A Temporal Foundation for Continuous Queries over Data Streams

Jürgen Krämer and Bernhard Seeger
Dept. of Mathematics and Computer Science, University of Marburg
e-mail: {kraemerj,seeger}@informatik.uni-marburg.de

Despite the surge of research in continuous stream processing, there is still a semantic gap. In many cases, continuous queries are formulated in an enriched SQL-like query language without specifying the semantics of such a query precisely enough. To overcome this problem, we present a sound and well-defined temporal operator algebra over data streams ensuring deterministic query results of continuous queries. In analogy to traditional database systems, we distinguish between a logical and physical operator algebra. While our logical operator algebra specifies the semantics of each operation in a descriptive way over temporal multisets, the physical operator algebra provides adequate implementations in form of stream-to-stream operators. We show that query plans built with either the logical or the physical algebra produce snapshot-equivalent results. Moreover, we introduce a rich set of transformation rules that forms a solid foundation for query optimization, one of the major research topics in the stream community. Examples throughout the paper motivate the applicability of our approach and illustrate the steps from query formulation to query execution.

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

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: continuous query processing, data streams, non-blocking data-driven algorithms, logical and physical operator algebra, query plan construction and optimization, temporal semantics

1. INTRODUCTION

Continuous queries over data streams have been emerged as an important type of queries. Their need is motivated by a variety of applications [Babcock et al. 2002; Golab and Özsu 2003; Carney et al. 2002; Sullivan and Heybey 1998; Cranor et al. 2003; Wang et al. 2003] like network and traffic monitoring. In order to express continuous queries, different query languages have been proposed recently [Abadi et al. 2003; Cranor et al. 2003; Wang et al. 2003; Arasu et al. 2003b; Golab and Özsu 2003]. However, most of these languages lack of a formal foundation since they are solely motivated by providing illustrative examples. This causes a semantic gap that makes it hard or even impossible to compute a deterministic output of a continuous query. This observation was the starting point of our work. We introduce a well-defined and expressive operator algebra with precise semantics for supporting continuous queries over data streams.

The most important task of a data stream management system (DSMS) is to support continuous queries over a set of heterogeneous data sources, mainly data streams. In analogy to traditional database management systems (DBMS), we propose the following well-known steps from query formulation to query execution:

(1) A query has to be expressed in some adequate query language, e. g. a declarative

language with windowing constructs such as CQL [Arasu et al. 2003a].
(2) A logical query plan is built from this syntactical query representation.
(3) Based on algebraic transformation rules, the logical query plan is optimized according to a specific cost model.
(4) The logical operations in the query plan are replaced by physical operators.
(5) The physical query plan is executed.

Due to the fact that in many stream applications, e. g. sensor streams, the elements within a data stream are associated with a timestamp attribute, we decided to define and implement a temporal operator algebra. In this paper, we show that the above mentioned process from query formulation to query execution is also feasible in the context of continuous queries over data streams. While this paper paves the way for rule-based optimization of continuous queries, there are many important optimization problems that may benefit from our approach. Since many queries are long-running, new cost models are required that take stream rates into account [Viglas and Naughton 2002]. Moreover, dynamic query re-optimization at runtime [Zhu et al. 2004] is required to adapt to changes in the system load. Eventually, multi-query optimization [Sellis 1988; Roy et al. 2000] is of utmost importance to save system resources. All of these optimization techniques employ rules for generating equivalent query plans and therefore, they require as a prerequisite a precise semantics of the continuous queries.

In this paper, we introduce a temporal semantics for continuous queries and provide a large set of optimization rules. The main contributions of the paper are:

—We define a logical temporal operator algebra for data streams that extends the well-known semantics of the extended relational algebra [Dayal et al. 1982]. This includes the definition of a novel operator to express both, temporal sliding and fixed windows. This allows us to map continuous queries expressed in a SQL-like query language to a logical operator plan.
—We outline the implementation concepts and advantages of our physical operator algebra, which provides efficient data-driven implementations of the logical operators in form of non-blocking stream-to-stream operators. Moreover, we employ and extend research results from the temporal database community [Slivinskas et al. 2000; 2001], because stream elements handled in our physical operator algebra are associated with time intervals that model their validity independent from the granularity of time. We demonstrate the beneficial usage of these validity information to perform window queries. This allows, for example, to unblock originally blocking operators such as difference or aggregation. Furthermore, we show that a physical operator produces a result that is snapshot-equivalent to the result of its logical counterpart. This proves the correctness of the physical operators and allows to replace a logical operator by its physical counterpart during the query translation process.
—We introduce a rich set of transformation rules, which consists of conventional as well as temporal transformation rules, forming an excellent foundation for algebraic query optimization. Since most of our operations are compliant to the temporal ones proposed by [Slivinskas et al. 2001], we are able to transfer temporal research results to stream processing. Moreover, we propose a novel

kind of physical optimization by introducing two new operators in the stream context, namely coalesce and split. These operators do not have any impact on the semantics, but allow to adaptively change the runtime behavior of a DSMS with respect to stream rates, memory consumption as well as the production of early results.

The rest of this paper is structured as follows. We start with a motivating example as well as the basic definitions and assumptions in Section 2. Then, we formalize the semantics of our operations in Section 3 by defining the logical operator algebra. The main concepts and underlying algorithms of the physical operator algebra are discussed in Section 4. Section 5 shows that our approach represents a good foundation for query optimization. Thereafter, we compare our approach with related ones and conclude finally.

2. PRELIMINARIES

This section motivates our approach by discussing an example query, which is first formulated declaratively and then transformed into an equivalent logical operator plan. Thereafter, we discuss the integration of external input streams and their internal stream representation. Thereby, we introduce underlying assumptions and give basic definitions.

2.1 A Running Example

At first, let us describe our example application scenario that represents an abstraction from the Freeway Service Patrol project. We consider a highway with five lanes where loop detectors are installed at measuring stations. Each measuring station consists of five detectors, one detector per lane. Each time a vehicle passes such a sensor, a new record is generated. This record contains the following information: lane at which the vehicle passed the detector, the vehicle's speed in meters per second, its length in meters and a timestamp. Hence, each detector generates a stream of records. In our application, the primary goal is to measure and analyze the traffic flow. In the following subsections, we give a brief overview of how we model, express, and execute queries in this use-case using our semantics and stream infrastructure [Krämer and Seeger 2004].

2.2 Query Formulation

The focus of this paper is neither on the definition of an adequate query language for continuous query processing over data streams nor on the translation of language constructs to logical operator plans. Instead, our goal is to establish a platform for possible stream query languages by defining a sound and expressive operator algebra with a precise semantics. In order to illustrate the complete process from query formulation to query execution as discussed in the introduction, we express an example query in some fictive SQL-like query language using the sliding window expressions from CQL [Arasu et al. 2003a].

Example: A realistic query in our running example might be: *"At which measuring stations of the highway has the average speed of vehicles been below 15 m/s over the last 15 minutes."* This query may indicate traffic-congested sections of the

highway. Let us assume that our query addresses 20 measuring stations. Then, the following text represents the query expressed in our fictive query language:

```
SELECT sectionID
    FROM (SELECT AVG(speed) AS avgSpeed, 1 AS sectionID
          FROM HighwayStream1 [Range 15 minutes]
          UNION ALL
          ...
          UNION ALL
          SELECT AVG(speed) AS avgSpeed, 20 AS sectionID
          FROM HighwayStream20 [Range 15 minutes]
    )
WHERE avgSpeed < 15;
```

2.3 Stream Types

In analogy to traditional DBMS, we distinguish between the logical operator algebra and its implementation, the physical operator algebra. We use the term *logical streams* to denote streams processed in the logical operator algebra, whereas *physical streams* refer to the ones processed in the physical operator algebra. In addition to logical and physical streams as our internal stream types, we also consider *raw input streams* as a third type of streams that model those arriving at our DSMS.

2.3.1 *Raw Input Streams.* The representation of the elements from a raw input stream depends on the specific application. We assume that an arbitrary but fixed schema exists for each raw input stream providing the necessary metadata information about the stream elements. However, this schema is not restricted to be relational, since our operators are parameterized by arbitrary functions and predicates. Our approach is powerful enough to support XML streams.

Let Ω be the universe, i. e. the set of all records of any schema.

Definition 2.1. (*Raw Input Stream*) A raw input stream S^r is a possibly infinite sequence of records $e \in \Omega$ sharing the same schema. $\mathbb{S}^r$ denotes the set of all raw input streams.

Note that this definition corresponds to the one of a list. Thus, a raw input stream may contain duplicates, and the ordering of its elements is significant.

Example: For simplicity reasons, we focus on the following flat schema in our example:

HighwayStream(`short` *lane,* `float` *speed,* `float` *length,* `Timestamp` *timestamp*);

A measuring station might generate the following raw input stream:

```
(5; 18.28; 5.27; 03/11/1993 05:00:08)
(2; 21.33; 4.62; 03/11/1993 05:01:32)
(4; 19.69; 9.97; 03/11/1993 05:02:16)
                    ...
```

2.3.2 *Internal Streams.* A *physical stream* is similar to a raw input stream, but each record is associated with a time interval modeling its validity. In general, this

validity refers to application time and not to system time. As long as such a stream element is valid, it is processed by the operators of the physical operator algebra. An element expires when it has no impact on future results anymore. Then, it can be removed from the system. In a *logical stream*, we break up the time intervals of a physical stream element into chronons that correspond to time units at finest time granularity.

In the following, we formalize our notions and representations of logical and physical streams. In particular, we show how a raw input stream is mapped to our internal representation and provide an equivalence relation for transforming a physical stream into a logical stream and vice versa.

2.4 Basic Definitions

Let $\mathbb{T} = (T; \leq)$ be a discrete time domain as proposed by [Bettini et al. 1997]. Let $\mathbb{I} := \{[t_S, t_E) \in T \times T \mid t_S < t_E\}$ be the set of time intervals.

Definition 2.2. (*Physical Stream*) A pair $S^p = (M, \leq_t)$ is a physical stream, if

—M is a potentially infinite sequence of tuples $(e, [t_S, t_E))$, where $e \in \Omega$ and $[t_S, t_E) \in \mathbb{I}$,

—all elements of M share the same schema,

—$\leq_t$ is the order relation over M such that tuples $(e, [t_S, t_E))$ are lexicographically ordered by timestamps, i. e. primarily by t_S and secondarily by t_E.

$\mathbb{S}^p$ denotes the set of all physical streams.

The meaning of a stream tuple $(e, [t_S, t_E))$ is that a record e is valid during the half-open time interval $[t_S, t_E)$. The schema of a physical stream is a combination of the record schema and a temporal schema that consists of two time attributes modeling the start and end timestamps.

Our approach relies on multisets for the following two reasons. First, applications may exist where duplicates in a raw input stream might arise. In our example, this would occur if two vehicles with the same length and speed would pass the same sensor within one second (assuming that the finest time resolution of the detectors is in seconds). Consequently, this would result in two identical records. Second, operators like projection may produce duplicates during runtime, even if all elements of the raw input stream are unique. In this case, the term duplicates has a slightly different meaning and we use *value-equivalent* stream elements instead.

Definition 2.3. (*Value-equivalence*) Let $S^p = (M, \leq_t) \in \mathbb{S}^p$ be a physical stream. We denote two elements $(e, [t_S, t_E)), (\hat{e}, [\hat{t}_S, \hat{t}_E)) \in M$ as value-equivalent, *iff* $e = \hat{e}$.

Note that ordering by $\leq_t$ enforces no order within real duplicates, i. e., when the records as well as time intervals of two elements are equal.

2.5 Transformation

Now we describe the transformation of a raw input stream $S^r \in \mathbb{S}^r$ into a physical stream $S^p \in \mathbb{S}^p$. Especially when sensors are involved, many applications produce a raw input stream where the elements are already associated with a timestamp attribute. Typically, these streams are implicitly ordered by their timestamps. This holds for instance in our running example. If streams arrive at a DSMS out of order

and uncoordinated with each other, e. g. due to latencies introduced by a network, techniques like the ones presented in [Srivastava and Widom 2004] can be applied. It is also possible that a stream does not provide any temporal information. In this case, a DSMS can stamp the elements at their arrival by using an internal system clock.

We then use the start timestamp of each raw input stream element as the start timestamp of a physical stream tuple. The corresponding end timestamp is set to infinity because initially we assume each record to be valid forever. That means, we map each element e in S^r to a tuple $(e, [t_S, \infty))$ in S^p where t_S is the explicit timestamp retrieved from e. This implies that the order of S^r is preserved in S^p. The schema of S^p extends the schema of S^r by two additional timestamp attributes modeling the start and end timestamps.

Example: Applying the transformation to the raw input stream of our running example would produce the following physical stream of tuples `(record, time interval)`:

```
((5; 18.28; 5.27; 03/11/1993 05:00:08), [03/11/1993 05:00:08, ∞))
((2; 21.33; 4.62; 03/11/1993 05:01:32), [03/11/1993 05:01:32, ∞))
((4; 19.69; 9.97; 03/11/1993 05:02:16), [03/11/1993 05:02:16, ∞))
                                ...
```

2.6 Window Operations

The usage of windows is a commonly applied technique in stream processing mainly for the following reasons [Golab and Özsu 2003]:

—At any time instant often an excerpt of a stream is only of interest.

—Stateful operators such as the difference would be blocking in the case of unbounded input streams.

—The memory requirements of stateful operators are limited, e. g. in a join.

—In temporally ordered streams, newly arrived elements are often more relevant than older ones.

In our logical as well as in our physical operator algebra, we model windows by introducing a novel window operator ω that assigns a finite validity to each stream element. For a given physical input stream, this is easily achieved by setting the end timestamp of each incoming stream element, which is initially set to infinity, to a certain point in time according to the type and size of the window.

Let $S^p = (M, \leq_t) \in \mathbb{S}^p$ be a physical stream. Let $w \in T$ be the window size. By using the window operator $\omega_w : \mathbb{S}^p \times T \rightarrow \mathbb{S}^p$, we are able to perform a variety of continuous window queries involving the following types of windows (see Figure 1):

—*Sliding windows:* In order to retrieve sliding window semantics, the window operator ω_w sets the end timestamp t_E of each physical stream tuple $(e, [t_S, \infty)) \in M$ to $t_S + w$. This means that each element e is valid for w time units starting from its corresponding start timestamp t_S.

—*Fixed windows:* In the case of fixed windows [Sullivan and Heybey 1998], we divide the time domain $\mathbb{T}$ in sections of fixed size $w \in T$. Hence, each section

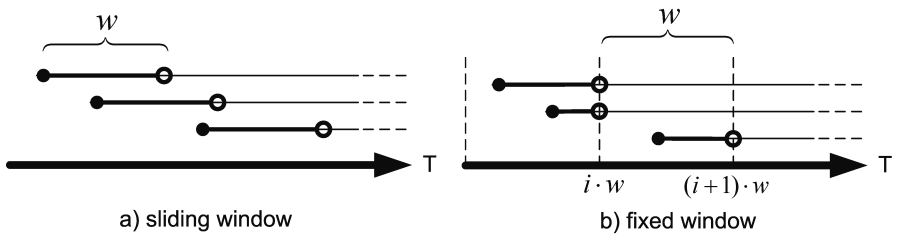

Fig. 1. Windowing constructs

contains exactly w subsequent points in time, where 0 stands for the earliest time instant. Thus, section i starts at $i \cdot w$ where $i \in \mathbb{N}_0$. Fixed window semantics can be obtained, if the window operator ω_w sets the end timestamp t_E of each physical stream tuple $(e, [t_S, \infty)) \in M$ to the point in time where the next section starts. Consequently, for a given element $(e, [t_S, \infty)) \in M$, the window operator determines the closest point in time $t_E = i \cdot w$ with $t_S < t_E$.

Note that it is sufficient for performing continuous window queries to place a single window operator on each path from a source to a sink in a query plan in order to set the validity of each record. These window operators are typically located near the sources of a query plan.

Example: Applying a sliding window of 15 minutes to the physical stream in our example would change the time intervals as follows:

```
((5; 18.28; 5.27; 03/11/1993 05:00:08),
                                [03/11/1993 05:00:08, 03/11/1993 05:15:08))
((2; 21.33; 4.62; 03/11/1993 05:01:32),
                                [03/11/1993 05:01:32, 03/11/1993 05:16:32))
((4; 19.69; 9.97; 03/11/1993 05:02:16),
                                [03/11/1993 05:02:16, 03/11/1993 05:17:16))
                                ...
```

At this point, we want to sketch the basic ideas of our physical algebra approach with regard to windowing constructs: We have physical streams consisting of record/time-interval pairs. The time intervals model the validity of each record which in turn is set via our window operator. The physical operators are aware of the time intervals and use them effectively to guarantee non-blocking behavior as well as limited memory requirements. Based on these physical operators we are able to build query plans that perform continuous window queries over arbitrary data streams while ensuring deterministic semantics.

Before we go into the details of the physical algebra in Section 4, we will first start the discussion of the logical algebra in the next section. The reason for introducing a logical algebra is similar to the approach in a traditional DBMS. The logical algebra abstracts from the physical implementation of the operators, while

providing powerful algebraic transformation rules to rearrange operators in a query plan.

3. LOGICAL OPERATOR ALGEBRA

This section formalizes the term *logical stream* and shows how a logical stream is derived from its physical counterpart. Then, the basic operators of our logical operator algebra are introduced by extending the work on multisets [Dayal et al. 1982] towards a temporal semantics and windowing constructs.

3.1 Logical Streams

Definition 3.1. (*Logical Stream*) A logical stream S^l is a possibly infinite multiset of triples (e, t, n) composed of a record $e \in \Omega$, a point in time $t \in T$, and a multiplicity $n \in \mathbb{N}$. All records e of a logical stream belong to the same schema. Moreover, the following condition holds for a logical stream S^l: $\forall\ (e,t,n) \in S^l.\ \nexists\ (\hat{e},\hat{t},\hat{n}) \in S^l.\ e = \hat{e}\ \wedge\ t = \hat{t}$. Let $\mathbb{S}^l$ be the set of all logical streams.

The condition in the definition ensures that exactly one element (e, t, n) exists in S^l for each record e valid at a point in time t. To put it in other words: The projection on the first two attributes of each stream triple in the set representation of a logical stream is unique.

A stream triple (e, t, n) has the following semantics: An element e is valid at time instant t and occurs exactly n times. Since we treat a logical stream as a multiset, we additionally store the multiplicity of each record in analogy to [Dayal et al. 1982]. This logical point of view implies that all records, their validity as well as their multiplicity are known in advance. We do not take the order in a logical stream into account. Therefore, it is only relevant in the logical model, when a record e is valid and how often it occurs at a certain point in time t.

The schema of a logical stream is composed of the record schema and two additional attributes, namely a timestamp and the multiplicity.

3.1.1 *Transformation: Physical to Logical Stream.* Let $S^p = (M, \leq_t) \in \mathbb{S}^p$ be a physical stream. We define the transformation $\tau : \wp(\Omega \times \mathbb{I}) \rightarrow \mathbb{S}^l$ from a physical stream S^p into its logical counterpart as follows:

$$\tau(M) := \{(e,t,n) \in \Omega \times T \times \mathbb{N} \mid n = |\{(e,[t_S,t_E)) \in M \mid t \in [t_S,t_E)\}|\}$$

For each tuple $(e, [t_S, t_E)) \in M$, we split the associated time interval into points of time at finest time granularity. Thus, we get all instants in time when the record e is valid. Since we allow value-equivalent elements in a physical stream, we have to add the multiplicity n of a record e at a certain point in time t.

3.2 Basic Operators

In our logical operator algebra, we introduce the following operations as basic ones since they are minimal and orthogonal [Slivinskas et al. 2001]: filter(σ), map (μ), Cartesian product ($\times$), duplicate elimination (δ), difference ($-$), group (γ), aggregation (α), union ($\cup$) and window (ω). Section 3.3 reports the definition of more complex operations derived from the basic ones, e. g. a join.

3.2.1 *Filter.* Let $\mathbb{P}$ be the set of all well-defined filter predicates. A filter $\sigma : \mathbb{S}^l \times \mathbb{P} \rightarrow \mathbb{S}^l$ returns all elements of a logical stream $S^l \in \mathbb{S}^l$ that fulfill the predicate $p \in \mathbb{P}$ with $p : (\Omega \times T) \rightarrow \{true, false\}$. We follow the notation of the extended relational algebra and express the argument predicate as subscript. Note that our definition also allows temporal filtering.

$$\sigma_p(S^l) := \{(e, t, n) \in S^l \mid p(e, t)\} \tag{1}$$

The schema of the logical stream S^l remains unchanged, if a filter operation is performed.

3.2.2 *Map.* Let $\mathbb{F}_{map}$ be the set of all mapping functions. The map operator $\mu_f : \mathbb{S}^l \times \mathbb{F}_{map} \rightarrow \mathbb{S}^l$ applies a mapping function f given as subscript on the record of each stream element in a logical stream $S^l \in \mathbb{S}^l$. Let $f \in \mathbb{F}_{map}$ with $f : \Omega \rightarrow \Omega$. Note, that f can also express an n-ary function due to the definition of Ω as a universe of all elements.

$$\mu_f(S^l) := \{(e, t, n) \mid n = \sum_{\{(\hat{e},t,\hat{n}) \in S^l | f(\hat{e}) = e\}} \hat{n}\} \tag{2}$$

This definition is more powerful than the projection operator of the relational algebra because the mapping function may generate new attributes or even new records. Thus, the schema of the resulting logical stream essentially depends on the mapping function. Note, that the mapping function does not change the timestamp attribute of an element.

3.2.3 *Cartesian Product.* The Cartesian product $\times : \mathbb{S}^l \times \mathbb{S}^l \rightarrow \mathbb{S}^l$ of two logical streams $S_1^l, S_2^l \in \mathbb{S}^l$ is defined by:

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

For each pair of elements from S_1^l and S_2^l valid at the same point in time t, a new result is created as concatenation of both records by the auxiliary function $\circ : \Omega \times \Omega \rightarrow \Omega$.
The multiplicity of the result is determined by the product of the multiplicities of the two qualifying elements. The resulting schema of the logical output stream is a concatenation of both record schemas, the timestamp, and the multiplicity attribute.

3.2.4 *Duplicate Elimination.* The duplicate elimination is an unary operation $\delta : \mathbb{S}^l \rightarrow \mathbb{S}^l$ that produces for a given logical stream $S^l \in \mathbb{S}^l$ a *set* of elements. This implies that each element in S^l occurs exactly once.

$$\delta(S^l) := \{(e, t, 1) \mid \exists\ n.\ (e, t, n) \in S^l\} \tag{4}$$

The definition intuitively shows how duplicate elimination works, because the multiplicity for each element in S^l is simply set to 1. The schema of a logical stream after a duplicate elimination corresponds to that of the logical input stream.

3.2.5 *Difference.* Applying a difference operation $- : \mathbb{S}^l \times \mathbb{S}^l \rightarrow \mathbb{S}^l$ enforces that all elements of the second logical stream $S_2^l \in \mathbb{S}^l$ are subtracted from the first logical stream $S_1^l \in \mathbb{S}^l$ in terms of their multiplicities. Thus, the schema of the difference matches that of S_1^l. Obviously, a difference operation can only be performed if the

schemas of both input streams are compliant.

$$-(S_1^l, S_2^l) := \{(e,t,n) \mid \exists\ n_1.\ (e,t,n_1) \in S_1^l\ \wedge\ \exists\ n_2.\ (e,t,n_2) \in S_2^l \\ \wedge\ n = n_1 \ominus n_2\ \wedge\ n > 0)\ \vee\ ((e,t,n) \in S_1^l\ \wedge\ \nexists\ n_2.\ (e,t,n_2) \in S_2^l)\} \\ \text{where } n_1 \ominus n_2 := \begin{cases} n_1 - n_2 & \text{, if } n_1 > n_2 \\ 0 & \text{, otherwise} \end{cases} \tag{5}$$

This definition distinguishes between two cases: The first one assumes that an element of S_1^l exists that is value-equivalent to one of S_2^l and both elements are valid at the same point in time t. Then, the resulting multiplicity is the subtraction of the corresponding multiplicities. An element only appears in the output if its resulting multiplicity is greater than 0. In the second case, no element of S_2^l matches with an element of S_1^l. In this case the element is retained.

At the end of this definition, we want to highlight one major benefit of our descriptive logical algebra approach, namely that the operator semantics can be expressed very compact and intuitive. For instance, the difference is simply reduced to the difference in multiplicities, whereas related approaches using the λ-calculus [Slivinskas et al. 2001] hide this property and turn out to be more complicated.

3.2.6 *Group.* Let $\mathbb{F}_{group}$ be the set of all grouping functions. The group operation

$$\gamma_f : \mathbb{S}^l \times \mathbb{F}_{group} \rightarrow (\underbrace{\mathbb{S}^l \times \ldots \times \mathbb{S}^l}_{\text{k times}})$$

produces a tuple of logical streams. It assigns a group to each element of a logical stream $S^l \in \mathbb{S}^l$ based on a grouping function $f \in \mathbb{F}_{group}$ with $f : \Omega \times T \rightarrow \{1, \ldots, k\}$. Each group S_j^l represents a new logical stream for $j \in \{1, \ldots, k\}$ having the same schema as S^l.

$$\gamma_f(S^l) := (S_1^l, \ldots, S_k^l) \\ \text{where } S_j^l := \{(e,t,n) \in S^l \mid f(e,t) = j\}. \tag{6}$$

The group operation solely assigns elements to groups without modifying them. The j-th group contains all elements for which the grouping function f returns j. This definition differs from its relational counterpart which includes an additional aggregation step.

We also define a projection operator, which is a map operator

$$\pi :\ (\underbrace{\mathbb{S}^l \times \ldots \times \mathbb{S}^l}_{\text{k times}}) \times \mathbb{N}\ \rightarrow\ \mathbb{S}^l$$

that is typically used in combination with the group operation. For a given index j, π returns the j-th logical output stream (group): $\pi_j(S_1^l, \ldots, S_k^l) := S_j^l$.

3.2.7 *Aggregation.* Let $\mathbb{F}_{agg}$ be the set of all well-defined aggregation functions. The aggregation operation $\alpha_f : \mathbb{S}^l \times \mathbb{F}_{agg} \rightarrow \mathbb{S}^l$ invokes an aggregation function $f \in \mathbb{F}_{agg}$ with $f : \mathbb{S}^l \rightarrow \Omega$ on all elements of a logical stream $S^l \in \mathbb{S}^l$ that are valid at the same point in time t:

$$\alpha_f(S^l) := \{(agg, t, 1) \mid agg = f(\{(e,t,n) \in S^l\})\} \tag{7}$$

The aggregation eliminates duplicates because an aggregate is computed on all elements valid at the same point in time. Thus, the aggregation operator returns

a set. The schema of a logical stream after an aggregation obviously depends on the aggregation function, but only the record schema has to be adopted, while the timestamp and multiplicity attributes of the input schema remain unchanged.

Contrary to DBMS, our definitions of group and aggregation additionally offer to use both operations independently in query plans. For example, it is possible to apply an aggregation to a stream without grouping.

3.2.8 *Union.* The union operation $\cup_+ : \mathbb{S}^l \times \mathbb{S}^l \rightarrow \mathbb{S}^l$ merges two logical data streams. Its result contains all elements of S_1^l and $S_2^l \in \mathbb{S}^l$:

$$\begin{gathered} \cup_+(S_1^l, S_2^l) := \{(e, t, n_1 + n_2) \mid \\ n_i = \begin{cases} \hat{n} & , \exists\ (e, t, \hat{n}) \in S_i^l \\ 0 & , \text{otherwise} \end{cases} \quad \text{for } i \in \{1, 2\}\} \end{gathered} \tag{8}$$

If an element only occurs in a single input stream, it is directly added to the result. If the same record is contained in both input streams and valid at the same point in time t, both instances are combined to a single element by summing up their multiplicities. Note, that a union can only be performed if both logical input streams are schema-compliant. Then, the resulting schema is taken from the more general input schema.

3.2.9 *Window.* The window operator $\omega_w : \mathbb{S}^l \times T \rightarrow \mathbb{S}^l$ restricts the validity of each record according to the window type and size $w \in T$. We assume as a precondition of the input stream that each record has an infinite validity as already mentioned in Section 2.6.

Let S^l be a logical stream whose records have an infinite validity. Therefore, the multiplicity of a record in S^l is monotonically increasing over time. We differ between the following two types of window operations:

(1) *Sliding Window:* Informally, a sliding window ω_w^s sets the validity of a record, which is valid for the first time at a starting point $t_S \in T$, to w time units, i. e., the element is valid from t_S to $t_S + w - 1$. However, the multiplicity of a record may change over time. An increase in the multiplicity at a certain point in time indicates that further value-equivalent elements start to be valid at this time instant. Consequently, we also have to set the validity of these value-equivalent elements correctly by assigning a validity of w time units relative to their starting points.

$$\begin{gathered} \omega_w^s(S^l) := \{(e, t, n) \mid \exists\ \hat{n}.\ (e, t, \hat{n}) \in S^l \\ \wedge\ [(\exists\ \tilde{n}.\ (e, t - w, \tilde{n}) \in S^l\ \wedge\ n = \hat{n} - \tilde{n}) \\ \vee\ (\nexists\ \tilde{n}.\ (e, t - w, \tilde{n}) \in S^l\ \wedge\ n = \hat{n})]\} \end{gathered} \tag{9}$$

A sliding window is expressed by setting the multiplicity n of a record e valid at a time instant t to the difference of the multiplicities $\hat{n}$ and $\tilde{n}$. Here, $\hat{n}$ and $\tilde{n}$ refer to the multiplicity of the record e at time t and $t - w$, respectively. If no record e exists at time instant $t - w$ in S^l, n is set to $\hat{n}$.

(2) *Fixed Window:* In the case of a fixed window ω_w^f, the time domain $\mathbb{T}$ is divided into sections of size $w \in T$. At first, we determine all starting points $t_S \in T$ of a record. Then, we determine the start of the next section $i \cdot w$ which is larger but temporally closest to t_S. The validity of each record is set to the start of

the next section.

$$\begin{gathered}\omega_w^f(S^l) := \{(e,t,n) \mid \exists\ \hat{n}.\ (e,t,\hat{n}) \in S^l \\ \wedge\ \exists\ i \in \mathbb{N}_0.\ (i \cdot w \leq t\ \wedge\ \forall\ c \in \mathbb{N}.\ (c > i \Rightarrow c \cdot w > t)) \\ \wedge\ [(\exists\ \tilde{n}.\ (e,(i \cdot w) - 1, \tilde{n}) \in S^l\ \wedge\ n = \hat{n} - \tilde{n}) \\ \vee\ (\nexists\ \tilde{n}.\ (e,(i \cdot w) - 1, \tilde{n}) \in S^l\ \wedge\ n = \hat{n})]\}\end{gathered} \tag{10}$$

In contrast to the definition of the sliding window, we consider the multiplicity at time instant $(i \cdot w) - 1$ which corresponds to the multiplicity of the record e at the last point in time belonging to the previous section. The parameter i is chosen such that the start of the section $i \cdot w$ is the timely closest start of a section with respect to t.

The schema of the resulting logical stream after a window operator is identical to that of the logical input stream.

3.3 Derived Operations

In this section, we shortly adapt some common but more complex operations known from traditional DBMS towards continuous query processing. We do not consider the following operations logically as basic operations, since they can be derived from the basic ones defined in previous section.

Let S_1^l, S_2^l be logical streams.

3.4 Theta-Join

A theta join is given by $\bowtie_{p,f}\colon \mathbb{S}^l \times \mathbb{P} \times \mathbb{F}_{map} \rightarrow \mathbb{S}^l$. Let p be a filter predicate that selects the qualifying join results from the Cartesian product. Let f be a mapping function that creates the resulting join tuples. We define a theta-join as:

$$\bowtie_{p,f}\ (S_1^l, S_2^l) := \mu_f(\sigma_p(S_1^l \times S_2^l)) \tag{11}$$

3.5 Semi-Join

A semi-join is a special join operation that returns all elements of S_1^l that join with an element of S_2^l according to a join predicate p. For that reason, the mapping function f in the join definition is replaced by a projection on the schema of S_1^l.

$$\ltimes_p(S_1^l, S_2^l) := S_1^l \bowtie_{p,\mu_{\pi(S_1^l)}} \delta(S_2^l) \tag{12}$$

3.6 Intersection

The intersection, $\cap : \mathbb{S}^l \times \mathbb{S}^l \rightarrow \mathbb{S}^l$, of two logical streams S_1^l and S_2^l can be expressed with the help of the difference operation.

$$\cap(S_1^l, S_2^l) := S_1^l - (S_1^l - S_2^l) \tag{13}$$

3.7 Max-Union

The max-union operation, $\cup_{max} : \mathbb{S}^l \times \mathbb{S}^l \rightarrow \mathbb{S}^l$ sets the multiplicity of an element to its maximum multiplicity in one of the logical input streams $S_1^l, S_2^l \in \mathbb{S}^l$.

$$\begin{array}{c} \cup_{max}(S_1^l, S_2^l) := \{(e,t,n) \mid (\exists\ n_1.\ (e,t,n_1) \in S_1^l \\ \wedge\ \exists\ n_2.\ (e,t,n_2) \in S_2^l\ \wedge\ n = max\{n_1, n_2\}) \\ \vee\ (\exists\ n_j.\ (e,t,n_j) \in S_j^l\ \wedge\ \nexists\ n_k.\ (e,t,n_k) \in S_k^l \\ \wedge\ n = n_j \text{ for } j,k \in \{1,2\}\ \wedge\ j \neq k)\} \\ = (S_1^l - S_2^l) \cup_+ (S_2^l - S_1^l) \cup_+ (S_1^l \cap S_2^l) \end{array} \tag{14}$$

This definition complies with the one proposed by [Slivinskas et al. 2001].

3.8 Strict Difference

Due to multiset semantics we also want to introduce a strict difference operation which differs from the difference presented in Subsection 3.2.5 by eliminating duplicates in the result:

$$-_{strict}(S_1^l, S_2^l) := S_1^l - (S_1^l \ltimes_= S_2^l) \tag{15}$$

The semi-join $\ltimes_=$ determines all elements in S_1^l that are equal to an element in S_2^l.

However, from an implementation point of view it may not be sufficient to compose these operations of the basic ones. For instance, the join can be implemented much more effectively and efficiently from scratch. For that reason, our infrastructure for stream processing, called PIPES, provides specific implementations in addition.

3.9 Logical Query Plans

A query formulated in some query language is generally translated into a semantically equivalent algebraic expression. Such an algebraic expression consists of a composition of logical operators. For our logical operator algebra this can be achieved similarly to traditional databases where SQL is translated into a logical operator plan in the extended relational algebra.

Example: The left drawing in Figure 2 depicts the logical query plan that results from mapping the query presented in Section 2.2 to the operators in our logical operator algebra. At first, the validity of the stream elements is set to 15 minutes, then a map to the attributes `speed` and `sectionID` is performed. Afterwards, the average speed is computed and all streams are merged, followed by a filter operation that selects all stream elements with an average speed lower than 15 m/s. Finally, a projection delivers the IDs of the qualifying sections.

4. PHYSICAL OPERATOR ALGEBRA

From an implementation perspective, it is not satisfying to process logical streams directly because this would cause a significant computational overhead. Since a physical stream has a much compacter representation of the same temporal information, we decided to implement an algebra over physical streams in PIPES [Krämer and Seeger 2004], our infrastructure for data stream processing.

The basic idea is to use time intervals to express the validity of stream elements. To the best of our knowledge, this approach is unique in the stream community.

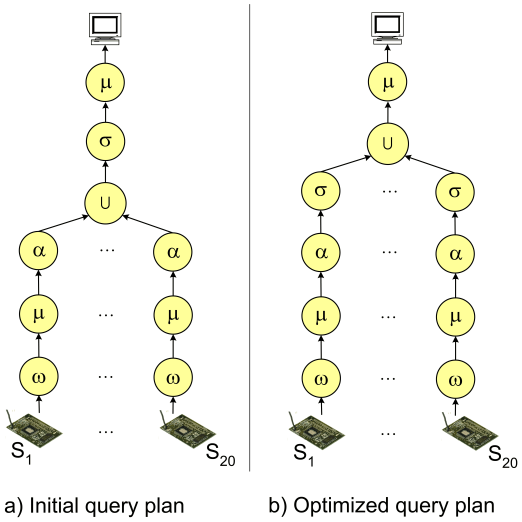

Fig. 2. Query plans composed of our operations

There are other approaches [Arasu et al. 2003b; Hammad et al. 2003] that are based on a quite similar temporal semantics. But they substantially differ in their implementation as they employ so-called positive-negative elements. This however has certain drawbacks as outlined in the following. When positive-negative elements are used, a window operator is required that explicitly controls element expiration. If a data source emits a new element, the window operator tags this element with '+' and sends it to the next operators afterwards. According to the window, the window operator buffers each incoming element until it expires. Then, it is assigned with a negative tag, i. e. a '-', and subsequently transferred to the next operators in the query plan. This implies that operators have to distinguish between positive and negative incoming elements. Furthermore, this approach doubles the number of elements being processed, since for each stream element in a raw input stream, two stream elements in a physical input stream are generated. These deficiencies are entirely avoided in our interval-based approach.

In the following, we describe how we transform a logical stream into a physical stream. This makes our transformations complete as a logical stream can be transformed into a physical one and vice versa (see Section 3.1.1). This fact is important for query optimization because it offers a seamless switching between logical and physical query plans.

4.1 Transformation: Logical to Physical Stream

Let $S^l \in \mathbb{S}^l$ be a logical stream. We transform a logical stream into a physical stream by two steps:

(1) We introduce time intervals by mapping each logical stream element $(e, t, n) \in S^l$ to a triple $(e, [t, t+1), n) \in \Omega \times \mathbb{I} \times \mathbb{N}$. This does not effect our semantics at all, since the time interval $[t, t+1)$ solely covers a single point in time, namely t. We denote this operation by $\iota : \mathbb{S}^l \rightarrow \wp(\Omega \times \mathbb{I} \times \mathbb{N})$.

(2) Then, we merge value-equivalent elements with adjacent time intervals in order to build larger time intervals. This operation termed *Coalesce* is commonly used in temporal databases [Slivinskas et al. 2000].
Let M, M' be in $\wp(\Omega \times \mathbb{I} \times \mathbb{N})$. We define a relation $M \triangleright M'$ that indicates if M can be coalesced to M' with:

$$\begin{gathered} M \triangleright M' :\Leftrightarrow (\exists\ m := (e, [t_S, t_E), n) \in M, \tilde{m} := (\tilde{e}, [\tilde{t}_S, \tilde{t}_E), \tilde{n}) \in M. \\ e = \tilde{e}\ \wedge\ t_E = \tilde{t}_S\ \wedge \\ (\exists\ M'' \in \wp(\Omega \times \mathbb{I} \times \mathbb{N}).\ M'' = (M - \{m, \tilde{m}\})\ \cup \{(e, [t_S, \tilde{t}_E), 1)\} \\ \wedge\ M'' \triangleright M'))\ \vee\ M = M' \\ \text{where } M - \{m, \tilde{m}\} := (M\ \setminus\ \{m, \tilde{m}\}) \cup \\ (\{(e, [t_S, t_E), n-1), (\tilde{e}, [\tilde{t}_S, \tilde{t}_E), \tilde{n}-1)\}\ \setminus\ \Omega \times \mathbb{I} \times \{0\}). \end{gathered}$$

When coalesce merges two triples in M, these elements are removed from M and the new triple containing the merged time intervals is inserted. Furthermore, the multiplicities have to be adopted.
This definition of coalescing is non-deterministic if M contains several elements whose start timestamp matches with the end timestamp of an other value-equivalent element. Therefore, coalescing $\zeta : \wp(\Omega \times \mathbb{I} \times \mathbb{N}) \to \wp(\Omega \times \mathbb{I} \times \mathbb{N})$ produces a set.

$$\begin{gathered} \zeta(M) := \{M' \in \wp(\Omega \times \mathbb{I} \times \mathbb{N}) \mid M \triangleright M' \wedge \\ (\forall\ M'' \in \wp(\Omega \times \mathbb{I} \times \mathbb{N}).\ (M' \not\triangleright M'')\ \vee\ (M' = M''))\} \end{gathered} \tag{16}$$

For a given logical stream S^l, we obtain a corresponding physical stream $S^p \in \mathbb{S}^p$ by ordering the elements of a multiset $M \in \zeta(\iota(S^l))$ according to $\leq_t$ while listing the duplicates as separate stream elements. As we will see in Section 5.2, our notion of stream equivalence is independent from the set chosen from $\zeta(\iota(S^l))$.

The schema of the physical stream can be derived from the logical stream by keeping the record schema and decorating it with the common temporal schema of a physical stream, namely the start and end timestamp attributes.

4.2 Operator Properties

For each operation of the logical algebra, PIPES provides at least one implementation based on physical streams, i. e., a physical operator takes one or multiple physical streams as input and produces one or multiple physical streams 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. This implies that a physical operator has to process the incoming elements directly without choosing the input from which the next element should be consumed.

Another important requirement for the implementation of physical operators over data streams is that these operators must be non-blocking. This is due to the potentially infinite length of the input streams and the request for early results. Our physical operator algebra meets this requirement by employing time intervals and introducing the window operator. This technique unblocks blocking operators, e. g. the difference, while guaranteeing deterministic semantics.

4.2.1 *Operator Classification.* The operators of our physical operator algebra can be classified in two categories:

—*Stateless operators:* A stateless operator is able to produce its results immediately without accessing any kind of internal data structure. Typical stateless operators are: filter, map, group and window. For instance, the filter operation evaluates a user-defined predicate for each incoming element. If the filter predicate is satisfied, the element is appended to the output stream, otherwise it is dropped. Another example is the group operation that invokes a grouping function on each incoming element. The result determines the physical output stream to which the element is appended to.

The implementation of stateless operators is straightforward and fulfills the requirements of data-driven query processing.

—*Stateful operators:* A stateful operator requires some kind of internal data structure for maintaining its state. Such a data structure has to support operations for efficient insertion, retrieval and reorganization. We identify the following physical operators in our algebra as stateful: Cartesian product/join, duplicate elimination, difference, union and aggregation.

The implementation of a stateful operator has to guarantee the ordering of physical streams (see Section 4.2.2). Moreover, it should be non-blocking while limiting memory usage. Most importantly, the implementation should produce deterministic results (see Section 4.2.3).

4.2.2 *Ordering Invariant.* A physical operator has to ensure that each of its physical output streams is ordered by $\leq_t$ (see Section 2.4), i. e., the stream elements in an output stream have to be in an ascending order, lexicographically by their start and end timestamps. This invariant of our implementation is assumed to hold for all input as well as output streams of a physical operator. This may cause delays in the result production of an operator. In a union for instance, the results have to be ordered, e. g. by maintaining an internal heap. This also explains why we consider the union operation to be stateful.

This ordering invariant seems to be very expensive to satisfy. However, it is commonly assumed in stream processing that raw input streams arrive temporally ordered [Babcock et al. 2002; Golab and Özsu 2003] or mechanisms exist that ensure such a temporal ordering [Srivastava and Widom 2004]. Besides, our efficient algorithms rely on this ordering invariant for the reorganization of the internal data structures of stateful operators.

4.2.3 *Reorganization.* Local reorganizations are necessary to restrict the memory usage of stateful physical operators. Such reorganizations are input-triggered, i. e., each time a physical operator processes an incoming element, a reorganization step is performed. In this reorganization step, all expired elements are removed from the internal data structures.

Let $S_1^p, \ldots, S_n^p \in \mathbb{S}^p$ be physical streams, $n \in \mathbb{N}$. For an arbitrary stateful operator with physical input streams $S_1^p, \ldots, S_n^p$, the reorganization is performed as follows: We store the start timestamps t_{S_j} for $j \in \{1, \ldots, n\}$ of the last incoming element of each physical input stream S_j^p. Then, all elements $(e, [t_S, t_E))$ can be safely removed from the internal data structures whose end timestamp t_E is smaller than $min\{t_{S_j} \mid j \in \{1, \ldots, n\}\}$ or equal. This condition ensures that only expired elements are removed from internal data structures. The correctness results from

the ordering invariant because if a new element $(\hat{e}, [\hat{t}_S, \hat{t}_E))$ of an input stream S_j^p arrives, all other elements of this stream processed before must have had a start timestamp that is equal or smaller than $\hat{t}_S$. Furthermore, a result of a stateful operator is only produced when the time intervals of the involved elements overlap. Let us consider a binary join, for example. Two stream elements $(e, [t_S, t_E)) \in S_1^p$ and $(\tilde{e}, [\tilde{t}_S, \tilde{t}_E)) \in S_2^p$ qualify if the join predicate holds for their records, e and $\tilde{e}$, and the time intervals, $[t_S, t_E)$ and $[\tilde{t}_S, \tilde{t}_E)$ overlap. The result contains the concatenation of the records and the intersection of the time intervals (see Definition 3.2.3). Hence, the reorganization condition specified above solely allows to remove an element from an internal data structure if it is guaranteed that there will be no future stream elements whose time interval will overlap with this element.

From this top-level point of view, it seems to be sufficient to require that a physical stream is in ascending order by the start timestamps of its elements. This is because the reorganization condition does not make use of the secondary order by end timestamps. However, we maintain the lexicographical order $\leq_t$ of physical streams since this generally leads to earlier results during reorganization. The reason is that the reorganization phase, which follows the internal linkage, can be stopped if an element is accessed whose end timestamp is larger than $min\{t_{S_j} \mid j \in \{1, \ldots, n\}\}$.

We made an interesting observation during our implementation work. When operations get unblocked by using windows, many stateful physical operators produce their results during the reorganization phase. Hence, expired elements are not only removed from the internal data structures but they are also appended to the physical output stream.

Input-triggered reorganization is only feasible if each physical input stream continuously delivers elements, which is a general assumption in stream processing. However, if one input stream totally fails, the minimum of all start timestamps $min\{t_{S_j} \mid j \in \{1, \ldots, n\}\}$ cannot be computed and, consequently, no elements can be removed from internal data structures. This kind of blocking has to be avoided, e. g. by introducing appropriate timeouts [Srivastava and Widom 2004]. A similar problem arises if the delays between subsequent elements within a physical stream are relatively long. In this case, the reorganization phase is seldom triggered, which may lead to an increased memory usage of the internal data structures. Hence, there is a latency-memory tradeoff for stateful operators.

4.2.4 *Coalesce and Split.* The *coalesce* operator merges value-equivalent stream elements with adjacent time intervals, while the *split* operator inverts this operation by splitting a stream element into several value-equivalent elements with adjacent time intervals. Note that both operations have no impact on the semantics of a query, since the records are valid at the same points in time and their multiplicities remain unchanged.

Both operators can effectively be used to control stream rates as well as element expiration adaptively. The latter has direct impact on the memory usage of internal data structures of stateful operators (see Section 4.2.3). Furthermore, earlier element expiration leads to earlier results because most stateful operators produce their results during the reorganization phase. Consequently, the coalesce and split operators can be used for physical optimization purposes. Coalesce generally de-

creases stream rates at the expense of a delayed element expiration in internal data structures. In contrast, split usually leads to earlier results and a reduced memory consumption in internal data structures at the expense of higher stream rates, which may cause an increase in the size of intermediate buffers. Hence, coalesce and split offer a way to adaptively control the tradeoff between scheduling costs and memory usage.

These operators are novel in the stream context and their effect on the runtime behavior of a DSMS will be investigated more detailed in our ongoing work.

4.3 Operator Implementation

This section reveals the implementation concepts of our physical operator algebra. Based on our time-interval approach, we present a purely data-driven implementation for each logical operation defined in Section 3.2.

4.3.1 *SweepArea.* A SweepArea (SA) is a dynamic data structure that is used to store the state of a stateful operator. Due to the fact that the reorganization of a stateful operator is time-based in general, we leverage the sweepline paradigm [Nievergelt and Preparata 1982], previously used in [Dittrich et al. 2002], to detect and remove expired elements efficiently. For that reason, all elements within a SweepArea are linked according to $\leq_t$. Furthermore, a SweepArea may perform additional reorganization steps as long as these do not conflict with our semantics. For instance, this might be exploited in the case of join operations when certain stream constraints ensure that some elements will not match in future.

Our algorithms use the following methods to manage a SweepArea SA:

—Procedure INSERT(*SweepArea SA, element s*)

—Procedure REMOVE(*SweepArea SA, element s*)

—Iterator QUERY(*SweepArea SA, element s*)

—Procedure REORGANIZE(*SweepArea SA, element s, stream* S_{out}).

Probing and reorganizing a SweepArea is controlled by two binary predicates:

—Predicate p_{query}(*element s*, *element* $\hat{s}$)

—Predicate p_{remove}(*element s*, *element* $\hat{s}$).

These predicates have to be appropriately defined for each stateful physical operation. Before we step into algorithmic details, we want to sketch the implementation of these methods from a top-level point of view.

Procedure INSERT(*SweepArea SA, element s*)

1 Add element s to SA;
2 Adjust linkage according to $\leq_t$;

The method INSERT adds a given element s to the SweepArea, while REMOVE deletes it. Both methods have to adjust the internal linkage according to $\leq_t$ in order to guarantee the correctness of our algorithms, in particular the ordering invariant. The costs for insertion and removal rely on the implementation of a SweepArea.

Depending on the algorithm and its underlying data structures used to implement an operator, the maintenance of the internal linkage causes additional costs per element, either constant or logarithmic in the size of the SweepArea.

Procedure REMOVE(*SweepArea SA, element s*)

```
1 Remove element s from SA;
2 Adjust linkage according to ≤_t;
```

The function QUERY is used to probe the SweepArea with a search element s. It delivers an iterator [Graefe 1993] over all elements $\hat{s}$ of the SweepArea that match the predicate p_{query} applied to s and $\hat{s}$. In analogy to insertion and removal, the costs for retrieval also depend on the specific implementation of a SweepArea.

Function QUERY(*SweepArea SA, element s*)

```
1 return all ŝ ∈ SA where p_query(s, ŝ) holds;
```

The method REORGANIZE performs the reorganization of a SweepArea. We distinguish between two different modes for reorganization indicated by the parameter S_{out}. Both modes have in common that they purge expired elements from a SweepArea. We consider an element $\hat{s}$ of the SweepArea to be *expired* if the predicate p_{remove} applied to the arguments s and $\hat{s}$ evaluates to *true*. REORGANIZE traverses the internal linkage of the SweepArea as long as p_{remove} holds. If an output stream S_{out} is specified, REORGANIZE appends each expired element $\hat{s}$ to S_{out} prior to its removal (see line 4). Otherwise, $\perp$ signals that no output stream is set. In this case, the expired elements are discarded directly. The latter also enables an efficient usage of bulk deletion techniques whenever it is not important to know which elements were expired.

Procedure REORGANIZE(*SweepArea SA, element s, stream S_{out}*)

```
1 while SA ≠ ∅ do
2 |   Element ŝ ← next element of SA according to ≤_t;
3 |   if p_remove(s, ŝ) then
4 |   |   if S_out ≠ ⊥ then ŝ ↪ S_out ;
5 |   |   REMOVE(SA, ŝ);
6 |   else return ;
```

If the implementation of a SweepArea guarantees a removal of the smallest element with respect to $\leq_t$ in $O(1)$ time on average, the amortized time for a REORGANIZE operation is $O(1)$. This could be achieved for instance in an equijoin or a duplicate elimination if the SweepAreas are implemented hash-based with an additional linkage according to $\leq_t$. During reorganization elements are removed from the SweepArea as long as p_{remove} holds by following the internal linkage. Hence,

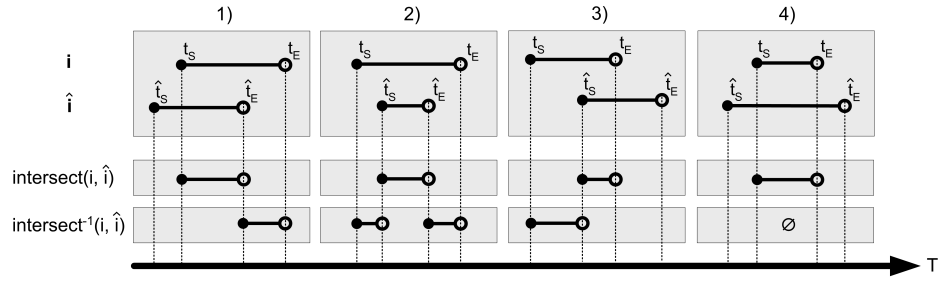

Fig. 3. Interval functions *intersect* and $intersect^{-1}$

accessing the next expired element has constant costs. If we assume the removal of an element from the hash table to have constant costs on average, REORGANIZE performs in amortized constant time.

Note that all methods defined above abstract from a particular implementation of a SweepArea with respect to its underlying data structure, e. g. a list or a hash table. This allows us to present the physical operators in a generic manner that widens their applicability. For instance in the case of a join, a hash-based implementation of the SweepAreas is suitable for equi-joins, but it is not suitable for similarity and spatial joins. This means that depending on the characteristics of an operation, an appropriate implementation of the SweepAreas has to be chosen in order to gain an efficient physical operator. Hence, the complexity of the following algorithms, which is not the focus of this paper, strongly depends on the specific implementation of the SweepAreas.

4.3.2 *Data-Driven Algorithms.* Before we start to present our data-driven algorithms, we define some basic predicates and functions. We also introduce our notation.

The predicate $overlaps : \mathbb{I} \times \mathbb{I} \rightarrow \{true, false\}$ determines if two time intervals overlap and is defined as follows:

$$overlaps([t_S, t_E), [\hat{t}_S, \hat{t}_E)) := \begin{cases} true & , (t_S < \hat{t}_E) \ \wedge \ (t_E > \hat{t}_S) \\ false & , \text{otherwise} \end{cases} .$$

Let $intersect : \mathbb{I} \times \mathbb{I} \rightarrow (\mathbb{I} \ \cup \ \emptyset)$ be the function that computes the intersection of two time intervals $i := [t_S, t_E)$ and $\hat{i} := [\hat{t}_S, \hat{t}_E)$ with

$$intersect(i, \hat{i}) := \begin{cases} [max\{t_S, \hat{t}_S\}, min\{t_E, \hat{t}_E\}) & , \text{if overlaps}(t, \hat{t}) \\ \emptyset & , \text{otherwise} \end{cases} .$$

Let $intersect^{-1} : \mathbb{I} \times \mathbb{I} \rightarrow \wp(\mathbb{I})$ be the function that computes all subintervals of maximum size of its first argument that are not overlapped by the second one. Two arbitrary time intervals $i := [t_S, t_E)$ and $\hat{i} := [\hat{t}_S, \hat{t}_E)$ are mapped as follows:

$$intersect^{-1}(i, \hat{i}) := \begin{cases} \emptyset & , \text{if } \neg overlaps(t, \hat{t}) \vee \\ & \quad ((\hat{t}_S \leq t_S) \ \wedge \ (\hat{t}_E \geq t_E)) \\ \{[\hat{t}_E, t_E)\} & , \text{if } (\hat{t}_S \leq t_S) \ \wedge \ (\hat{t}_E < t_E) \\ \{[t_S, \hat{t}_S)\} & , \text{if } (\hat{t}_S > t_S) \ \wedge \ (t_E \leq \hat{t}_E) \\ \{[t_S, \hat{t}_S), [\hat{t}_E, t_E)\} & , \text{if } (\hat{t}_S > t_S) \ \wedge \ (t_E > \hat{t}_E) \end{cases} .$$

Figure 3 illustrates the results of applying the functions $intersect$ and $intersect^{-1}$ to two overlapping time intervals.

Notation. The subsequent algorithms rely on the following notation: $s \hookleftarrow S_{in}$ indicates that an element s comes from the input stream S_{in}. This statement is typically used within a `for each` loop which means that a specified action will be performed for each incoming element s of an input stream S_{in}. $s \hookrightarrow S_{out}$ denotes that an element s is appended to the output stream S_{out}. Let $\oslash$ be the empty stream.

Algorithm 1: Filter

Input : stream S_{in}, filter predicate p
Output: stream S_{out}

```
1 S_out ← ∅;
2 foreach s ↩ S_in do
3     if p(s) then
4         s ↪ S_out;
```

4.3.2.1 *Filter*

Description. The filter operator (see Algorithm 1) evaluates an unary user-defined predicate p for each incoming element s of an input stream S_{in}. If the predicate holds, s is appended to the output stream S_{out}; otherwise it is discarded.

Correctness. The filter operator does not affect the stream order $\leq_t$ because it drops all stream elements where p evaluates to $false$.

Algorithm 2: Map

Input : stream S_{in}, mapping function f
Output: stream S_{out}

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

4.3.2.2 *Map*

Description. The map operator (see Algorithm 2) invokes an unary mapping function f to the record component of each incoming stream element s of the input stream S_{in}. The associated time interval $[t_S, t_E)$ remains unchanged. A new element $(f(e), [t_S, t_E))$ is created and appended to the output stream S_{out}.

Correctness. The record component of each element is set to the return value of the user-defined mapping function. Since the time intervals are not modified, the elements of S_{out} appear in the same order as in S_{in}.

Algorithm 3: Group

Input : stream S_{in}, grouping function f
Output: streams $S_{out_1}, \ldots, S_{out_k}$

```
1 S_out1 ← ∅; ...; S_outk ← ∅;
2 foreach s ↩ S_in do
3     s ↪ S_out_f(s);
```

4.3.2.3 *Group*

Description. The group operation (see Algorithm 3) splits an input stream S_{in} into k disjoint output streams $S_{out_1}, \ldots, S_{out_k}$ where k depends on the unary grouping function f. This function determines for each incoming element s the corresponding output stream, namely $S_{out_{f(s)}}$.

Correctness. The elements of S_{in} are processed in a FIFO fashion and distributed among a set of output streams according to the grouping function f. Thus, the time interval component of an element is not affected. Consequently, each of the output streams is ordered correctly, because we assume an ordering of S_{in} by $\leq_t$.

Algorithm 4: Window

Input : stream S_{in}, window size w, boolean *sliding*
Output: stream S_{out}

```
1  S_out ← ∅;
2  if sliding then
3      foreach s := (e, [t_S, ∞)) ↩ S_in do
4          (e, [t_S, t_S + w)) ↪ S_out;
5  else
6      t̃_S, t̃_E ∈ T;
7      foreach s := (e, [t_S, ∞)) ↩ S_in do
8          t̃_E ← w;
9          while t_S ≥ t̃_E do
10             t̃_S ← t̃_E;
11             t̃_E ← t̃_S + w;
12         (e, [t_S, t̃_E)) ↪ S_out;
```

4.3.2.4 *Window*

Description. Algorithm 4 differs between *sliding* and *fixed* windows specified by the flag *sliding*. If this flag is *true*, the window operator sets the end timestamp of each incoming stream element $(e, [t_S, \infty))$ to $t_S + w$. As a consequence, a sliding window semantics is modeled due to the validity of w time units for each record.

In the case that the flag *sliding* is $false$, the end timestamp of each incoming element is set to $\tilde{t}$ where $\tilde{t}$ denotes the closest multiple of w that is larger than t_S. This parameterization of the window operator results in a fixed window semantics.

Correctness. The algorithm is a straightforward implementation of the window concept introduced in Section 2.6, and thus defines the validity of a record accordingly. In both cases, the window operator sets the end timestamps with respect to the start timestamps. Therefore, the ordering $\leq_t$ of the input stream S_{in} is preserved in the output stream S_{out}.

Algorithm 5: Cartesian Product / Join

Input : streams S_{in_1}, S_{in_2}, join predicate θ, SweepAreas SA_1, SA_2, Min-Heap H
Output: stream S_{out}

```
1  S_out ← ∅;
2  t_S1, t_S2, min_ts ∈ T ∪ {⊥}; t_S1 ← ⊥; t_S2 ← ⊥; min_ts ← ⊥;
3  k ∈ {1, 2};
4  foreach s := (e, [t_S, t_E)) ↩ S_in_j, j ∈ {1, 2} do
5      k = j mod 2 + 1;
6      REORGANIZE(SA_k, s, ⊥);
7      INSERT(SA_j, s);
8      Iterator qualifies ← QUERY(SA_k, s);
9      foreach (ê, [t̂_S, t̂_E)) ∈ qualifies do
10         Insert (e ∘ ê, intersect([t_S, t_E), [t̂_S, t̂_E))) into H;
11     t_Sj ← t_S;
12     min_ts ← min(t_S1, t_S2);
13     if min_ts ≠ ⊥ then
14         while H ≠ ∅ do
15             Element top := (ẽ, [t̃_S, t̃_E)) ← top element of H;
16             if t̃_S < min_ts then
17                 top ↪ S_out;
18                 Remove top from H;
19             else return ;
```

4.3.2.5 *Cartesian Product / Join.* For the presentation of our algorithms, we chose a different order as in Section 3.2 because we first introduced the stateless ones and now proceed with the stateful operations. As already mentioned, stateful operations rely on SweepAreas that are parametrized by two binary predicates p_{query} and p_{remove}. Therefore, theses predicates have to be appropriately defined for each stateful operator.

Our purely data-driven algorithms permit that stream elements of multiple input streams might arrive at an operator concurrently. Since we do not focus on synchronization details and buffering in this paper, we assume that the processing

of a single stream element is atomic and refer the interested reader to [Cammert et al. 2003].

Description. Algorithm 5 transfers the ripple join technique [Haas and Hellerstein 1999] towards data-driven query processing. Whenever a new element s arrives from input stream S_{in_j}, the opposite SweepArea SA_k is reorganized at first (see line 6). The necessary reorganization predicate $p_{remove} : (\Omega \times \mathbb{I}) \times (\Omega \times \mathbb{I}) \rightarrow \{true, false\}$ is defined as

$$p_{remove}((e, [t_S, t_E)), (\hat{e}, [\hat{t}_S, \hat{t}_E))) := \begin{cases} true & , t_S \geq \hat{t}_E \\ false & , \text{otherwise} \end{cases} .$$

REORGANIZE uses the internal linkage of SA_k and removes all elements as long as their end timestamp is smaller than the start timestamp of the element s passed to the reorganization call.

In the next step, s is inserted into SA_j. At this, the internal linkage of the SweepAreas can be maintained easily as the elements arrive in the right order so that no reordering is necessary. After the insertion, SA_k is queried with s in line 8. For that purpose, the query predicate $p_{query} : (\Omega \times \mathbb{I}) \times (\Omega \times \mathbb{I}) \rightarrow \{true, false\}$ is composed of two binary predicates, namely a user-defined join predicate θ and a predicate determining the overlap of the associated time intervals:

$$p_{query}((e, i), (\hat{e}, \hat{i})) := \begin{cases} true & , \theta((e, i), (\hat{e}, \hat{i})) \ \wedge \ overlaps(i, \hat{i}) \\ false & , \text{otherwise} \end{cases} .$$

Our concept of SweepAreas parametrized by flexible query predicates offers to implement various join types ranging from simple equijoins to similarity joins. If θ is specified in a way that it always evaluates to *true*, the *Cartesian Product* is computed. Note that even in this case the overlap condition has to hold. For an *equijoin*, the join predicate θ has to verify that $e = \hat{e}$.

The join results are built by iteratively concatenating e with the records $\hat{e}$ of the query result (see line 10). The associated time intervals correspond to the intersection of the two involved time intervals. All join results are inserted into a min-heap H that reorders the join results according to the order relation $\leq_t$. As long as the top element $\tilde{t}_S$ of H is smaller than the minimum timestamp of both inputs min_{t_S}, $\tilde{t}_S$ is appended to the output stream S_{out} and removed from the heap. Invoking the function *min* on (t_{S_1}, t_{S_2}) returns $\perp$ as long as one of its arguments is undefined, i. e. $t_{S_i} = \perp$ for $i \in \{1, 2\}$. In the case of finite input streams, the heap has to be emptied after processing the last element which enforces all elements being appended to S_{out}.

Correctness. The SweepAreas of the join contain all elements that have a chance to qualify as a result. This property is ensured by inserting each incoming element, and not violated by the reorganization which purges only expired elements. The predicate p_{remove} guarantees that no element is discarded which might possibly overlap with an incoming element in future. Due to the correctness of the underlying ripple join technique, our algorithm produces sound join results for a variety of joins depending on the user-defined join predicate θ. In accordance with our logical operator algebra (see Section 3.2.3), all types of joins have in common that in addition to the join predicate θ, two elements solely qualify if their time intervals overlap.

The order of the output stream is correct since the join results are re-arranged by a min-heap. Furthermore, their release is controlled by the condition that the start timestamp of the heap's top element $\tilde{t}_S$ has to be smaller than the minimum start timestamp min_{t_S} of both inputs seen so far. This is necessary to ensure that no element is appended to S_{out} too early, which otherwise might violate the ordering invariant.

Algorithm 6: Duplicate Elimination

Input : stream S_{in}, value-equivalence predicate θ, SweepArea SA
Output: stream S_{out}

```
1 S_out ← ∅;
2 foreach s := (e, [t_S, t_E)) ↩ S_in do
3     REORGANIZE(SA, s, S_out);
4     Iterator qualifies ← QUERY(SA, s);
5     if qualifies = ∅ then INSERT(SA, s);
6     else
7         T t̃_E ← maximum end timestamp in qualifies ;
8         if t̃_E < t_E then INSERT(SA, (e, [t̃_E, t_E)));
```

4.3.2.6 *Duplicate Elimination*

Description. Algorithm 6 performs a duplicate elimination. Each time a new stream element arrives, a reorganization phase is triggered by calling REORGANIZE (see line 3). As a consequence, all expired elements are purged from the SweepArea and appended to the output stream S_{out}. Thus, this operator produces its results during the reorganization phase (see Section 4.2.3). The predicate p_{remove} corresponds to that of the join. For finite input streams, all elements of the SweepArea can be appended to S_{out} and discarded immediately after the last element was processed.

After the reorganization phase the SweepArea is probed for duplicates. The method QUERY returns all elements $\hat{s}$ of the SweepArea where the predicate p_{query} evaluates to $true$. The predicate p_{query} is defined analogously to that of the join algorithm. For the duplicate elimination, the join predicate θ is replaced by the given value-equivalence predicate θ which verifies if its first argument is a duplicate of its second argument. The overlap condition in p_{query} has still to be checked. Generally, θ is a map $(\Omega \times \mathbb{I}) \times (\Omega \times \mathbb{I}) \rightarrow \{true, false\}$ where

$$\theta((e,i),(\hat{e},\hat{i})) := \begin{cases} true & , e = \hat{e} \\ false & , \text{otherwise} \end{cases} .$$

Since θ serves as a parameter, this kind of algorithm is not limited to a duplicate elimination based on exact duplicates. It also permits to identify elements as duplicates by matching selected record attributes or other similarity measures. If no duplicate exists, i. e. the iterator $qualifies$ delivers no element, s is inserted into the SweepArea. Otherwise, the maximum end timestamp $\tilde{t}_E$ in $qualifies$ is determined (see line 7). This can either be done by sequentially scanning the iterator

or by a specific implementation of the QUERY method, e. g. delivering the qualifying elements in descending order by their end timestamps. If the maximum end timestamp $\tilde{t}_E$ is smaller than the end timestamp t_E of the incoming element s, the validity of s exceeds the validity of all matching elements delivered by $qualifies$. For that reason, a new element $(e, [\tilde{t}_E, t_E))$ is inserted into the SweepArea to handle the validity extension. In contrast to the join, INSERT has additional costs logarithmic in the size of the SweepArea to maintain the internal linkage.

Correctness. At every time instant the operator is in a correct state due to the following two reasons. First, REORGANIZE works analogously to the join algorithm except that the expired elements are appended to the output stream prior to their removal. The order of the output stream is retained due to the implementation of the REORGANIZE method as well as the link-sensitive insertion. Second, calling QUERY determines all elements of the SweepArea that might affect the state and thus the result. These elements have to be value-equivalent to s, which is checked by θ, and the associated time interval has to overlap with $[t_S, t_E)$. Due to the order of S_{in}, $[t_S, t_E)$ is only relevant to the operator state if t_E is larger than the maximum end timestamp $\tilde{t}_E$ in $qualifies$. In this case, INSERT adds the element $(e, [\tilde{t}_E, t_E))$ to the SweepArea to cover the validity extension.

4.3.2.7 *Difference*

Description. The difference operator (see Algorithm 7) removes all elements occurring in S_{in_2} from S_{in_1}. The implementation is based on two SweepAreas SA_1 and SA_2. Internally, the temporal symmetric difference over multisets, $S_{in_1} - S_{in_2}$ and $S_{in_2} - S_{in_1}$, is computed. The resultant elements of $S_{in_1} - S_{in_2}$ are appended to S_{out}, whereas the computation of $S_{in_2} - S_{in_1}$ is a requisite for the correctness. Not only the symmetric processing is similar to the join, but also the same query and remove predicates are utilized. Whenever an element $s := (e, [t_S, t_E))$ arrives from S_{in_j}, QUERY probes the opposite SweepArea SA_k with p_{query}. The returned iterator $qualifies$ delivers all elements of SA_k ordered by $\leq_t$ that are value-equivalent to s according to θ and overlap with s with respect to their time intervals.

If $qualifies$ is empty, s is inserted into SA_j. Otherwise, we have to subtract s from $qualifies$, and vice versa, by adjusting the involved time intervals. For that purpose, we defined the function $intersect^{-1}$ at the beginning of Section 4.3.2. For the time interval given as first argument, this function computes the minimum number of subintervals that do not overlap with the time interval specified as second argument. The *while*-loop starting in line 12 uses a temporary list $insert_{SA_j}$ initialized with s to store the fragments of s that still might have a temporal overlap with an element in $qualifies$. For that reason, the elements in $qualifies$ are successively probed for an overlap with an element in $insert_{SA_j}$. If an overlap exists, the difference of the elements $\tilde{s} \in insert_{SA_j}$ and $\hat{s} \in qualifies$ is computed in terms of time intervals, i. e., $intersect^{-1}$ is invoked on $(\tilde{i}, \hat{i})$ which subtracts $\hat{i}$ from $\tilde{i}$. The difference algorithm applies $intersect^{-1}$ twice while swapping the arguments. Consequently, the symmetric difference is computed and new elements are created that do not contain the overlapping part. Prior to the insertion of these new elements into the corresponding SweepAreas, $\tilde{s}$ is removed from $insert_{SA_j}$ and $\hat{s}$ is removed

Algorithm 7: Difference

Input : streams S_{in_1}, S_{in_2}, value-equivalence predicate θ, SweepAreas SA_1, SA_2
Output: stream S_{out}

```
1  S_out ← ∅;
2  t_S1, t_S2, min_ts ∈ T ∪ {⊥}; t_S1 ← ⊥; t_S2 ← ⊥; min_ts ← ⊥;
3  k ∈ {1, 2};
4  foreach s := (e, [t_S, t_E)) ↩ S_in_j, j ∈ {1, 2} do
5      k = j mod 2 + 1;
6      t_Sj ← t_S;
7      Iterator qualifies ← QUERY(SA_k, s);
8      if qualifies = ∅ then INSERT(SA_j, s);
9      else
10         List insert_SAj ← ∅;
11         Append s to insert_SAj;
12         while insert_SAj ≠ ∅ ∧ qualifies ≠ ∅ do
13             Element ŝ := (ê, [t̂_S, t̂_E)) ← next element from qualifies;
14             Iterator overlaps ← all s̃ := (ẽ, [t̃_S, t̃_E)) from insert_SAj where
               overlaps([t̂_S, t̂_E), [t̃_S, t̃_E)) holds;
15             if overlaps ≠ ∅ then
16                 Element s̃ := (ẽ, [t̃_S, t̃_E)) ← first element from overlaps;
17                 Remove s̃ from insert_SAj;
18                 REMOVE(SA_k, ŝ);
19                 foreach s̄ := (ē, [t̄_S, t̄_E)) ∈ {ẽ} × intersect^-1(ĩ, î) do
20                     if t̄_E ≤ t̂_S then INSERT(SA_j, s̄);
21                     else Append s̄ to insert_SAj;
22                 foreach s̄ ∈ {ê} × intersect^-1(î, ĩ) do INSERT(SA_k, s̄);
23         foreach s̃ ∈ insert_SAj do INSERT(SA_j, s̃);
24     min_ts ← min(t_S1, t_S2);
25     if min_ts ≠ ⊥ then
26         if j = 1 then REORGANIZE(SA_2,(e, [min_ts, t_E)), ⊥);
27         else REORGANIZE(SA_1,(e, [min_ts, t_E)), S_out);
```

from SA_k (see line 18). The new elements $\{\tilde{e}\} \times intersect^{-1}(\tilde{i}, \hat{i})$ resulting from the subtraction of $\hat{s}$ from $\tilde{s}$ are a part of the necessary changes to the SweepArea SA_j. All elements $\bar{s}$ whose end timestamp $\bar{t}_E$ is equal to or smaller than $\hat{t}_S$ can directly be inserted into the SweepArea SA_j (see line 20) since an overlap with further elements in $qualifies$ is impossible due to the ordering of $qualifies$ according to $\leq_t$. The other elements have to be appended to $insert_{SA_j}$ for further checks. The resultant elements from subtracting $\tilde{s}$ from $\hat{s}$, namely $\{\hat{e}\} \times intersect^{-1}(\hat{i}, \tilde{i})$, are inserted into SA_k to update the SweepArea. After the termination of the *while*-loop, all elements of the list $insert_{SA_j}$ can be inserted to SA_j because it is ensured that no overlap with an element in $qualifies$ exists anymore. Note that this dif-

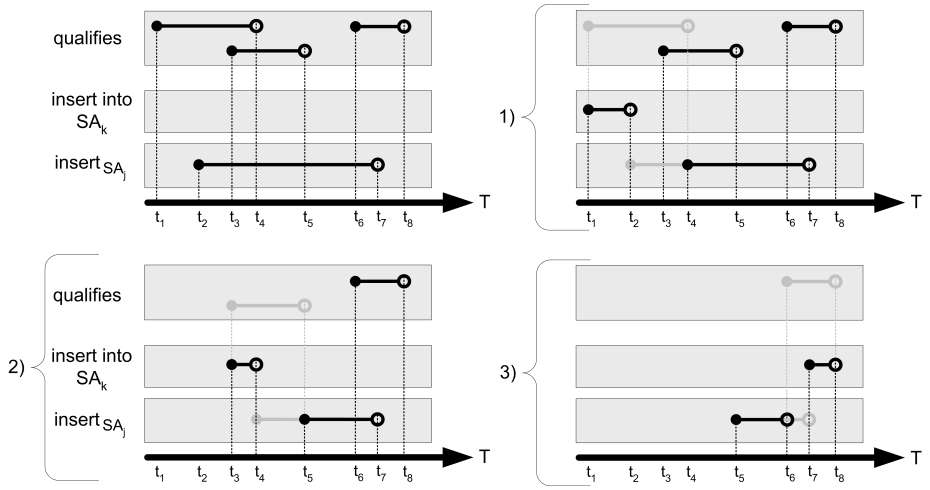

Fig. 4. Computing the difference on time intervals

ference operation complies with the difference defined in Subsection 3.2.5 and thus is sensitive to duplicates. Therefore, each overlap with $[t_S, t_E)$ is exactly removed once.

Finally, the opposite SweepArea SA_k is reorganized with the minimum start timestamp min_{t_S} of both inputs. Recall that the elements of the SweepAreas are not considered as part of the input anymore. The expired elements in SA_2 are just dropped by REORGANIZE without appending them to S_{out}, but the expired elements in SA_1 contribute to the result of the difference operator. For that reason, REORGANIZE is called with the argument S_{out} instead of $\perp$. This is contrary to the join algorithm that reorganizes both SweepAreas by calling REORGANIZE with argument $\perp$. Although it would be possible to perform the reorganization as first step of the algorithm, we decided to place it at the end since it is likely that more elements fulfill the predicate p_{remove} due to the modifications in the *while*-loop. In the case of finite input streams, which is not explicitly addressed, all elements of SA_1 can be appended to S_{out} after the last incoming element of both inputs was processed. Thereafter both SweepAreas can be discarded.

Figure 4 illustrates the stepwise progress of the *while*-loop by taking a closer look at the involved time intervals, the temporary list $insert_{SA_j}$ and the insertions into SA_k. The steps 1 to 3 show how the qualifying elements are subtracted from the incoming element s initially stored in the list $insert_{SA_j}$.

Correctness. In order to support a difference operator with control over duplicates, the algorithm is implemented symmetrically. This means that for each incoming element from S_{in_j} the opposite SweepArea SA_k is queried and reorganized. In accordance with our logical operator algebra, the difference refers to value-equivalent elements that are valid at the same point in time. Thus, QUERY returns all elements of SA_k that are value-equivalent to s and overlap with $[t_S, t_E)$. Computing the difference means to eliminate the overlap of $[t_S, t_E)$ with the time intervals in $qualifies$. This is iteratively achieved inside the *while*-loop by applying

the function $intersect^{-1}$ in a symmetric manner, i. e., the arguments of the function are swapped which removes the overlapping part from both involved time intervals. Thereby each overlap is removed exactly once, which is an important criteria for the correctness of a duplicate-sensitive difference operator. Based on the results of calling $intersect^{-1}$ twice, new elements are created to update the SweepAreas SA_j and SA_k appropriately. As a precondition for these insertions, the involved elements have to be deleted from the operator state before. Hence, $\tilde{s}$ is removed from $insert_{SA_j}$ and $\hat{s}$ from SA_k. The temporary list $insert_{SA_j}$ is used to store the remainder of the incoming element s during the progress of subtracting s from $qualifies$ and vice versa. The optimization in line 20 directly inserts those elements into the SweepArea SA_j for which an overlap with an element in $qualifies$ is not possible anymore, instead of appending them to $insert_{SA_j}$. As the difference is only applied for value-equivalent elements with overlapping time intervals, this optimization does not conflict with the correctness of the algorithm.

The reorganization works similar to the join algorithm except that REORGANIZE is called for SA_1 with the argument S_{out} instead of $\perp$. However, this only causes expired elements to be appended to the output stream prior to their removal. With regard to the correctness, it is important to reorganize with the minimum timestamp min_{t_S} of both input streams. This guarantees that no elements are released from the SweepAreas too early. The output stream S_{out} is ordered according to $\leq_t$ since REORGANIZE appends the elements by following the internal linkage of SA_1 that is maintained during insertion.

4.3.2.8 *Aggregation*

Description. Algorithm 8 illustrates the implementation of the aggregation operator based on a binary, user-defined aggregation function $f : (\Omega \times \mathbb{I}) \cup \{\perp\} \times (\Omega \times \mathbb{I}) \rightarrow \Omega$ that is applied successively to the current aggregate $\hat{s} := (\hat{e}, [\hat{t}_S, \hat{t}_E))$ and an incoming element $s := (e, [t_S, t_E))$. Hence, the aggregation values are computed iteratively as proposed in [Hellerstein et al. 1997]. This differs from the aggregation defined in our logical operator algebra but saves memory resources since only the current aggregates have to be managed in the SweepArea SA. An aggregate $\hat{s}$ consists of an aggregation value $\hat{e}$ and a time interval $[\hat{t}_S, \hat{t}_E)$ for which the aggregation value is valid.

To ensure the semantics of the logical aggregation operation, all elements $\hat{s}$ of SA have to be updated that overlap with s in terms of time intervals. Therefore, the QUERY method employs the following query predicate:

$$p_{query}((e, [t_S, t_E), (\hat{e}, [\hat{t}_S, \hat{t}_E))) := \begin{cases} true & , overlaps([t_S, t_E), [\hat{t}_S, \hat{t}_E)) \\ false & , \text{otherwise} \end{cases} .$$

For the case of an overlap, we remove the element $\hat{s}$ from SA (see line 8). Then, we split $[\hat{t}_S, \hat{t}_E)$ into maximum subintervals with either no or full overlap, while keeping the associated aggregation value for each of them. We adjust the aggregation value for the intersection by invoking f on $(\hat{s}, s)$. In addition, we have to create new aggregates for each maximum subinterval i of $[t_S, t_E)$ for which no overlap is found in SA. This new aggregate consists of an initialized aggregation value $f(\perp, s)$ and the time interval i. All aggregates are inserted into the SweepArea right after their creation.

Algorithm 8: Aggregation

Input : stream S_{in}, SweepArea SA, aggregation function f
Output: stream S_{out}

```
1  S_out ← ∅;
2  foreach s := (e, [t_S, t_E)) ↩ S_in do
3      Iterator qualifies ← QUERY(SA, s);
4      if qualifies = ∅ then INSERT(SA, (f(⊥, s), [t_S, t_E)));
5      else
6          T last_{t_E} ← t_S;
7          foreach ŝ := (ê, [t̂_S, t̂_E)) ∈ qualifies do
8              REMOVE(SA, ŝ);
9              if t̂_S < t_S then
10                 INSERT(SA, (ê, [t̂_S, t_S)));
11                 if t_E < t̂_E then
12                     INSERT(SA, (f(ŝ, s), intersect([t_S, t_E), [t̂_S, t̂_E))));
13                     INSERT(SA, (ê, [t_E, t̂_E)));
14                 else INSERT(SA, (f(ŝ, s), [t_S, t̂_E)));
15             else
16                 if last_{t_E} < t̂_S then INSERT(SA, (f(⊥, s), [last_{t_E}, t̂_S))) ;
17                 if [t̂_S, t̂_E) = intersect([t_S, t_E), [t̂_S, t̂_E)) then
18                     INSERT(SA, (f(ŝ, s), [t̂_S, t̂_E)));
19                 else
20                     INSERT(SA, (f(ŝ, s), [t̂_S, t_E)));
21                     INSERT(SA, (ê, [t_E, t̂_E)));
22             last_{t_E} ← t̂_E;
23         if last_{t_E} < t_E then INSERT(SA, (f(⊥, s), [last_{t_E}, t_E))) ;
24     REORGANIZE(SA, s, S_out);
```

Furthermore, it is important for the correctness of the algorithm that QUERY returns an iterator that is in ascending order by $\leq_t$. Since the aggregation has not to check for value-equivalence as other algorithms, the QUERY method can be implemented efficiently by an index structure supporting range queries over interval data, e. g. a priority search tree.

As each expired element contains the final aggregation value for the associated time interval, all expired elements are appended to the output stream S_{out} finally. REORGANIZE performs this step by using the same remove predicate p_{remove} as specified in the join algorithm, which releases all aggregates whose end timestamp is equal to or smaller than t_S by following the internal linkage. While constant removal costs are guaranteed by the iterator $qualifies$, inserting an element into SA may cause additional logarithmic costs in the size of the SweepArea to maintain the internal linkage. In analogy to the difference operator, REORGANIZE is performed at the end as it is likely that more aggregates can be released then. In the case of a

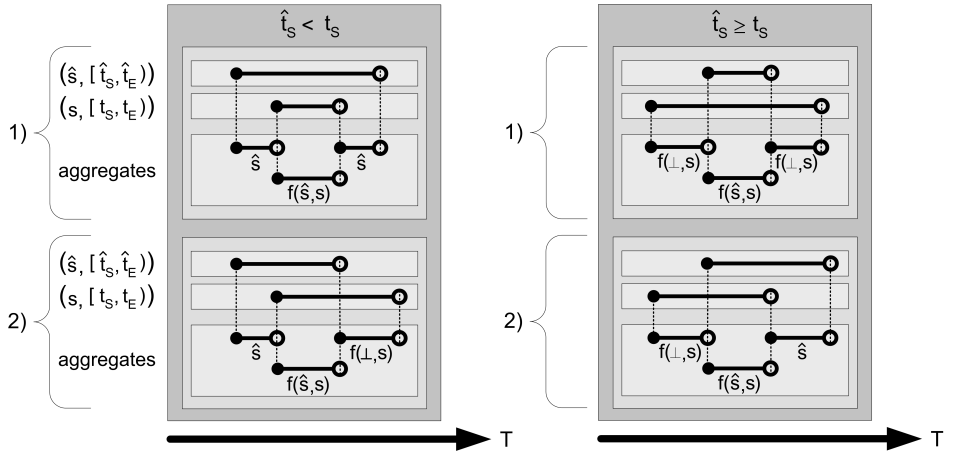

Fig. 5. Case differentiation for the incrementally computed aggregation

finite input stream S_{in}, all elements of SA can be appended to S_{out} and discarded after the last element was processed.

Correctness. The aggregation operator produces a correctly ordered output stream S_{out} since REORGANIZE traverses the internal linkage of the SweepArea that aligns the elements according to $\leq_t$ during insertion. Another invariant of this algorithm is that the SweepArea contains no duplicates. This results from the temporal segmentation of the elements in $qualifies$ into elements with either no or full overlap with $[t_S, t_E)$. It remains to show that we segment the involved time intervals correctly and compute the related aggregation values properly. Therefore, we distinguish between two top-level cases as sketched in Figure 5:

—Case $\hat{t}_S < t_S$: Then, $[t_S, t_E)$ is either inside of $[\hat{t}_S, \hat{t}_E)$ or exceeds it at the right border.

—Case $\hat{t}_S \geq t_S$: This case has to be considered as our algorithm may produce time intervals with a start timestamp larger than t_S. If $[\hat{t}_S, \hat{t}_E) = intersect([t_S, t_E), [\hat{t}_S, \hat{t}_E))$, $[t_S, t_E)$ contains $[\hat{t}_S, \hat{t}_E)$. Otherwise, $[t_S, t_E)$ only exceeds the left border $\hat{t}_S$.

Depending on these cases, we create disjoint time intervals by separating the overlapping sections from the unique sections of each time interval. Afterwards, appropriate aggregation values are assigned to these time intervals by considering the following three cases:

(1) The generated time interval is a subinterval of $[\hat{t}_S, \hat{t}_E)$ and does not overlap with $[t_S, t_E)$. In that case, the aggregation value $\hat{e}$ remains unchanged.
(2) The generated time interval is a subinterval of $[t_S, t_E)$ and does not overlap with $[\hat{t}_S, \hat{t}_E)$. Then, a new aggregation value is computed by invoking the aggregation function f on $(\perp, s)$.
(3) The generated time interval corresponds to the intersection of both time intervals. In this case we compute a new aggregation value by calling the aggregation function f for the old aggregate $\hat{s}$ and the new element s.

In order to clarify the role of the variable $last_{t_E}$ (see lines 16 and 23), we want

to point out that $[t_S, t_E)$ may overlap with multiple time intervals from $qualifies$. Hence it may be necessary to fill up possible intermediate gaps between two subsequent time intervals in $qualifies$. For that reason, $last_{t_E}$ stores the end timestamp of the last accessed element $\hat{s} \in qualifies$. This information is also required if $[t_S, t_E)$ overlaps with the last element in $qualifies$ and exceeds its right border, i. e. $t_E > \hat{t}_E$. Again a precondition for the correct usage of $last_{t_E}$ is that the iterator $qualifies$ delivers its elements according to $\leq_t$.

Example. Let us consider our running example, where we compute the average speeds of vehicles. The physical query plan is obtained by replacing all logical operations in the logical query plan (see Section 3.9) by their corresponding physical counterparts.

The listing below shows the elements within the status of the aggregation operator stopped before performing the reorganization phase triggered by the third incoming element for the example given in Section 2.6.

```
((18.280), [03/11/1993 05:00:08, 03/11/1993 05:01:32))
((19.805), [03/11/1993 05:01:32, 03/11/1993 05:02:16))
((19.766), [03/11/1993 05:02:16, 03/11/1993 05:15:08))
((20.510), [03/11/1993 05:15:08, 03/11/1993 05:16:32))
((19.690), [03/11/1993 05:16:32, 03/11/1993 05:17:16))
```

Because the start timestamp `05:02:16` of the third element is greater than the end timestamp `05:01:32`, the reorganization phase produces the first element of our listing as result and removes it from the status.

Algorithm 9: Union

Input : stream S_{in_1}, S_{in_2}, SweepArea SA
Output: stream S_{out}

```
1 S_out ← ∅;
2 t_S1, t_S2, min_ts ∈ T ∪ {⊥}; t_S1 ← ⊥; t_S2 ← ⊥; min_ts ← ⊥;
3 foreach s := (e, [t_S, t_E)) ↩ S_in_j, j ∈ {1, 2} do
4     t_Sj ← t_S;
5     min_ts ← min(t_S1, t_S2);
6     if min_ts ≠ ⊥ then REORGANIZE(SA, (e, [min_ts, t_E)), S_out);
7     INSERT(SA, s);
```

4.3.2.9 *Union*

Description. The union operator (see Algorithm 9) merges two input streams. At first, the minimum start timestamp min_{t_S} of the latest start timestamps t_{S_j} of both input streams S_{in_j} is determined. If min_{t_S} is initialized, i. e., it differs from $\perp$, it serves as argument to REORGANIZE where the remove predicate

$$p_{remove}((e, [t_S, t_E)), (\hat{e}, [\hat{t}_S, \hat{t}_E))) := \begin{cases} true & , t_S > \hat{t}_S \\ false & , \text{otherwise} \end{cases}$$

is evaluated. REORGANIZE consequently releases all elements $\hat{s} \in SA$ as output whose start timestamp is smaller than min_{t_S}. After reorganizing the SweepArea the stream element s is inserted into SA. If both inputs are finite, all elements of the SweepArea can be appended to S_{out} and purged after the last element was inserted.

For the generalized union operation with more than two input streams, the SweepArea can be implemented as a min-heap due to the order-sensitive insertion.

Correctness. The union operator is implemented statefully to ensure the ordering of the output stream S_{out} by merging the input streams appropriately. For that reason, INSERT reorders the incoming elements according to $\leq_t$ and REORGANIZE solely appends elements to S_{out} as long as their start timestamp is smaller than min_{t_S}. Combined with the ordering invariant of the input streams, this ensures that no future elements may violate the ordering of S_{out}.

Algorithm 10: Coalesce

Input : stream S_{in}, value-equivalence predicate θ, SweepArea SA, maximum time interval size ε
Output: stream S_{out}

```
1 S_out ← ∅;
2 foreach s := (e, [t_S, t_E)) ↩ S_in do
3     REORGANIZE(SA, s, S_out);
4     Iterator qualifies ← QUERY(SA, s);
5     if qualifies = ∅ then INSERT(SA, s);
6     else
7         Element ŝ := (ê, [t̂_S, t̂_E)) ← first element of qualifies;
8         REMOVE(SA, ŝ);
9         INSERT(SA, (e, [t̂_S, t_E)));
```

4.3.2.10 *Coalesce*

Description. Algorithm 10 merges value-equivalent elements with adjacent time intervals. In the first step, REORGANIZE appends all elements $(\hat{e}, [\hat{t}_S, \hat{t}_E))$ of SA to S_{out} that fulfill the remove predicate

$$p_{remove}((e, [t_S, t_E)), (\hat{e}, [\hat{t}_S, \hat{t}_E))) := \begin{cases} true & , (t_S > \hat{t}_E) \ \vee \ (\hat{t}_E - \hat{t}_S > \varepsilon) \\ false & , \text{otherwise} \end{cases}$$

and removes them afterwards. In contrast to the remove predicate specified for the Cartesian Product (see Paragraph 4.3.2.5), elements $\hat{s} \in SA$ where $t_S = \hat{t}_E$ remain in the SweepArea since these qualify for coalescing. The second part of the condition, $\hat{t}_E - \hat{t}_S > \varepsilon$, defines an upper bound for the size of time intervals produced during coalesce. Whenever an element $\hat{s}$ exceeds this bound, it is appended to the output stream and removed from SA, even if future elements might exist that would qualify for coalescing. This condition restricts the blocking behavior of the coalesce operation in cases when S_{in} might contain long sequences of value-equivalent elements.

In the next step of the algorithm, the SweepArea is probed for qualifying elements by calling QUERY with the following query predicate:

$$p_{query}((e,[t_S,t_E)),(\hat{e},[\hat{t}_S,\hat{t}_E))) := \begin{cases} true & ,\theta(e,\hat{e}) \ \wedge \ \hat{t}_E = t_S \\ false & ,\text{otherwise} \end{cases} .$$

This predicate clearly reflects when coalesce can be applied, namely if both records e and $\hat{e}$ are value-equivalent, which is determined by the user-defined predicate θ. Additionally, the time interval $[t_S, t_E)$ has to be adjacent to $[\hat{t}_S, \hat{t}_E)$. If a qualifying element $\hat{s}$ is found in SA, it is replaced by $(e, [\hat{t}_S, t_E))$ by calling REMOVE(SA, $\hat{s}$) and INSERT(SA, $(e, [\hat{t}_S, t_E))$). For this special situation, it might be more efficient to introduce a separate *update* method as only the secondary order has to be re-established. In the case of a finite input stream, all elements of SA can be released and removed after the last element was processed.

Correctness. The coalesce operation is a kind of physical optimization that diminishes the number of elements streaming through a query graph. Applying coalesce has no effect on the validity of stream elements since only adjacent time intervals of value-equivalent elements are merged. Thus at chronon level, S_{in} and S_{out} are transformed into identical logical streams. Therefore, S_{out} is just a more compact representation of S_{in}.

The QUERY method determines elements within the SweepArea that qualify for a merge. If such an element $\hat{s}$ is found, it is merged with the incoming element s and the SweepArea is updated. So, there is no loss of information. The correct ordering inherently results from calling REMOVE and INSERT, or the more efficient update strategy mentioned above.

During reorganization p_{remove} enforces to hold possibly qualifying elements in SA where $\hat{t}_E = t_S$. This property differs from our common way of reorganization. The additional condition $\hat{t}_E - \hat{t}_S > \varepsilon$ does not violate the ordering $\leq_t$ of S_{out}, but avoids a blocking behavior in the worst case by restricting the maximum length of time intervals.

Algorithm 11: Split

Input : stream S_{in}, value-equivalence predicate θ, SweepArea SA, time interval size ε
Output: stream S_{out}

```
1 S_out ← ∅;
2 foreach s := (e, [t_S, t_E)) ↩ S_in do
3 |   T t̃_S ← t_S;
4 |   while t̃_S + ε < t_E do
5 |   |   INSERT(SA, (e, [t̃_S, t̃_S + ε)));
6 |   |   t̃_S ← t̃_S + ε;
7 |   if t̃_S < t_E then INSERT(SA, (e, [t̃_S, t_E)));
8 |   REORGANIZE(SA, s, S_out);
```

4.3.2.11 *Split*

Description. The split operator (see Algorithm 11) is inverse to the coalesce operator. For each incoming element $s := (e, [t_S, t_E))$, split fragments the time interval $[t_S, t_E)$ in subintervals of size ε, if possible. Hence, a sequence of value-equivalent elements with adjacent time intervals is created from a single incoming element. These elements are inserted into the SweepArea.

The remove predicate used by REORGANIZE is defined as follows:

$$p_{remove}((e, [t_S, t_E)), (\hat{e}, [\hat{t}_S, \hat{t}_E))) := \begin{cases} true & , t_S \geq \hat{t}_S \\ false & , \text{otherwise} \end{cases}.$$

Contrary to other algorithms were REORGANIZE is often called before any insertions, REORGANIZE is executed after INSERT in this case. Semantically, it does not matter when REORGANIZE is called. But the latter case has the advantage that the first fragment of s can already be released to S_{out}. If S_{in} is finite, the whole SweepArea can be appended to S_{out} and deleted after the last element was processed.

Correctness. In analogy to coalesce, S_{in} and S_{out} represent the same logical stream. However, split fragments an incoming stream element into a sequence of value-equivalent elements with adjacent time intervals. Hence, split performs a lossless segmentation.

All elements produced by split have time intervals of size ε, except for the last generated time interval. For that purpose, the timestamp $\tilde{t}_S$, which is initially set to t_S, is incremented each time by ε time units as long as the end timestamp t_E is smaller than $\tilde{t}_S$. The created stream elements consisting of a record e and adjacent subintervals of $[t_S, t_E)$ are inserted into the SweepArea. S_{out} is ordered by $\leq_t$ since INSERT maintains the ordering $\leq_t$ in SA and REORGANIZE follows this internal linkage. The ordering in S_{out} is not violated as the predicate p_{remove} does not expire elements too early due to the ordering invariant of S_{in}.

4.4 PIPES

PIPES (*Public Infrastructure for Processing and Exploring Streams*) [Krämer and Seeger 2004] is an infrastructure with fundamental building blocks that allow the construction of a fully functional DSMS tailored to a specific application scenario. Contrary to existing approaches, we do not intend to build a monolithic DSMS since we believe that it is almost impossible to develop a general, performant as well as flexible system that can cope with the manifold requirements of data stream applications.

The core of PIPES is a powerful and generic physical operator algebra whose semantics and implementation concepts are presented in this paper. In addition, PIPES provides frameworks for the necessary runtime components such as the scheduler, memory manager, and query optimizer to execute physical operator plans.

Since PIPES seamlessly extends the Java library XXL [Bercken et al. 2001] towards continuous data-driven query processing over autonomous data sources, it has full access to XXL's query processing frameworks such as the extended relational algebra, connectivity to remote data sources or index structures. Therefore,

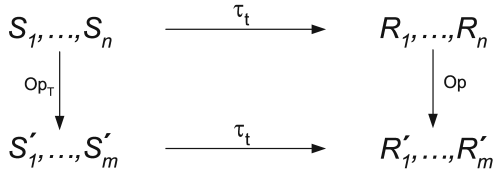

Fig. 6. Snapshot reducibility

PIPES inherently offers to run queries over streams and relations [Arasu et al. 2003b].

5. QUERY OPTIMIZATION

The foundation for a logical as well as physical query optimization is a precisely defined semantics. Therefore, we formally presented a sound logical operator algebra over data streams (see Section 3) that is expressive enough to support state-of-the-art continuous query processing.

It remains to provide an equivalence relation that defines when two logical query plans are equivalent. Based on this definition, we also derive an equivalence relation for physical streams in this section.

5.1 Snapshot-Reducibility

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

Definition 5.1. (*Timeslice*) The timeslice operator is a map $\tau_t : (\mathbb{S}^l \times T) \rightarrow \wp(\Omega \times \mathbb{N})$ given by

$$\tau_t(S^l) := \{(e, n) \in \Omega \times \mathbb{N} \mid (e, t, n) \in S^l\} \tag{17}$$

For a given logical stream S^l and a specified point in time t, the timeslice operation returns a non-temporal multiset of all records in S^l that are valid at time instant t. Note that the argument timestamp is given as subscript. The corresponding schema results from a projection to the record schema and the multiplicity attribute.

Definition 5.2. (*Snapshot-Reducibility*) A logical stream operator op_T is *snapshot-reducible* to its non-temporal counterpart *op* over multisets, if for any point in time $t \in T$ and for all logical input streams $S_1^l, \ldots, S_n^l \in \mathbb{S}^l$, the snapshot at t of the results of applying op_T to $S_1^l, \ldots, S_n^l$ is equal to the results of applying *op* to the snapshot $R_1, \ldots, R_n$ of $S_1^l, \ldots, S_n^l$ at time t.

For example, the duplicate elimination over logical streams is snapshot-reducible to the duplicate elimination over multisets. Figure 6 gives a commuting diagram that illustrates snapshot-reducibility.

Snapshot-reducibility is a well-known concept from the temporal database community [Slivinskas et al. 2000; Böhlen et al. 1998] and guarantees that the semantics of a non-temporal operator is preserved in its more complex, temporal counterpart. If we assume the record schema of a logical or physical stream to be relational, we

can show via snapshot-reducibility that our operators extend the well-understood semantics of the extended relational algebra. In addition, we introduced novel temporal operators, like the window operator, in order to provide an adequate basis for temporal continuous query formulation and execution over data streams.

Applying snapshot-reducibility, we can also prove that our semantics covers the relational approach proposed by Arasu et al. [Arasu et al. 2003b], while maintaining the advantages of our implementation described in Section 4.

5.2 Stream Equivalences

Based on the timeslice operator, we define the following equivalence relations for schema-compliant logical and physical streams, respectively:

Definition 5.3. (*Logical stream equivalence*) We define two logical streams S_1^l, $S_2^l \in \mathbb{S}^l$ to be *equal iff* all snapshots of them are equal.

$$S_1^l \doteq S_2^l :\Leftrightarrow \forall\ t \in T.\ \tau_t(S_1^l) = \tau_t(S_2^l) \tag{18}$$

Definition 5.4. (*Physical stream equivalence*) Let $S_1^p = (M_1, \leq_t)$, $S_2^p = (M_2, \leq_t) \in \mathbb{S}^p$ be two physical streams. We denote two physical streams as *snapshot-equivalent iff* their corresponding logical streams are equal.

$$S_1^p \ \backsimeq\ S_2^p \ :\Leftrightarrow\ \tau(M_1) \ \doteq\ \tau(M_2) \tag{19}$$

Note that snapshot-equivalence over physical streams abstracts from their ordering.

We denote two query plans over the same set of input streams as *equivalent* if each output stream of the first query plan is stream-equivalent and schema-compliant to exactly one output stream of the second query plan, and vice versa.

5.3 Transformation Rules

Based on the previous equivalence relations that rely on snapshot equivalence over multisets, we can derive a plethora of transformation rules to optimize algebraic expression, i. e. logical query plans. Due to the fact that we defined most of our operations, except group and window, in compliance with [Slivinskas et al. 2001], the huge set of conventional and temporal transformation rules for snapshot-equivalence over multisets listed in [Slivinskas et al. 2001] also holds in the stream context. This includes common transformation rules such as join reordering or predicate push-down, and additional temporal transformation rules for duplicate elimination, coalescing etc. Let S_1^l, S_2^l, and S_3^l be logical streams. Transformed into our notation, the following transformation rules hold, inter alia:

$$\begin{aligned}
(S_1^l \times S_2^l) \times S_3^l &\doteq S_1^l \times (S_2^l \times S_3^l) \\
\sigma_p(S_1^l \cup_+ S_2^l) &\doteq \sigma_p(S_1^l) \cup_+ \sigma_p(S_2^l) \\
\delta(S_1^l \times S_2^l) &\doteq \delta(S_1^l) \times \delta(S_2^l) \\
&\ldots
\end{aligned}$$

Example: Figure 2 (b) depicts a possible algebraic optimization of the query plan in our example query by pushing the selection down the union operator. There, we apply a generalized variant of the second transformation rule listed above. For

logical streams $S_1^l, \ldots S_n^l$, the following equation holds:

$$\sigma_p(S_1^l \cup_+ \ldots \cup_+ S_n^l) \doteq \sigma_p(S_1^l) \cup_+ \ldots \cup_+ \sigma_p(S_n^l)$$

The physical union operation is a stateful operator that internally reorders the incoming elements to ensure the ordering invariant of the physical output stream. Therefore, this transformation rule generally reduces the memory usage of the union operator.

The transformation rules for the aggregation specified in [Slivinskas et al. 2001] are only applicable if we combine our group, aggregation and union operator such that we compute the aggregate for each group and merge the results of the aggregation operators. Let S^l be a logical stream, γ_{gf} be the group operator with a grouping function g that splits the input stream into k output streams, π_i be map operator that maps to the i-th group for $i \in \{1, \ldots, k\}$, α_f be the aggregation operator with an aggregation function f, and $\cup_+$ the union operator generalized to multiple input streams. Then, we build the conventional aggregation operator by $\cup_+(\alpha_f(\pi_1(\gamma_g(S^l))), \ldots, \alpha_f(\pi_k(\gamma_g(S^l))))$. However, we decided to split the group and aggregation operation because in the context of continuous stream processing, a group operation that splits an input stream into multiple output streams (see Section 3.2.6) may be beneficial for subquery sharing.

These transformation rules are only a first step towards static and dynamic query optimization over data streams and relations. Due to continuous queries, DSMS generally run a large number of queries in parallel. So, it is not sufficient to apply transformation rules solely for a single query plan. Instead, the complete query graph should be optimized. This includes the sharing of preferably large subqueries as well as the need for a dynamic re-optimization of subgraphs during runtime. Currently, we are investigating to what extent research results from multi-query optimization [Sellis 1988; Roy et al. 2000; Leung and Muntz 1993] can be applied to optimize multiple continuous queries over streams.

5.3.1 *Window Transformation Rules.* The window operator is typically placed near the sources in a query plan because it sets the validity of the stream elements (see Section 2.6). Stateful operators (see Section 4.2.1) use the end timestamps of stream elements for reorganization. For that reason, the window operator has to be placed previous to the first stateful operator in a query plan which means that it is not commutative with stateful operators. However, we can derive some transformation rules for stateless operations. A stateless operator is commutative with the window operator, if it does not consider the end timestamp. Then, the following transformation rules hold for a logical stream $S^l \in \mathbb{S}^l$:

$$\begin{aligned}
\sigma_p(\omega_w(S^l)) &\doteq \omega_w(\sigma_p(S^l)) \\
\mu_f(\omega_w(S^l)) &\doteq \omega_w(\mu_f(S^l)) \\
\gamma_f(\omega_w(S^l)) &\doteq (\omega_w(\pi_1(\gamma_f(S^l))), \ldots, \omega_w(\pi_k(\gamma_f(S^l))))
\end{aligned}$$

The group operation (see Section 3.2.6) produces a tuple of k logical streams since it splits the logical input stream S^l into k groups according to a user-defined grouping function f. Therefore, the window operator ω_w has to be applied to each logical output stream of the group operator.

6. RELATED WORK

A preliminary version of this work will appear in [Krämer and Seeger 2005]. In comparison, this paper broadens the focus of [Krämer and Seeger 2005] by introducing the logical as well as the corresponding physical operator algebra. Hence, it gives an additional insight into our implementation issues and underlying algorithms, which eventually emphasizes the feasibility of our approach. Moreover, this paper includes helpful enhancements to some sections for a better understanding.

Motivated by several approaches creating the *timestamps* of stream elements in different ways [Babcock et al. 2002; Carney et al. 2002; Babcock et al. 2003; Golab and Özsu 2003], we decided to develop an operator algebra with a well-defined, deterministic semantics as a foundation for query optimization. [Babcock et al. 2002] and [Carney et al. 2002] propose that the user should specify a function that creates the timestamp to be assigned to a result of an operation. This very flexible and powerful solution has the drawback that the semantics of an operation with multiple inputs may become ambiguous as it depends on a user-defined function. Babcock et. al. suggest in [Babcock et al. 2003] to assign the maximum associated timestamp of two qualifying join elements to a join result, whereas [Golab and Öszu 2003] keep all timestamps of qualifying elements in the resultant join tuple of their multi-way join. In contrast to assigning the maximum timestamp to a result, this approach is more powerful as it does not omit any temporal information. However, it lacks from the fact that the size to store the temporal information of a stream element is not constant. Our approach shows that it is sufficient to take time intervals to represent the temporal information in order to achieve an unambiguous semantics, while guaranteeing constant costs to store the temporal information.

Our work is closely related to *multiset (bag) semantics* and *algebraic equivalences* of the relational algebra [Dayal et al. 1982; Albert 1991; Garcia-Molina et al. 2000] as it transfers the well-known operators of the extended relational algebra towards continuous query processing over data streams. In order to acquire a deterministic semantics, we had to introduce the notion of time. Therefore, our work is strongly related to *temporal databases*, although the construction of time intervals according to temporal windows is novel. We found the work of [Slivinskas et al. 2000; 2001] to be an appropriate starting point for our research. While still being in agreement with the semantics proposed in [Slivinskas et al. 2001], we extended this work in the following points. First, we defined a descriptive, temporal, logical operator algebra similar to the non-temporal counterpart proposed by [Dayal et al. 1982]. This is contrary to Slivinskas et al. who do not distinguish between a logical and a physical operator algebra. They rather specify the semantics of their operations from an implementation point of view using the λ-calculus. However, we are convinced that separating the logical operator algebra from the physical goes along with several advantages. It is especially helpful, even for the temporal database community, to clearly illustrate the snapshot semantics because our logical algebra considers time instants at finest time granularity (chronons) which correspond to snapshots. We already mentioned this advantage in Section 3.2, e. g. for the difference operation which is simply reduced to the difference in the multiplicities at each time instant. This definition is much more intuitive and compact than the one given in [Slivinskas et al. 2001] based on the λ-calculus. In addition, the logical

algebra can be leveraged to prove semantic equivalences of operator plans while abstracting from a specific implementation. Second, we extended the operators in [Slivinskas et al. 2001] towards data-driven stream processing and discussed their non-blocking implementation. This extension includes the definition of suitable windowing constructs and novel transformation rules as well as the separate handling of grouping and aggregation to permit subquery sharing. Third, our approach is not limited to relational schemas due to the parameterization of our physical operators by functions and predicates. Fourth, the distinction between the logical and the physical algebra is an extension of well-established database technology. We are convinced and also know from the experience with our library XXL that such a distinction facilitates the adaption of traditional components to stream processing, e. g. the query optimizer. Moreover, it enables to combine stream processing and traditional query processing within a single query. This involves query translation as well as query execution under a unique framework.

Since our physical operators produce a snapshot-equivalent output to the operators presented in [Slivinskas et al. 2001], our approach benefits from [Slivinskas et al. 2001] with regard to *query optimization* by making a plethora of conventional and temporal transformation rules applicable in the stream context. In particular, existing work on temporal query optimization is considered [Gunadhi and Segev 1990; Leung and Muntz 1993]. Altogether, our approach establishes a semantic foundation for query optimization over data streams. This is contrary to other approaches [Viglas and Naughton 2002; Kang et al. 2003; Avnur and Hellerstein 2000] that focus on cost models considering stream rates and selectivities as optimization criteria, typically for a reduced operator set consisting of selection, projection, and join. These do not delve into semantic equivalences and transformation rules of general queries.

Formulating and executing continuous queries over data streams has also been a field of considerable research. Closest to our work is STREAM [Arasu et al. 2003b]. The semantics of queries inside this DSMS is specified in [Arasu et al. 2003a] and [Arasu and Widom 2004]. [Arasu et al. 2003a] propose an abstract semantics for a concrete query language over streams and relations. The semantics exploits the relational semantics and relies on specific conversion-operators to transform a stream into a relation and vice versa. Due to the fact that Arasu et al. define a relation as a map from the time domain to a finite but unbounded bag of tuples, they implicitly introduce snapshot-reducibility. For that reason, we can prove that the subset of relational operators in our algebra produces snapshot-equivalent results to the operators in [Arasu et al. 2003a]. Hence, our temporal approach is at least as expressive as the one proposed by Arasu et al., but also offers to construct more complex, temporal operator variants. Windows are expressed by a stream-to-relation operator defining the output relation over time. From a semantic point of view, this is equivalent to our approach, whereas [Arasu et al. 2003a] do not consider fixed windows. Arasu et al. define three different types of stream operators: `Istream`, `Dstream`, and `Rstream`. `Istream` applied to a relation R creates a new stream element whenever a tuple was inserted into R within the recent time instant. `Dstream` streams the deleted elements accordingly, whereas `Rstream` streams all elements of R at the current time instant. These conversion techniques can also be applied in our

stream algebra to enable continuous queries over both, streams and relations. Altogether, our approach is compatible to the stream query language *CQL* (standing for *Continuous Query Language*) proposed in [Arasu et al. 2003a]. The semantics of this query language is specified in a denotational style in [Arasu and Widom 2004], whereas our approach, in particular the algorithmic description of the operations, specifies the semantics in an operational manner. The operator semantics is discussed only for some examples in [Arasu and Widom 2004], whereas we discuss the semantics of each operator in detail and provide a descriptive operator semantics based on temporal multisets in addition.

The main difference between STREAM and PIPES is not the semantics but rather the *implementation*. [Arasu et al. 2003b] gives an insight into the implementation aspects of STREAM. As already mentioned in Section 4, Arasu et al. implemented a positive-negative approach to incrementally maintain the changing state of a relation in the STREAM system correctly. At this, an incoming element tagged with a '+' indicates that this element starts to be valid at the associated point in time, whereas a '-' tag indicates that this elements expires at the associated time instant. This is totally different to our approach that relies on time intervals to model the validity of records. Illustratively argued, an element $(e, [t_S, t_E))$ of a physical stream is replaced by two stream elements $(e, t_S, +)$ and $(e, t_E, -)$ in the positive-negative approach. This implies that the number of elements streaming through a query graph is doubled which is a significant drawback due to the following reasons. First, there is a substantial computational overhead, especially with regard to scheduling. Second, the total memory usage is largely increased since operators communicate through intermediate queues. Third, system stability and scalability may suffer from the inherently higher stream rates. Our contrary approach avoids all these disadvantages as it is based on time intervals.

To the best of our knowledge, no work has been published on a physical operator algebra that utilizes time intervals similar to our approach for the execution of continuous queries. Although [Hammad et al. 2003] consider time intervals for stream processing, they prefer a positive-negative approach in analogy to [Arasu et al. 2003b] with the argument of a simpler implementation. However, their time interval approach is different to ours since their time intervals do not correspond to validities. Furthermore, the presented algorithms are not purely data-driven as in our approach because they assume that an operator can choose from which input the next element is consumed due to separating the operators by queues. PIPES allows direct interoperability [Cammert et al. 2003], i. e., operators may be plugged together without intermediate queues. This is possible because a direct publish-subscribe mechanism and additional synchronization concepts are inherently integrated into our operators. In comparison to [Arasu et al. 2003b], who also use queues for implementing operator communication, Hammad et al. discuss the handling of tagged elements more detailed by specifying concrete algorithms for their query processing engine *Nile*.

The following projects are also related to our work but not as closely as the ones mentioned above. *Tribeca* [Sullivan and Heybey 1998] introduces fixed and moving window queries over network streams. Because their operators are restricted to a single input and output stream, there is no support for joins or difference

across streams. *TelegraphCQ* [Chandrasekaran et al. 2003] relies on a declarative language to express a sequence of windows over a stream. ATLaS [Wang et al. 2003] is a SQL extension that permits a delta-based computation of aggregates on windows over data streams, particularly in the data mining field. Gigascope [Cranor et al. 2003] is a stream database for network monitoring applications based on a SQL-like query language. In analogy to [Tucker et al. 2001], input streams are analyzed to infer constraints which bound the state required to evaluate blocking operators. Consequently, stream constraints are used instead of windows to unblock otherwise blocking operators. In a similar way, logical data expiration [Toman 2003] explores techniques that can be used to limit the growth of historical data in warehouses. Taking a look at our approach, the ordering invariant of a stream (see Section 4.2.2) is comparable to such a constraint as we use it to reorganize the SweepAreas efficiently. *Aurora* [Carney et al. 2002; Abadi et al. 2003] builds a query graph of stream operators parameterized by functions and predicates while abstracting from a certain query language, which is similar to our approach. The operations in Aurora are defined in a procedural manner and allow out-of-order elements in streams as well as certain actions which may cause a nondeterministic semantics due to scheduling dependencies. This is contrary to our work that assigns an unambiguous meaning to every query by formalizing a logical and a physical operator algebra. The *Tapestry* system [Terry et al. 1992] transforms a continuous query into an incremental query that is run periodically. Tapestry ensures snapshot-reducibility but does not support any kind of window queries.

In a broader context, our approach is related to *sequence databases* [Seshadri et al. 1996] since raw input streams are a temporally ordered sequence of records. Note that the semantics of sequence languages includes one-time but not continuous queries. The *chronicle data model* [Jagadish et al. 1995] provides operators over relations and chronicles, which can be considered as a raw input stream, but focuses on the space complexity of an incremental maintenance of materialized views over chronicles. It does not include continuous queries or aspects of data-driven processing. We finally refer the interested reader to [Arasu et al. 2003b; Golab and Özsu 2003] for a broader *overview* on data stream processing and stream query languages.

7. CONCLUSIONS

Inspired by the lack of a formal semantics for continuous queries over data streams, we first proposed a sound temporal logical operator algebra that exploits and extends the well-known semantics of the extended relational algebra as well as existing work in temporal databases. Second, we described the implementation issues of our physical operator algebra which consists of efficient, non-blocking, data-driven, stream-to-stream implementations of the logical operations. To the best of our knowledge, this approach is unique as it assigns stream elements with time intervals to model their validity, independent from the granularity of time. Due to snapshot-reducibility, our approach is logically compliant to related approaches [Arasu et al. 2003b], while it does not suffer from higher stream rates arising from positive-negative elements used to indicate element expiration. We explained why our physical operations produce sound results in terms of a snapshot-equivalent

output to their logical counterparts. We further showed how to efficiently reorganize stateful operators by exploiting the time intervals of stream elements as well as the ordering invariant assumed for streams. Third, appropriate stream equivalences derived from the snapshot-multiset equivalence specified in [Slivinskas et al. 2001] allow us to apply most conventional as well as temporal transformation rules. To support sliding and fixed window queries, we furthermore introduced a novel window operator, by defining its semantics, describing its implementation, and extending the set of transformation rules. Moreover, we motivated a novel kind of physical optimization in the stream context by proposing two physical operators, coalesce and split, which can effectively be used to adaptively influence the runtime behavior of a DSMS with regard to stream rates, memory consumption as well as early results. Consequently, our work forms a solid foundation for query formulation and optimization in the context of continuous query processing over data streams, while it relies on the common, well-known steps from query formulation to query execution established in DBMS over the years.

We already proved the feasibility of our approach during the development of PIPES [Krämer and Seeger 2004], our infrastructure for continuous query processing over heterogeneous data sources where the temporal semantics proposed in this paper was implemented.

Acknowledgments

This project has been supported by the German Research Society (DFG) under grant no. SE 553/4-1. In addition, we are grateful to Michael Cammert and Christoph Heinz for the helpful discussions on the semantics of stream operations.

REFERENCES

Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. B. 2003. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal 12,* 2, 120–139.

Albert, J. 1991. Algebraic Properties of Bag Data Types. In *Proc. of the Conf. on Very Large Databases (VLDB)*. 211–219.

Arasu, A., Babu, S., and Widom, J. 2003a. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. In *Proc. of the Intl. Conf. on Data Base Programming Languages (DBPL)*.

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

Arasu, A. and Widom, J. 2004. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record 33,* 3, 6–12.

Avnur, R. and Hellerstein, J. M. 2000. Eddies: Continuously Adaptive Query Processing. In *Proc. of the ACM SIGMOD*. ACM Press, 261–272.

Babcock, B., Babu, S., Datar, M., and Motwani, R. 2003. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. of the ACM SIGMOD*. 253–264.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. 2002. Models and Issues in Data Stream Systems. In *Symp. on Principles of Database Systems (PODS)*. 1–16.

Bercken, J., Blohsfeld, B., Dittrich, J.-P., Krämer, J., Schäfer, T., Schneider, M., and Seeger, B. 2001. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proc. of the Conf. on Very Large Databases (VLDB)*. 39–48.

Bettini, C., Dyreson, C. E., Evans, W. S., Snodgrass, R. T., and Wang, X. S. 1997. A Glossary of Time Granularity Concepts. In *Temporal Databases: Research and Practice*. Lecture Notes in Computer Science, 406–413.

Böhlen, M. H., Busatto, R., and Jensen, C. S. 1998. Point-Versus Interval-Based Temporal Data Models. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*. 192–200.

Cammert, M., Heinz, C., Krämer, J., Markowetz, A., and Seeger, B. 2003. PIPES - A Multi-Threaded Publish-Subscribe Architecture for Continuous Queries over Streaming Data Sources. Technical report, University of Marburg. CS-32.

Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. B. 2002. Monitoring Streams: A New Class of Data Management Applications. In *Proc. of the Conf. on Very Large Databases (VLDB)*. 215–226.

Chandrasekaran, S., Cooper, O., and et al., A. D. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*.

Cranor, C. D., Johnson, T., Spatscheck, O., and Shkapenyuk, V. 2003. Gigascope: A Stream Database for Network Applications. In *Proc. of the ACM SIGMOD*. 647–651.

Dayal, U., Goodman, N., and Katz, R. H. 1982. An Extended Relational Algebra with Control Over Duplicate Elimination. In *Proc. of the ACM SIGMOD*. 117–123.

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 *Proc. of the Conf. on Very Large Databases (VLDB)*. 299–310.

Garcia-Molina, H., Ullman, J. D., and Widom, J. 2000. *Database System Implementation*. Prentice Hall.

Golab, L. and Öszu, M. T. 2003. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. of the Conf. on Very Large Databases (VLDB)*. 500–511.

Golab, L. and Özsu, M. T. 2003. Issues in Data Stream Management. *SIGMOD Record 32,* 2, 5–14.

Graefe, G. 1993. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys 25,* 2, 73–170.

Gunadhi, H. and Segev, A. 1990. A Framework for Query Optimization in Temporal Databases. In *Proc. of the Int. Conf. on Statistical and Scientific Database Management*. Springer-Verlag New York, Inc., 131–147.

Haas, P. J. and Hellerstein, J. M. 1999. Ripple Joins for Online Aggregation. In *Proc. of the ACM SIGMOD*. ACM Press, 287–298.

Hammad, M., Aref, W., Franklin, M., Mokbel, M., and Elmagarmid, A. 2003. Efficient Execution of Sliding Window Queries over Data Streams. Technical report, Purdue University.

Hellerstein, J. M., Haas, P. J., and Wang, H. 1997. Online Aggregation. In *Proc. of the ACM SIGMOD*. 171–182.

Jagadish, H. V., Mumick, I. S., and Silberschatz, A. 1995. View Maintenance Issues for the Chronicle Data Model. In *Proc. of the ACM SIGMOD*. 113–124.

Kang, J., Naughton, J., and Viglas, S. 2003. Evaluating Window Joins over Unbounded Streams. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*.

Krämer, J. and Seeger, B. 2004. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of the ACM SIGMOD*. 925–926.

Krämer, J. and Seeger, B. 2005. A Temporal Foundation for Continuous Queries over Data Streams. In *Proc. of the Int. Conf. on Management of Data (COMAD)*. (to appear).

Leung, T. Y. C. and Muntz, R. R. 1993. Stream Processing: Temporal Query Processing and Optimization. In *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 329–355.

Nievergelt, J. and Preparata, F. P. 1982. Plane-Sweep Algorithms for Intersecting Geometric Figures. *Communications of the ACM 25,* 10, 739–747.

Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proc. of the ACM SIGMOD*. 249–260.

Sellis, T. K. 1988. Multiple-Query Optimization. *ACM Transactions on Database Systems (TODS) 13,* 1, 23–52.

Seshadri, P., Livny, M., and Ramakrishnan, R. 1996. The Design and Implementation of a Sequence Database System. In *Proc. of the Conf. on Very Large Databases (VLDB)*. 99–110.

Slivinskas, G., Jensen, C. S., and Snodgrass, R. T. 2000. Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*. 547–558.

Slivinskas, G., Jensen, C. S., and Snodgrass, R. T. 2001. A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 13,* 1, 21–49.

Srivastava, U. and Widom, J. 2004. Flexible Time Management in Data Stream Systems. In *Symp. on Principles of Database Systems (PODS)*. 263–274.

Sullivan, M. and Heybey, A. 1998. Tribeca: A System for Managing Large Databases of Network Traffic. In *In Proc. of the USENIX Annual Technical Conference*. 13–24.

Terry, D. B., Goldberg, D., Nichols, D., and Oki, B. M. 1992. Continuous Queries over Append-Only Databases. In *Proc. of the ACM SIGMOD*. 321–330.

Toman, D. 2003. Logical Data Expiration. In *Logics for Emerging Applications of Databases*. 203–238.

Tucker, P. A., Maier, D., Sheard, T., and Fegaras, L. 2001. Exploiting Punctuation Semantics in Continuous Data Streams. *Transactions on Knowledge and Data Engineering 15,* 3, 555–568.

Viglas, S. D. and Naughton, J. F. 2002. Rate-based Query Optimization for Streaming Information Sources. In *Proc. of the ACM SIGMOD*. 37–48.

Wang, H., Zaniolo, C., and Luo, C. 2003. ATLAS: A Small but Complete SQL Extension for Data Mining and Data Streams. In *Proc. of the Conf. on Very Large Databases (VLDB)*. 1113–1116.

Zhu, Y., Rundensteiner, E. A., and Heineman, G. T. 2004. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proc. of the ACM SIGMOD*. 431–442.