optimization of a logical query plan.

windowed `OpenAuction` stream first, followed by the join with the windowed `ClosedAuction` stream.

During plan enumeration the optimizer will have to decide based on a cost model whether one of the proposed optimizations actually leads to an improved query execution plan. Note that for all considered query plans of the NEXMark benchmark, the snapshot-reducible portion is defined by the subplan downstream of the window operators. Nonetheless, our approach is more general and allows plans to have multiple window operators on a path. When this occurs, a snapshot-reducible subplan ends at the last operator upstream of the window.

6.4.2 *Nonsnapshot-Reducible Subplans.* Nonsnapshot-reducible subplans consist of window operators and standard operators whose functionality is beyond that of their relational analogs. For instance, our map operator is more powerful than its relational counterpart as it permits arbitrary functions to be invoked on tuples. Furthermore, it is possible to enhance our algebra with nonsnapshot-reducible operators like temporal joins [Gao et al. 2005] to increase expressiveness. The downside of this approach is that query optimization becomes more difficult. Apart from our window transformation rules, novel transformation rules have to be identified. The optimizer should consider these in addition to the conventional transformation rules during plan enumeration. For our window transformation rules, the optimizer has to check each window operator along with adjacent operators in the given query plan.

Example 11. By applying window transformation rule (14), it would be possible to push the map operators down the window operators for the plans shown in Figure 9. A further optimization would be to pull the window operator up the union because the window is identical for both inputs (rule (15)). However, these optimizations would conflict with the subquery sharing performed on the window over the `OpenAuction` stream.

6.4.3 *Extensions to Multiple Queries.* Due to the long-running queries and the large number of queries executed concurrently by a DSMS, the stand-alone optimization of individual query plans is not appropriate. The optimizer should rather take other running queries into account to identify common subexpressions. To profit from subquery sharing, DSMSs unify all query plans into a global, directed, acyclic query graph. Therefore, one objective during plan enumeration is whether transformation rules can be applied to enable sharing of common subqueries. DSMSs should consequently exploit multiquery optimization techniques [Sellis 1988; Roy et al. 2000], but this is not enough. As queries are long-running, their efficiency may degrade over time because of changing stream characteristics such as value distributions and stream rates. Overcoming the resultant inefficiencies necessitates reoptimizations of continuous queries at runtime, called *dynamic query optimization* [Zhu et al. 2004; Yang et al. 2007], in addition to the static optimization prior to execution.

7. PHYSICAL OPERATOR ALGEBRA

This section presents our physical operator algebra. Section 7.1 shows the motivation behind our time-interval approach, while Section 7.2 characterizes important properties of our physical operators. Our data structure for state maintenance is explained in Section 7.3. Prior to the presentation of the standard operators in Section 7.5, Section 7.4 spends some words on the notation used in our algorithms. Section 7.6 reveals our implementation for the window operators.

7.1 Basic Idea

From the implementation point of view, it is not satisfactory to process logical streams directly because evaluating the operator output separately for every time instant would cause a tremendous computational overhead. Hence, we developed operators that process physical streams. Recall that a physical stream is a more compact representation of its logical counterpart that describes the validity of tuples by time intervals. Any physical stream can be transformed into a logical stream with the function $\varphi^{p \mapsto l}$ defined in Section 2.6. The transformation splits the time intervals into chronons and summarizes the multiplicity of value-equivalent tuples being valid at identical chronons. This semantic equivalence is the sole requirement between a logical stream and its physical counterpart for algebraic query optimization. The inverse transformation, namely from a logical stream to a semantically equivalent physical stream, is not needed in practice because query execution does not make use of the logical algebra. Moreover, multiple physical streams may represent the same logical stream. The reason is that multiple value-equivalent elements with consecutive timestamps from a logical stream can be merged into elements of a physical stream in different ways because coalescing timestamps to time intervals is nondeterministic. The example in Figure 3 already demonstrated that a logical stream can have several physical counterparts.

The basic idea of our physical algebra is to use novel window operators for adjusting the validity of tuples, namely, the time intervals, according to the

window specifications, along with appropriately defined standard operators, so that the results of a physical query plan are snapshot-equivalent to its logical counterpart. While stateless operators do not consider the associated time intervals and thus can cope with potentially infinite windows, time intervals affect stateful operators as follows. Elements are relevant to a stateful operator as long as their time interval may overlap with the time interval of any future stream element. This also means that a stateful operator can purge those elements from its state whose time interval cannot intersect with the time interval of any incoming stream element in the future. The latter explains how window specifications restrict the resource usage of stateful operators in our physical algebra.

To the best of our knowledge, our time-interval approach [Krämer and Seeger 2005, 2004] is unique in the stream community. Related work embracing similar semantics substantially differs from our approach (see Appendix C).

7.2 Operator Properties

Our physical algebra provides at least a single implementation for each operation of the logical algebra. A physical operator takes one or multiple physical streams as input and produces a physical stream as output. These physical stream-to-stream operators are implemented in a data-driven manner, assuming that stream elements are pushed through the query plan. Physical operators are connected directly when composing query plans. This implies that a physical operator has to process the incoming elements instantly on arrival without the ability to select the input from which the next element should be consumed. The push-based processing methodology is another important distinction between our implementation and related ones using pull-based techniques and interoperator queues for communication [Arasu et al. 2006; Abadi et al. 2003; Ghanem et al. 2007].

Remark 7.1. PIPES [Krämer and Seeger 2004] relies on a multithreaded query execution framework that permits the concurrent arrival of stream elements at an operator with multiple input streams. Since we do not want to discuss synchronization and scheduling issues here, we assume the processing of a single stream element to be atomic.

7.2.1 *Operator State.* The operators of our physical algebra can be classified into two categories.

- *Stateless operators* produce a result based on the evaluation of a single incoming element. As no other stream elements need to be considered, stateless operators do not need to maintain internal data structures keeping an extract of their input streams. Examples are filter, map, or time-based window.
- *Stateful operators* need to access an internal data structure to generate a result. The internal data structure maintaining the operator state has to hold all elements that may contribute to any future query results. Examples are Cartesian product/join, duplicate elimination, or scalar aggregation.

7.2.2 *Nonblocking Behavior.* According to Babcock et al. [2002], "[a] blocking operator is a query operator that is unable to produce the first tuple of the output until it has seen the entire input." Blocking operators are not suitable for stream processing due to the potentially unbounded input streams. Therefore, all stream-to-stream operators of our physical algebra have to be nonblocking. However, it is generally known that some relational operators are blocking, for example, the aggregation or difference. The reason why we can provide stream variants of these operators is that we exploit windowing and incremental evaluation in our algorithms [Golab and Özsu 2003]. If a query plan contains any stateful operators downstream of the window operators, we require the window sizes to be finite. As a result, the window operators restrict the scope of stateful operators to finite, sliding windows. As a consequence, the stateful operators downstream of the window operators do not block. With regard to time intervals, the following happens. Due to the finite window sizes, the length of time intervals in any physical stream downstream of the window operators is finite. As only a finite number of stream elements are allowed to have identical start timestamps (see Section 2), only a finite number of elements may overlap at an arbitrary time instant. Due to the fact that a stateful operator needs to keep only those elements in the state whose time interval may overlap with that of incoming elements in the future in order to generate the correct results, the total size of the operator state is limited. Any relational operator, except for sorting which is inherently blocking, can be unblocked by windowing techniques.

Stateless operators can even cope with unbounded streams, as they do not require to maintain a state. We thus allow the user to specify unbounded windows in those queries that do not contain stateful operations.

7.2.3 *Ordering Requirement.* Each physical operator assumes that every input stream is correctly ordered. Moreover, each physical operator has to ensure that its output stream is properly ordered, that is, elements are emitted in nondecreasing order by start timestamps (see Section 2.3.3). The ordering requirement is crucial to correctness because the decision which elements an operator can purge from its state depends on the progression of (application) time obtained from the start timestamps of the operator's input streams. Recall that for the majority of streaming applications, the elements provided by raw streams already arrive in ascending order by start timestamps [Babcock et al. 2002; Golab and Özsu 2003]. Consequently, the ordering requirement is implicitly satisfied for the physical base streams of a query plan (see Section 2.5.1). If raw streams may receive out-of-order elements, for example, due to network latencies, mechanisms like *heartbeats* and *slack* parameters can be applied in our approach as well for correcting stream order [Srivastava and Widom 2004; Abadi et al. 2003]. How PIPES deals with liveness and disorder is an orthogonal problem and, thus, not the scope of this article. For detailed information, see Krämer [2007].

7.2.4 *Temporal Expiration.* Windowing constructs: (i) restrict resource usage and (ii) unblock otherwise blocking operators over infinite streams. In

stateful operators, elements in the state expire due to the validity set by the window operators. A stateful operator considers an element $(e, [t_S, t_E))$ in its state as expired if it is guaranteed that no element in one of its input streams will arrive in the future whose time interval will overlap with $[t_S, t_E)$. According to the total order claimed for streams, this condition holds if the minimum of all start timestamps of the latest incoming element from each input stream is greater than t_E. A stateful operator can delete all expired elements from its state. We will show that some operators such as the aggregation emit these expired elements as results prior to their removal.

7.3 SweepAreas

Since our algorithms for stateful operators utilize specific data structures for state maintenance, we first want to introduce the *abstract data type SweepArea (SA)* and outline possible implementations. Furthermore, we utilize iterators [Graefe 1993] for traversing data structures.

7.3.1 *Abstract Data Type SweepArea.* The Abstract Data Type (ADT) *SweepArea* models a dynamic data structure to manage a collection of elements having the same type $\mathcal{T}$. Besides functionality to insert and replace elements and to determine the size, a SweepArea provides in particular generic methods for probing and eviction. Our development of this data structure was originally inspired by the sweepline paradigm [Nievergelt and Preparata 1982].

ADT SweepArea

SweepArea (*total order relation* $\leq$, *binary predicate* p_{query}, *binary predicate* p_{remove})
Procedure **insert**(*element s*)
Procedure **replace**(*element* $\hat{s}$, *element s*)
Iterator **iterator**()
Iterator **query**(*element s,* $j \in \{1, 2\}$)
Iterator **extractElements**(element s, $j \in \{1, 2\}$)
Procedure **purgeElements**(*element s,* $j \in \{1, 2\}$)
int **Size**()

The behavior of a SweepArea is controlled by three parameters passed to the constructor.

(1) The *total order relation* $\leq$ determines the order in which elements are returned by the methods *iterator* and *extractElements*.
(2) The *binary query predicate* p_{query} is used in the method *query* to check whether the elements in the SweepArea qualify.
(3) The *binary remove predicate* p_{remove} is used in the methods *extractElements* and *purgeElements* to identify elements that can be removed from the SweepArea.

—*Probing*. The function *query* probes the SweepArea with a *reference element s* given as parameter. It delivers an iterator over all elements $\hat{s}$ of the SweepArea that satisfy $p_{query}(s, \hat{s})$ for $j = 1$, or $p_{query}(\hat{s}, s)$ for $j = 2$. Parameter

j determines whether s becomes the first or second argument of the query predicate. This argument swapping is required to cope with: (i) asymmetric query predicates and (ii) query predicates over different argument types.

In contrast to the function *query*, where the order in which qualifying elements are delivered depends on the actual implementation of the SweepArea, the iterator returned by a call to the function *iterator* delivers all elements of the SweepArea in ascending order by $\leq$.

—*Eviction.* The methods *extractElements* and *purgeElements* share a common attribute in that they purge *unnecessary* elements from the SweepArea. While the function *extractElements* permits access to the unnecessary elements via the returned iterator prior to their removal, the method *purgeElements* instantly deletes them. We have defined two separate methods because the method *purgeElements* can often be implemented more efficiently than consuming the entire iterator returned by a call to the method *extractElements*. For example, the method *purgeElements* is superior to *extractElements* whenever the SweepArea is based on data structures that support bulk deletions. Both methods make use of the remove predicate p_{remove} and have a *reference element* s as parameter. An element $\hat{s}$ from the SweepArea is considered as unnecessary if $p_{remove}(s, \hat{s})$ evaluates to true. Analogously to the method *query*, parameter j is used to deal with asymmetric remove predicates and remove predicates over different argument types.

Note that it is also possible to remove elements from the SweepArea via the iterator returned by the method *query*. Whenever the method *remove* is called on this iterator, the last element returned by the iterator is removed from the underlying SweepArea.

7.3.2 *Use of SweepAreas.* Dittrich et al. [2002] nicely shows how to define the order relation and query and remove predicates to implement a plethora of join algorithms, such as, band-, temporal-, spatial-, and similarity-joins. Due to the generic design, SweepAreas can be adapted easily to multiway joins [Viglas et al. 2003; Golab and Öszu 2003a]. The ADT SweepArea presented here is an extension to meet the requirements of our physical stream operator algebra. In the corresponding algorithms, SweepAreas are used to efficiently manage the state of operators. A SweepArea is maintained for each input of a stateful operator. From a top-level point of view, SweepAreas can be compared with synopses [Arasu et al. 2006] or state machine modules [Raman et al. 2003] used in other DSMSs for state maintenance.

We decided to use SweepAreas inside our operators because a SweepArea gracefully combines functionality for efficient probing and purging on different criteria in a single generic data structure.

(1) *Probing* is primarily based on the tuple component of stream elements. Hence, a data structure for state maintenance should arrange stream elements in an appropriate manner to ensure efficient retrieval.
(2) *Purging* depends on the time interval component of stream elements. Due to temporal expiration, elements in the state become unnecessary over time. Keeping the state information small is important to save system resources.

The smaller the state of an operator, the lower its memory usage and processing costs. Thus, it is essential to get rid of unnecessary elements, namely, the expired ones. Therefore, a data structure for state maintenance should not only provide efficient access to the tuples, but also to the time intervals.

7.3.3 *Implementation Considerations.* Up to this point we defined the semantics of the methods for the ADT SweepArea without discussing any particular implementation issues. Logically, a SweepArea can be viewed as a single data structure storing a collection of elements. In our case, however, a single data structure is often not adequate to ensure both efficient probing and purging. Because of this, we usually construct a SweepArea by combining two data structures. The *primary* data structure contains the elements and arranges them for probing, whereas the *secondary* data structure keeps references to these elements and arranges the references to facilitate purging. The downside of this dual approach is the increased cost of operations that have to be performed on both data structures to ensure consistency.

7.4 Notation

7.4.1 *Algorithm Structure.* Our algorithms obey the following basic structure, which consists of three successive parts. The different parts in the algorithms are separated by blank lines.

(1) *Initialization*. The required variables and data structures are initialized.
(2) *Processing*. A *foreach*-loop processes the elements that arrive from the input streams of an operator.
(3) *Termination*. For the case of finite input streams, additional steps might be necessary to produce the correct result, although all stream elements have already been processed. After executing the termination part, an operator can delete all internal data structures and variables to release the allocated memory resources.

The termination case results from our input-triggered state maintenance concept which updates the operator state only at arrival of new stream elements.

7.4.2 *Specific Syntax and Symbols.* $s \hookleftarrow S_{in}$ denotes element s being delivered from input stream S_{in}. $s \hookrightarrow S_{out}$ indicates that element s is appended to the output stream S_{out}. $\oslash$ denotes the empty stream. The syntax $s := (e, [t_S, t_E))$ means that s is a shortcut for the physical stream element $(e, [t_S, t_E))$. A $\leftarrow$ stands for variable assignments. We distinguish elements retrieved from a SweepArea from those being provided by the input stream by attaching a hat symbol, namely, $\hat{s} := (\hat{e}, [\hat{t}_S, \hat{t}_E))$ signals that $\hat{s}$ belongs to a SweepArea.

7.5 Standard Operators

Despite the different representations, the stream types for a logical operator and its physical counterpart are identical since these depend on the tuple components. Unless explicitly specified, operator predicates and functions

Table V. Overview of Parameters Used to Tailor the ADT SweepArea to Specific Operators.

Operations	Order Relation	Query Predicate	Remove Predicate
Theta-Join	$\leq_{t_E}$	$\theta(e,\hat{e}) \wedge [t_S,t_E) \cap [\hat{t}_S,\hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Duplicate Elimination	$\leq_{t_E}$	$e = \hat{e} \wedge [t_S,t_E) \cap [\hat{t}_S,\hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Difference	$\leq_{t_S}$	$e = \hat{e} \wedge [t_S,t_E) \cap [\hat{t}_S,\hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Scalar Aggregation	$\leq_{t_S}$	$[t_S,t_E) \cap [\hat{t}_S,\hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Grouping with Aggregation	$\leq_{t_S}$	$[t_S,t_E) \cap [\hat{t}_S,\hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Count-based Window	$\leq_{t_S}$	–	$SA.*** = N$
Partitioned Window	$\leq_{t_S}$	$SA.**** = N$	–
Split	$\leq_{t_E}$	$true$	$t_S \geq \hat{t}_E$
Coalesce	$\leq_{t_S}$	$e = \hat{e} \ \wedge \ \hat{t}_E = t_S$	$t_S > \hat{t}_E \ \vee \ \hat{t}_E - \hat{t}_S \geq l_{max}$

Let $(e,[t_S,t_E))$ and $(\hat{e},[\hat{t}_S,\hat{t}_E))$ be the first and second argument of the predicates respectively.

comply with the definitions given in Section 4. Due to space constraints, we can only present the most essential operators. Krämer [2007] presents our complete operator set in detail.

7.5.1 *Cartesian Product/Theta-Join.* Algorithm 2 computes a theta-join over two physical streams by adapting the ripple join technique [Haas and Hellerstein 1999] to data-driven query processing [Graefe 1993]. The join state consists of two SweepAreas, one for each input stream, and a min-priority queue at the output to order the join results. We summarized the Cartesian product and theta-join in a single algorithm as the sole difference is the parameterization of the SweepAreas, in particular the query predicate.

—*SweepArea Parameters.* The initialization fragment of the algorithm defines the parameters for the SweepAreas (see Table V). The internal *order relation* of both SweepAreas is set to $\leq_{t_E}$. The order relations $\leq_{t_S}$ and $\leq_{t_E}$ on physical stream elements are the less-than-or-equal-to relations on *start* and *end timestamps*, respectively, which are a total order. The *query predicate* (see Table V) is evaluated in the method *query*. The choice of the SweepArea implementation and the definition of the query predicate have to ensure that the method *query* returns an iterator over all elements in the SweepArea qualifying for the join result. It checks that: (i) their tuples qualify the join predicate θ, and (ii) their time intervals overlap. The method *purgeElements* removes all elements from the SweepArea that fulfill the *remove predicate* p_{remove} (see Table V). The remove predicate guarantees that only those elements are dropped that will not participate in any future join results. Here, the condition checks a time interval overlap. If the start timestamp of s is equal to or greater than the end timestamp of $\hat{s}$, the respective time intervals of both elements cannot overlap. Hence, this pair of elements does not qualify as join result. For all other cases, an overlap in terms of time intervals is still possible. As a consequence, these elements need to be retained.

—*Choice of SweepArea Implementation.* The SweepArea implementation has to be chosen according to the join predicate. Using an inappropriate SweepArea implementation might produce incorrect join results. While

Algorithm 2. Cartesian Product ($\times$) / Theta-Join ($\bowtie_\theta$)

```
    Input   : physical streams S_in1, S_in2; join predicate θ
    Output  : physical stream S_out
 1  S_out ← ∅;
 2  Let SA_1, SA_2 be the empty SweepAreas(≤_tE, p^θ_query, p_remove);
 3  t_S1, t_S2, min_tS ∈ T ∪ {⊥}; t_S1 ← ⊥; t_S2 ← ⊥; min_tS ← ⊥;
 4  Let Q be an empty min-priority queue with priority ≤_tS;
 5  j, k ∈ {1, 2};
 6  foreach s := (e, [t_S, t_E)) ←↩ S_in_j do
 7      k ← j mod 2 + 1;
 8      SA_k.purgeElements(s, j);
 9      SA_j.insert(s);
10      Iterator qualifies ← SA_k.query(s, j);
11      while qualifies.hasNext() do
12          Element (ê, [t̂_S, t̂_E)) ← qualifies.next();
13          if j = 1 then Q.insert((e ∘ ê, [t_S, t_E) ∩ [t̂_S, t̂_E)));
14          else Q.insert((ê ∘ e, [t_S, t_E) ∩ [t̂_S, t̂_E)));
15      t_S_j ← t_S;
16      min_tS ← min(t_S1, t_S2);
17      if min_tS ≠ ⊥ then
18          while ¬Q.isEmpty() do
19              Element (ẽ, [t̃_S, t̃_E)) ← Q.min();
20              if t̃_S ≤ min_tS then
21                  Q.extractMin() ↪ S_out;
22              else break;
23  while ¬Q.isEmpty() do
24      Q.extractMin() ↪ S_out;
```

list-based SweepArea implementations (nested-loops join) are suitable for arbitrary join predicates, hash-based variants are restricted to equijoins. The Cartesian product can be expressed as a special nested-loops join where join predicate θ always returns true. Note that even for the Cartesian product the query predicate still has to verify time interval overlap.

—*Symmetric Hash Join* (SHJ). Each SweepArea is implemented as a hash table (primary data structure). The elements in the hash table are additionally linked in ascending order by their end timestamps (priority queue as secondary data structure). Inserting a stream element into a SweepArea means: (i) to put it into the hash table based on a hash function, and (ii) to adjust the linkage. The hash function must map value-equivalent elements into the same bucket. Whenever a SweepArea is probed, the hash function determines the bucket containing the qualifying elements. The linkage is used for purging elements effectively from the SweepArea. The method *purgeElements* follows the linkage and discards elements as long as their end timestamp is less than or equal to the start timestamp of the incoming element.

—*Symmetric Nested-Loops Join* (SNJ). Each SweepArea is implemented as an ordered list that links elements in ascending order by end timestamps. Insertion has to preserve the order. Probing the SweepArea is a sequential scan of the linked list. Expired elements are removed from the SweepArea by

Table VI. Example Input Streams

(a) Stream S_1

Tuple	t_S	t_E
c	1	8
a	5	11
d	6	14
a	9	10
b	12	17

(b) Stream S_2

Tuple	t_S	t_E
b	1	7
d	3	9
a	4	5
b	7	15
e	10	18

Table VII. Equijoin of Physical Streams S_1 and S_2

$S_1 \bowtie_= S_2$

Tuple	t_S	t_E
d	6	9
b	12	15

a traversal of the list that stops at the first element with an end timestamp greater than the start timestamp of the incoming element.

—*Asymmetric Variants*. With our SweepArea framework it is also possible to construct and take advantage of the *asymmetric join variants* described in Kang et al. [2003]. In this case, the two previous implementation alternatives for SweepAreas are combined for state maintenance. This means the join utilizes one SHJ and one SNJ SweepArea.

Remark 7.2. Recall that the operators in our physical operator algebra rely on a multithreaded, push-based processing paradigm. Operators can be connected directly, that is, without an interoperator queue. Hence, it is not possible to consume elements across multiple operator input queues in a sorted manner as done in other DSMSs [Arasu et al. 2006; Abadi et al. 2003; Ghanem et al. 2007]. Instead of performing a sort-merge step at the input of an n-ary operator, we employ a priority queue at the operator's output to satisfy the ordering requirement of the output stream.

Example 12. Table VI shows example input streams. The results of an equijoin over streams S_1 and S_2 are listed in Table VII. Two stream elements qualify if their tuples are equal and their time intervals overlap. The join result is composed of the tuple and the intersection of the time intervals. Here, the concatenation function $\circ$ is a projection to the first argument.

7.5.2 *Scalar Aggregation*. Algorithm 3 shows the implementation of the scalar aggregation. Stream processing demands a continuous output of aggregates [Hellerstein et al. 1997]. In our case, an *aggregate* is a physical stream element, that is, it is composed of a tuple containing the aggregate value and a time interval. The state of the scalar aggregation consists of a SweepArea which maintains so-called *partial aggregates*. A partial aggregate is also a physical stream element, but the tuple component stores state information required to compute the final aggregate value. Partial aggregates are updated whenever their time interval intersects with that of an incoming stream element. The

computation of a partial aggregate is finished if it is guaranteed that it will not be affected by any future stream elements. Then, the final aggregate value is computed from the tuple of the corresponding partial aggregate. The aggregate appended to the output stream is composed of the final aggregate value and the time interval of the corresponding partial aggregate.

—*Aggregate Computation.* Algorithm 3 takes three functions as input parameters to compute the results of an aggregate function (f_{agg}). Types $\mathcal{T}_1$ and $\mathcal{T}_2$ match the ones of the corresponding logical operator and specify the input and output type, respectively. The new type $\mathcal{T}_3$ defines the type of partial aggregates, namely, the type of tuples in the SweepArea.

—The *initializer* $f_{init} : \Omega_{\mathcal{T}_1} \rightarrow \Omega_{\mathcal{T}_3}$ is applied to a tuple of an incoming stream element to instantiate the tuple of a partial aggregate.

—The *merger* $f_{merge} : \Omega_{\mathcal{T}_3} \times \Omega_{\mathcal{T}_1} \rightarrow \Omega_{\mathcal{T}_3}$ updates the state information of a partial aggregate. This is accomplished by merging the tuple of an incoming element with the tuple of a partial aggregate in the SweepArea. The result is a new tuple which contains the updated state information.

—The *evaluator* $f_{eval} : \Omega_{\mathcal{T}_3} \rightarrow \Omega_{\mathcal{T}_2}$ takes the tuple of a partial aggregate and computes the final aggregate value.

Property 7.3 (*Expressiveness*). "These three functions can easily be derived for the basic SQL aggregates; in general, any operation that can be expressed as commutative applications of a binary function is expressible." [Madden et al. 2002].

Remark 7.4. Although our approach to manage tuples of partial aggregates in the SweepArea resembles the basic aggregation concept proposed for sensor networks in Madden et al. [2002], we extended it towards the temporal semantics addressed in this work. Bear in mind that our aggregates not only consist of an aggregate value but are also equipped with a time interval. User-defined aggregates [Law et al. 2004] follow a similar pattern for aggregate computation. A user-defined aggregate is a procedure written in SQL, which is grouped into three blocks labeled: `INITIALIZE`, `ITERATE`, and `TERMINATE`. The state of the aggregate is stored in local tables. Like our three functions, these blocks are used to initialize and update the state of the aggregate, and to compute its final value.

—*State of Partial Aggregates.* Aggregate functions can be classified into three basic categories [Gray et al. 1997; Madden et al. 2002]: distributive, algebraic, and holistic. Depending on the category, the size requirements to store the internal state of partial aggregates varies. Recall that the tuple of a partial aggregate encapsulates the required state information. In the following, we discuss how to represent tuples of partial aggregates to meet the requirements of the different categories of aggregate functions.

—For *distributive aggregate functions* like `MIN`, `MAX`, `SUM`, and `COUNT`, a tuple of an element in the SweepArea is either a single tuple from the input stream or a natural number. The tuple corresponds to the aggregate value. Hence, type $\Omega_{\mathcal{T}_3}$ is equal to $\Omega_{\mathcal{T}_2}$ and function f_{eval} is the identity.

—For *algebraic aggregate functions* such as AVERAGE, STANDARD DEVIATION, VARIANCE, and TOP-K, a tuple in the SweepArea needs to contain more information than a single value. However, the total state information stored in a tuple of a partial aggregate is of constant size. For instance, it is necessary to keep the sum and the count for the average. The final aggregate value results from dividing the sum by the count.

—For *holistic aggregate functions* like MEDIAN and COUNT DISTINCT, the state cannot be restricted to a constant size. As a consequence, the tuple of an element in the SweepArea is a data structure, for example, a list, set, or even histogram, built over tuples from the input stream. For COUNT DISTINCT, the tuple could be a hash set, which implicitly eliminates duplicates. The size of this set would be the final aggregate value.

Example 13 (*Average*). Let us consider the SQL aggregate function AVG. The tuples for partial aggregates are pairs $(sum, count)$ with schema (DOUBLE, INTEGER). Let e be the tuple of an incoming stream element, and let (s, c) be the tuple of a partial aggregate. Then, the three functions used to compute the aggregate value would be defined as follows: $f_{init}(e) := (e, 1)$, $f_{merge}((s, c), e) := (s + e, c + 1)$, $f_{eval}((s, c)) := s/c$.

—*SweepArea Parameters and Implementation.* While the remove predicate is equal to that of the previous operator, the query predicate merely checks time interval overlap (see Table V). Algorithm 3 requires the iterators returned by the methods *query* and *extractElements* to be sorted in ascending order by start timestamps. Hence, the order relation of the SweepArea is $\leq_{t_S}$. The implementation of the SweepArea is an ordered list, such as a randomized skip list. Because the method *query* does not need to check value-equivalence but only interval overlap, further implementations supporting range queries over interval data might be preferable, for example, a dynamic variant of the priority search tree [Cormen et al. 2001].

—*Construction of Partial Aggregates.* Whenever the interval of a partial aggregate $(\hat{e}, [\hat{t}_S, \hat{t}_E))$ in the SweepArea intersects with the time interval $[t_S, t_E)$ of the incoming element, an update on the SweepArea has to be performed. Thereby, the interval $[\hat{t}_S, \hat{t}_E)$ is partitioned into contiguous subintervals of maximum length that either do not overlap with $[t_S, t_E)$ or entirely overlap with $[t_S, t_E)$. Each of these new time intervals is assigned with a tuple. Both together form a new partial aggregate. Figure 10 shows the different cases of overlap and the respective computation of partial aggregates. We distinguish between two top-level cases.

—For the case $\hat{t}_S < t_S$, $[t_S, t_E)$ is either contained in $[\hat{t}_S, \hat{t}_E)$ or exceeds it at the right border.

—For the case $\hat{t}_S \geq t_S$, $[t_S, t_E)$ contains $[\hat{t}_S, \hat{t}_E)$ or exceeds it at the left border.

The latter case has to be considered, as our algorithm may generate time intervals with a start timestamp greater than the start timestamp t_S of the latest incoming element.

Algorithm 3. Scalar Aggregation ($\alpha_{f_{agg}}$)

```
   Input  : physical stream S_in; functions f_init, f_merge, f_eval
   Output : physical stream S_out
1  S_out ← ∅;
2  Let SA be the empty SweepArea(≤_{t_S}, p_query, p_remove);
3  foreach s := (e, [t_S, t_E)) ↩ S_in do
4      Iterator qualifies ← SA.query(s, 1);
5      if ¬qualifies.hasNext() then
6          SA.insert((f_init(e), [t_S, t_E)));
7      else
8          T last_{t_E} ← t_S;
9          while qualifies.hasNext() do
10             Element (ê, [t̂_S, t̂_E)) ← qualifies.next();
11             qualifies.remove();
12             if t̂_S < t_S then
13                 SA.insert((ê, [t̂_S, t_S)));
14                 if t_E < t̂_E then
15                     SA.insert((f_merge(ê, e), [t_S, t_E) ∩ [t̂_S, t̂_E)));
16                     SA.insert((ê, [t_E, t̂_E)));
17                 else SA.insert((f_merge(ê, e), [t_S, t̂_E)));
18             else
19                 if [t̂_S, t̂_E) = [t_S, t_E) ∩ [t̂_S, t̂_E) then
20                     SA.insert((f_merge(ê, e), [t̂_S, t̂_E)));
21                 else
22                     SA.insert((f_merge(ê, e), [t̂_S, t_E)));
23                     SA.insert((ê, [t_E, t̂_E)));
24             last_{t_E} ← t̂_E;
25         if last_{t_E} < t_E then SA.insert((f_init(e), [last_{t_E}, t_E))) ;
26     Iterator results ← SA.extractElements(s, 1);
27     while results.hasNext() do
28         Element (ê, [t̂_S, t̂_E)) ← results.next();
29         (f_eval(ê), [t̂_S, t̂_E)) ↪ S_out;
30 Iterator results ← SA.iterator();
31 while results.hasNext() do
32     Element (ê, [t̂_S, t̂_E)) ← results.next();
33     (f_eval(ê), [t̂_S, t̂_E)) ↪ S_out;
```

Depending on the preceding two cases, we create disjoint time intervals by separating the overlapping parts from the unique ones. Conceptually, tuples are built as follows.

(1) The tuple $\hat{e}$ is kept for the new partial aggregate if the generated time interval is a subinterval of $[\hat{t}_S, \hat{t}_E)$ and does not overlap with $[t_S, t_E)$.
(2) The tuple of the new partial aggregate is computed by $f_{init}(e)$ if the generated time interval is a subinterval of $[t_S, t_E)$ and does not overlap with $[\hat{t}_S, \hat{t}_E)$.
(3) The tuple of the new partial aggregate is computed by invoking the aggregate function f_{merge} on the old tuple $\hat{e}$ and the new tuple e if the generated time interval is the intersection of both time intervals.

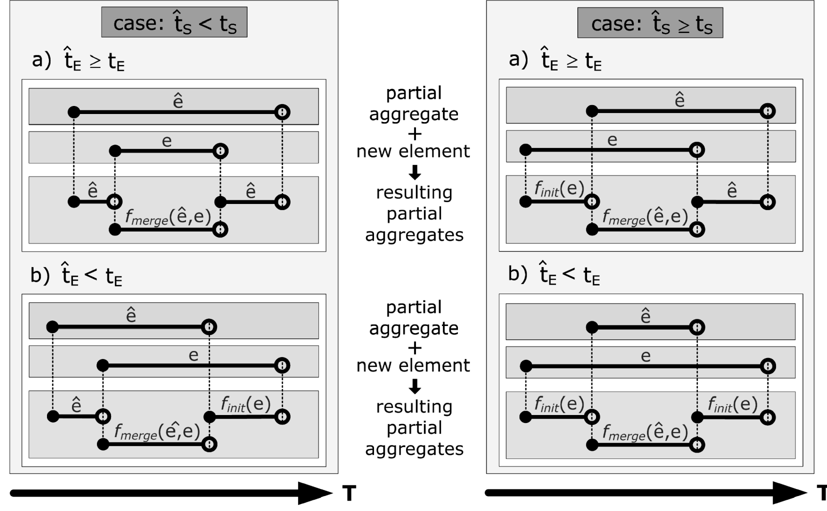

Fig. 10. Incremental computation of aggregates from a partial aggregate $(\hat{e}, [\hat{t}_S, \hat{t}_E))$ and an incoming element $(e, [t_S, t_E))$.

Table VIII. Scalar Aggregation Over Physical Stream S_1

$\alpha_{\text{SUM}}(S_1)$

Tuple	t_S	t_E
c	1	5
$c + a$	5	6
$c + a + d$	6	8
$a + d$	8	9
$a + d + a$	9	10
$a + d$	10	11
d	11	12
$d + b$	12	14
b	14	17

For the special cases, $\hat{t}_S = t_S$, and $\hat{t}_E = t_E$ some intervals shown in Figure 10 may reduce to empty intervals. Although Figure 10 allows these cases for ease of concise presentation, the design of Algorithm 3 prevents any (partial) aggregates with empty intervals entering the state. The interval construction for partial aggregates satisfies the following property.

Property 7.5 (*Interval Construction*). The scalar aggregation eliminates any time interval overlap. As a consequence, the time intervals of aggregates in the output stream are disjoint. If a SweepArea contains multiple partial aggregates, their time intervals are consecutive, that is, they do not overlap and there is no gap between them.

Example 14. Table VIII contains the output stream of the scalar aggregation applied to stream S_1 computing the sum. Because the aggregation

eliminates any time interval overlap, the time intervals of the results are the disjoint segments of all intervals from the input stream. The aggregate value (tuple) of a result is the sum over all tuples from the input stream whose time interval intersects with the interval assigned to the result.

7.6 Window Operators

The *window operators* in our physical operator algebra take a chronon stream as input. Hence, the time intervals of incoming elements have to be chronons. The *split* operator is able to convert an arbitrary physical stream into a chronon stream (see Krämer [2007]). Every window operator assigns a new time interval to the tuple of an incoming element. The window type determines how interval starts and endings are set.

7.6.1 *Time-Based Sliding Window.* The time-based sliding window is a stateless operator that sets the validity of incoming elements to the window size w (see Algorithm 4). Recall that the window size represents a period in (application) time, $w \in T$, $w > 0$.

Algorithm 4. Time-Based Sliding Window (ω_w^{time})

Input : chronon stream S_{in}; window size w
Output : physical stream S_{out}

```
1 S_out ← ∅;
2 foreach s := (e, [t_S, t_S + 1)) ↩ S_in do
3     (e, [t_S, t_S + w)) ↪ S_out;
```

—*Impact.* The implementation nicely illustrates the effect of the time-based sliding window operator on a query plan. To comply with the semantics specified in Section 4.2.1, the time-based sliding window operator should slide a temporal window of size w over its input stream. This means, for a chronon stream S_{in}, at a time instant $t \in T$ all elements of S_{in} with a start timestamp t_S, where $t - w + 1 \leq t_S \leq t$, should be in the window. The physical time-based window achieves this effect by adjusting the validity of each incoming element according to the window size. Whenever the time-based window is installed in a query plan, downstream operators implicitly consider the new validities, namely, the new time intervals. As stateful operators have to keep elements in their state as long as these are not expired, there is an apparent correlation between the window size and the size of the state. Be aware that the time-based window operator does not affect stateless operators such as filter and map because these do not regard time intervals.

Property 7.6 (*Now-Window*). Applying a time-based window with size $w = 1$ to a chronon stream has no effects because the output stream will be identical to the input stream.

Example 15. Table IX(a) defines chronon stream S_3. Table IX(b) shows the results of the time-based sliding window applied to S_3 with window size 50 time units. While the tuples and start timestamps of the input stream are retained

Table IX. Chronon Stream S_3 and Time-Based Window of Size 50 Time Units Over S_3

(a) S_3

Tuple	t_S	t_E
b	1	2
a	3	4
c	4	5
a	7	8
b	10	11

(b) $\omega_{50}^{\text{time}}(S_3)$

Tuple	t_S	t_E
b	1	51
a	3	53
c	4	54
a	7	57
b	10	60

in the output stream, the time-based sliding window sets the end timestamps according to the window size. Each end timestamp has an offset of w chronons relative to its corresponding start timestamp.

8. PHYSICAL QUERY OPTIMIZATION

This section discusses the basics for physical query optimization. First, Section 8.1 defines equivalences for physical streams and physical plans. Then, Section 8.2 describes how to generate a physical plan from its logical counterpart. Appendix B presents physical optimizations, addressing the impact of expiration patterns as well as our specific split and coalesce operators.

8.1 Equivalences

Physical stream and plan equivalence derive from the corresponding logical equivalences.

Definition 8.1 (*Physical Stream Equivalence*). We define two physical streams $S_1^p, S_2^p \in \mathbb{S}_{\mathcal{T}}^p$ to be *equivalent iff* their corresponding logical streams are equivalent.

$$S_1^p \doteq S_2^p :\Leftrightarrow \varphi^{p\mapsto l}(S_1^p) \doteq \varphi^{p\mapsto l}(S_2^p) \tag{16}$$

The function $\varphi^{p\mapsto l}$ transforms the given physical stream S^p into its logical counterpart (see Section 2.6). Be aware that physical stream equivalence neither implies that elements in both streams occur in the same order nor that the time intervals associated to tuples are built in the same way. We already consider two physical streams as equivalent if their snapshots are equal at every time instant.

Definition 8.2 (*Physical Plan Equivalence*). Two physical query plans over the same set of physical input streams are denoted *equivalent* if their results are snapshot-equivalent, namely, iff for every time instant t the snapshots of their output streams at t are equal.

The definition reduces physical plan equivalence to a pure semantic equivalence. If S_1^p and S_2^p are the physical output streams of two query plans respectively, physical plan equivalence enforces that $S_1^p \doteq S_2^p$. Physical optimizations do not conflict with the correctness of query results as long as plan equivalence is preserved.

8.2 Physical Plan Generation

During physical plan generation the operators in the logical plan are replaced with their physical counterparts.

8.2.1 *Standard Operators.* In order to choose a suitable implementation for a standard operator, the common techniques known from conventional database systems can be applied. For example, which type of join implementation is used highly depends on the join predicate. In the case of an equijoin, a hash-based implementation is adequate, whereas a similarity join requires a nested-loops implementation in general.

Section 7 presented an abstract implementation for every standard operator. Various implementations of the same physical operator can be derived from exchanging the SweepAreas. With regard to our join example, Algorithm 2 can be used with hash-based or a list-based SweepAreas, or a mixture [Kang et al. 2003]. Hence, the concrete implementation of a physical operator depends on the choice and parameterization of its SweepAreas. In addition, a DSMS may provide further specializations of these operators optimized for a particular purpose, for example, one-to-one or one-to-many joins. Based on the available metadata, the optimizer determines the best operator implementation and installs the corresponding physical operator in the query plan.

8.2.2 *Window Operators.* With regard to window operators, we distinguish between two cases.

(1) If the window operator in the logical plan refers to a *source stream* being a chronon stream, it can be replaced directly with its physical counterpart.
(2) If the window operator in the logical plan refers to a *derived stream* (subquery) or a source stream that already provides time intervals, then the logical window operator is replaced with a *split operator* having output interval length 1 followed by the corresponding physical window operator.

The first case occurs whenever a raw stream is converted into a physical stream (see Section 2.5.1). The time intervals in such a stream are chronons because they cover only a single time instant. According to our semantics, the logical time-based sliding window operator expands the validity of every tuple at every time instant. A tuple being valid at time instant t in the input becomes valid during all instants in $[t, t+w)$ in the output. If the physical input stream is a chronon stream, the physical time-based sliding window preserves the desired semantics. In the second case, the time interval length exceeds 1 and the time-based sliding window would not produce correct results because it would simply adjust the time intervals to a new length. As a consequence, the physical plan would not generate results snapshot-equivalent to those of the corresponding logical plan. The same argumentation applies to count-based and partitioned windows. To ensure consistent results with our logical algebra, we solely permit chronon streams as inputs for window operators. Any physical stream which is not a chronon stream has to be transformed into a chronon stream first. This can be achieved by placing a split operator with output interval length 1 upstream of the window operator.

The split operator is therefore not only an instrument for physical optimization but actually required in our physical algebra to deal with windows over: (i) subqueries and also (ii) source streams whose tuples are already tagged with a time interval instead of a timestamp.

9. RELATED WORK

In the following, we focus on closely related approaches. For more details on stream processing, we refer the interested reader to the comprehensive surveys presented in Babcock et al. [2002], Golab and Özsu [2003], and Krämer [2007].

9.1 Use of Timestamps

Timestamps are used in todays DSMSs to define windows or control expiration [Babcock et al. 2002; Golab and Özsu 2003]. In recent years we have observed a substantial divergence in the use and creation of timestamps among the various stream processing approaches. Babcock et al. [2002] and Abadi et al. [2003] propose that the user should specify a function for creating the timestamp assigned to a result of an operator. Such a flexible technique has the drawback that query semantics depends on user-defined functions, as downstream operators are sensitive to timestamps. Consequently, query optimization becomes nearly impossible. Babcock et al. suggest the join result to be associated with the maximum timestamp of the two contributing join elements [Babcock et al. 2003], while Chandrasekaran and Franklin [2002] takes the minimal timestamp. The proposals in Golab and Öszu [2003a] and Zhu et al. [2004] keep all timestamps of qualifying elements in the resultant tuples. This leads to serve deficiencies for operators like aggregation and duplicate elimination, as the number of timestamps associated to a result in the output stream is no longer constant.

This lack of consensus in the use of timestamps inspired us to develop an operator algebra that: (i) consistently deals with timestamps and (ii) provides a precisely defined, sound semantics as a foundation for query optimization. Our physical algebra shows that it is sufficient to tag tuples with time intervals to accomplish a cohesive operator model [Krämer and Seeger 2005; Krämer 2007].

9.2 Stream Algebras

A lot of papers focus on individual stream operators rather than on a complete stream algebra. As a consequence, these approaches often rely on specific semantics and operator properties. Due to their restricted scope, they disregard aspects of the larger system picture such as operator plans and operator interaction. Unlike these approaches, we strive to establish a powerful algebra for stream processing that consistently unifies the various stream operators along with an appropriate stream model. Despite the high relevance of a general-purpose algebra for continuous queries over data streams, only a few papers have addressed this complex and broad topic so far.

9.2.1 *STREAM.* Arasu et al. [2006] propose the abstract semantics of CQL. Although our work is closely related to Arasu et al. [2006], there are important

distinctions. (i) Our focus is not on the query language but rather on the operator semantics and implementation. (ii) Our logical algebra defines our operator and query semantics in a concrete and formal manner, rather than in an abstract manner. (iii) Although Arasu et al. sketch query execution, they do not discuss important implementation aspects such as algorithms. However, the implementation of these algorithms is nontrivial for stateful operators. Conversely, we proposed a novel and unique physical algebra that illustrates the implementation of our time-interval approach. (iv) We also implemented the positive-negative approach of CQL, which is explained in more detail by another research group [Ghanem et al. 2007], and compared it with our approach. (v) We proved that our query language is at least as expressive as CQL (see Appendix D). Due to our generic and flexible operator design based on functions and predicates and the fact that our streams are not limited to relational tuples, our approach can easily be enhanced with more powerful operators. (vi) We explicitly discussed logical and physical query optimization issues, including novel transformation rules and the effects of coalesce and expiration patterns.

9.2.2 *Nile.* Just like STREAM, *Nile* [Ghanem et al. 2007] is based on the positive-negative approach (PNA). In contrast to our time-interval approach (TIA), PNA propagates different types of elements, namely positive and negative ones tagged with a timestamp, through an operator plan to control expiration. As a consequence, stream rates are doubled, which means that twice as many elements as for TIA have to be processed by operators in general. However, for queries with aggregation, difference, or count-based windows, PNA profits from reduced latency. A more detailed comparison is given in Appendix C and Krämer [2007].

9.2.3 *Punctuations.* Tucker et al. [2003] propose an interesting, alternative approach to dealing with unbounded stateful and blocking operators. Instead of using windowing constructs, a priori knowledge of a data stream is used to embed *punctuations* in the stream with the aim to permit the use of the aforementioned class of operators. A punctuation can be considered as a predicate on stream elements that must evaluate to false for all elements following the punctuation. Operators are stream iterators that access the input stream elements incrementally and make use of the embedded punctuations. Operators output a punctuated stream and manage a state. Enhancing our operators for punctuated streams would be a general concept for optimizing state maintenance.

9.2.4 *Aurora.* It is somewhat difficult to compare the query semantics of Aurora [Abadi et al. 2003] with our semantics. Because stream elements are tagged with system timestamps, the semantics depends on element arrival and scheduling. Furthermore, the majority of operators are not snapshot-reducible. As a consequence, query optimizations are nearly impossible. Our work has opposed objectives, namely: (i) the definition of an unambiguous query semantics to establish a foundation for query optimization and (ii) the development of a physical operator algebra that produces query answers invariant under

scheduling and allowed optimizations. Aurora mainly aims at satisfying user-defined QoS specifications.

9.2.5 *Other Approaches.* *TelegraphCQ* [Chandrasekaran et al. 2003] has been among the first system proposals for data streams. While highly flexible query processing techniques provide interesting optimization potential for conjunctive queries, their extension to arbitrary query plans that may contain grouping or difference is nontrivial. The declarative query language of TelegraphCQ is *StreaQuel*, a stream-only query language. According to Arasu et al. [2006], a stream-only query language derived from CQL would be quite similar to the purely stream-based approach implemented in TelegraphCQ.

Gigascope [Cranor et al. 2003] is a high-performance stream database for network monitoring applications with its own SQL-style query language termed GSQL, being mostly a restriction of SQL. In Gigascope, windows are strictly tied to the operators. Because any primary operation in GSQL is expressible in CQL [Arasu et al. 2006], it can also be expressed in our language.

The *StreamMill* system [Bai et al. 2006] effectively applies user-defined aggregate functions on a wide range of applications, including data stream mining, streaming XML processing, and RFID event processing. Queries are expressed through ESL, the *Expressive Stream Language*, which extends the current SQL:2003 standards. ESL supports the incremental computation of UDAs as proposed in ATLaS [Wang et al. 2003] and provides additional support for window aggregates.

The general-purpose event monitoring system *Cayuga* [Demers et al. 2007] extends traditional content-based publish-subscribe systems to handle stateful subscriptions. The implementation of Cayuga differs totally from our approach, as it is based on Nondeterministic Finite State Automata (NFA).

The event streaming system *CEDR* [Barga et al. 2007] relies on an operator semantics that derives from view update compliance. A view can be considered as a time-varying relation [Arasu et al. 2006] that is updated on arrival of positive and negative elements, which correspond to insertions and deletions, respectively. A snapshot at time instant t covers all elements being in the view at time t. Hence, the CEDR semantics follows the update semantics presented in Arasu et al. [2006] and Ghanem et al. [2007]. Based on a tritemporal stream model, the CEDR query language supports features such as event sequencing, negation, and temporal slicing.

9.3 Temporal Algebra

Our work exploits the well-understood relational semantics and its temporal extensions to establish a sound semantic foundation for continuous queries over data streams. Hence, our work is closely related to the area of *temporal databases* [Tansel et al. 1993]. In fact, we found the temporal algebra with support for duplicates and ordering proposed in Slivinskas et al. [2001] to be an appropriate starting point for our research. We extended this work towards continuous query processing over data streams as follows. (1) Slivinskas et al. do

not provide a logical algebra as an abstraction of their semantics. They rather specify the semantics of their operations from an implementation point of view, using the λ-calculus. However, the separation of the logical from the physical algebra entails several advantages. Our logical stream algebra defines operator semantics in an explicit, precise, and descriptive manner. Since a logical stream can be viewed as a potentially infinite, temporal multiset, such an algebra is beneficial to the temporal database community as well. It clearly illustrates the snapshot semantics because our logical algebra addresses time instants. Moreover, the logical algebra can be leveraged to prove semantic equivalences of operator plans, independent from any specific implementations. (2) Despite the similarity of using time intervals to denote validity, the temporal operator algebra in Slivinskas et al. [2001] does not satisfy the demanding requirements of stream processing applications like the support of continuous queries on unbounded streams. In particular, the algebra does not provide any windowing constructs nor data-driven operator implementations. Nonetheless, this temporal algebra is a fruitful and suitable basis which can be enhanced for stream processing. Because of this, we have introduced the notion of physical streams and have developed an appropriate physical stream algebra.

10. CONCLUSIONS

Despite the surge in stream processing publications in recent years, the development of a sound semantic foundation for a general-purpose stream algebra has attracted only little research attention. While this article addressed this challenge, it also gave seamless insight into our query processing engine, illustrating the coherence between the individual query processing steps ranging from query formulation to query execution. Hence, this article has fostered the fusion between theoretical results and practical considerations in the streaming context. Our logical algebra precisely defines the semantics of each operation in a direct way over temporal multisets. We are convinced that the logical algebra is extremely valuable, as it separates semantics from implementation. By assigning an exact meaning to any query at any point in time, our logical algebra describes what results to expect from a continuous query, independent of how the system operates internally. Moreover, it represents a tool for exploring and proving equivalences. Our stream-oriented physical algebra relies on a novel stream representation that tags tuples with time intervals to indicate tuple validity. It provides push-based, nonblocking algorithms that harness sweepline techniques and input stream order for the efficient eviction of elements expiring in the operator state due to windowing constructs. From a theoretical point of view, this article carried over the well-known transformation rules from conventional and temporal databases to stream processing, based on the concept of snapshot-equivalence. We enhanced this large set of rules with novel ones for the window operators. Furthermore, we defined equivalences for streams and query plans, pointed out plan generation, as well as algebraic and physical optimizations. We proved that our query language has at least the same expressive power as CQL, while our language stays closer to SQL:2003 standards. From the implementation side, our unique time-interval approach is superior to the

semantically equivalent positive-negative approach for the majority of time-based sliding window queries because it does not suffer from doubling stream rates to signal tuple expiration. In summary, this work established a semantically sound and powerful foundation for data stream management.

REFERENCES

Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. 2003. Aurora: A new model and architecture for data stream management. *VLDB J. 12*, 2, 120–139.

Arasu, A., Babcock, B., Babu, S., McAlister, J., and Widom, J. 2002. Characterizing memory requirements for queries over continuous data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 221–232.

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: Semantic foundations and query execution. *VLDB J. 15*, 2, 121–142.

Babcock, B., Babu, S., Datar, M., and Motwani, R. 2003. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 253–264.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. 2002. Models and issues in data stream systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,* 1–16.

Babu, S., Munagala, K., Widom, J., and Motwani, R. 2005. Adaptive caching for continuous queries. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 118–129.

Bai, Y., Thakkar, H., Wang, H., Luo, C., and Zaniolo, C. 2006. A data stream language and system designed for power and extensibility. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 337–346.

Barga, R. S., Goldstein, J., Ali, M. H., and Hong, M. 2007. Consistent streaming through time: A vision for event stream processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR),* 363–374.

Cammert, M., Krämer, J., Seeger, B., and Vaupel, S. 2008. A cost-based approach to adaptive resource management in data stream systems. *IEEE Trans. Knowl. Data Eng. 20*, 2, 230–245.

Carney, D., Cetintemel, U., Zdonik, S., Rasin, A., Cerniak, M., and Stonebraker, M. 2003. Operator scheduling in a data stream manager. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 838–849.

Chandrasekaran, S., Cooper, O., Deshpande, A., and et al. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

Chandrasekaran, S. and Franklin, M. J. 2002. Streaming queries over streaming data. *In Proceedings of the International Conference on Very Large Databases (VLDB)*, 203–214.

Chaudhuri, S., Motwani, R., and Narasayya, V. 1999. On random sampling over joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 263–274.

Chen, J., DeWitt, D., Tian, F., and Wang, Y. 2000. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 379–390.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms*, 2nd ed. The MIT Press.

Cranor, C. D., Johnson, T., Spatscheck, O., and Shkapenyuk, V. 2003. Gigascope: A stream database for network applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 647–651.

Datar, M., Gionis, A., Indyk, P., and Motwani, R. 2002. Maintaining stream statistics over sliding windows. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA),* 635–644.

Dayal, U. 1987. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the International Conference on Very Large Databases*, 197–208.

DAYAL, U., GOODMAN, N., AND KATZ, R. H. 1982. An extended relational algebra with control over duplicate elimination. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 117–123.

DEMERS, A. J., GEHRKE, J., PANDA, B., RIEDEWALD, M., SHARMA, V., AND WHITE, W. M. 2007. Cayuga: A General purpose event monitoring system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 412–422.

DITTRICH, J.-P., SEEGER, B., TAYLOR, D. S., AND WIDMAYER, P. 2002. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proceedings of the International Conference on Very Large Databases (VLDB),* 299–310.

GAO, D., JENSEN, C. S., SNODGRASS, R. T., AND SOO, M. D. 2005. Join operations in temporal databases. *VLDB J. 14*, 1, 2–29.

GARCIA-MOLINA, H., ULLMAN, J. D., AND WIDOM, J. 2000. *Database System Implementation*. Prentice Hall.

GHANEM, T. M., HAMMAD, M. A., MOKBEL, M. F., AREF, W. G., AND ELMAGARMID, A. K. 2007. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. Knowl. Data Eng. 19*, 1, 57–72.

GOLAB, L. AND OZSU, M. 2003a. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 500–511.

GOLAB, L. AND OZSU, M. 2003b. Issues in data stream management. *SIGMOD Rec. 32*, 2, 5–14.

GOLAB, L., OZSU, M. 2005. Update-Pattern-Aware modeling and processing of continuous queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 658–669.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv. 25*, 2, 73–170.

GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining Knowl. Discov. 1*, 1, 29–53.

GUHA, S., MEYERSON, A., MISHRA, N., MOTWANI, R., AND O'CALLAGHAN, L. 2003. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng. 15*, 3, 515–528.

HAAS, P. J. AND HELLERSTEIN, J. M. 1999. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 287–298.

HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. 1997. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 171–182.

JENSEN, C. S., CLIFFORD, J., ELMASRI, R., GADIA, S. K., HAYES, P. J., AND JAJODIA, S. 1994. A consensus glossary of temporal database concepts. *SIGMOD Rec. 23*, 1, 52–64.

KANG, J., NAUGHTON, J., AND VIGLAS, S. 2003. Evaluating window joins over unbounded streams. In *Proceedings of the International Conference on Data Engineering (ICDE),* 341–352.

KARP, R. M., SHENKER, S., AND PAPADIMITRIOU, C. H. 2003. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst. 28*, 51–55.

KRÄMER, J. 2007. Continuous queries over data streams – Semantics and implementation. Ph.D. thesis, University of Marburg.

KRÄMER, J. AND SEEGER, B. 2004. PIPES - A public infrastructure for processing and exploring streams. In *Proceedings of the ACM SIGMOD International Conference on Database Management*, 925–926.

KRÄMER, J. AND SEEGER, B. 2005. A temporal foundation for continuous queries over data streams. In *Proceedings of the International Conference on Management of Data (COMAD),* 70–82.

LAW, Y.-N., WANG, H., AND ZANIOLO, C. 2004. Query languages and data models for database sequences and data streams. In *Proceedings of the International Conference on Very Large Databases (VLDB),* 492–503.

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

MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. 2002. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Symposium on Operating System Design and Implementation (OSDI)*.

MANKU, G. S. AND MOTWANI, R. 2002. Approximate frequency counts over data streams. In *Proceedings of the International Conference on Very Large Databases (VLDB)*.

NIEVERGELT, J., AND PREPARATA, F. P. 1982. Plane-Sweep algorithms for intersecting geometric figures. *Commun. ACM 25*, 10, 739–747.

PATROUMPAS, K. AND SELLIS, T. K. 2006. Window specification over data streams. In *Proceedings of the EDBT Workshops*, 445–464.

RAMAN, V., DESHPANDE, A., AND HELLERSTEIN, J. M. 2003. Using state modules for adaptive query processing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 353.

ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBE, S. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 249–260.

SELLIS, T. K. 1988. Multiple-Query optimization. *ACM Trans. Database Syst. 13*, 1, 23–52.

SLIVINSKAS, G., JENSEN, C. S., AND SNODGRASS, R. T. 2001. A foundation for conventional and temporal query optimization addressing duplicates and ordering. *IEEE Trans. Knowl. Data Eng. 13*, 1, 21–49.

SQR. 2003. SQR – A stream query repository. http://www.db.stanford.edu/stream/sqr.

SRIVASTAVA, U. AND WIDOM, J. 2004. Flexible time management in data stream systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.*

TANSEL, A., CLIFFORD, J., GADIA, S., JAJODIA, S., SEGEV, A., AND SNODGRASS, R. T. 1993. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings.

TATBUL, N., CETINTEMEL, U., ZDONIK, S. B., CHERNIACK, M., AND STONEBRAKER, M. 2003. Load shedding in a data stream manager. In *Proceedings of the International Conference on Very Large Databases (VLDB),* 309–320.

TUCKER, P. A., MAIER, D., SHEARD, T., AND FEGARAS, L. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng. 15*, 3, 555–568.

TUCKER, P. A., TUFTE, K., PAPADIMOS, V., AND MAIER, D. 2002. NEXMark – A benchmark for queries over data streams. http://www.cse.ogi.edu/dot/niagara/NEXMark.

VIGLAS, S. D. AND NAUGHTON, J. F. 2002. Rate-Based query optimization for streaming information sources. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 37–48.

VIGLAS, S. D., NAUGHTON, J. F., AND BURGER, J. 2003. Maximizing the output rate of multi-join queries over streaming information sources. In *Proceedings of the International Conference on Very Large Databases (VLDB),* 285–296.

WANG, H., ZANIOLO, C., AND LUO, C. 2003. ATLaS: A small but complete SQL extension for data mining and data streams. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1113–1116.

YANG, J. AND WIDOM, J. 2001. Incremental computation and maintenance of temporal aggregates. In *Proceedings of the International Conference on Data Engineering*, 51–60.

YANG, Y., KRÄMER, J., PAPADIAS, D., AND SEEGER, B. 2007. HybMig: A hybrid approach to dynamic plan migration for continuous queries. *IEEE Trans. Knowl. Data Eng. 19*, 3, 398–411.

ZHU, Y., RUNDENSTEINER, E. A., AND HEINEMAN, G. T. 2004. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 431–442.

Received April 2007; revised October 2007; accepted June 2008