A Temporal Foundation for Continuous Queries over Data Streams

Jürgen Krämer and Bernhard Seeger

Department of Mathematics and Computer Science
Philipps-University Marburg, Germany

{kraemerj,seeger}@informatik.uni-marburg.de

ABSTRACT

Despite the surge of research in continuous stream processing, there is still a semantical 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.

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 [4, 13, 8, 25, 10, 29] like network and traffic monitoring. In order to express continuous queries, different query languages have been proposed recently [1, 10, 29, 3, 13]. 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.

ADVANCES IN DATA MANAGEMENT 2005
Jayant Haritsa, T.M. Vijayaraman (Editors)
© CSI 2005

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 [2].
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 [28]. Moreover, dynamic query re-optimization at runtime [30] is required to adapt to changes in the system load. Eventually, multi-query optimization [20, 19] 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 [11]. 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 [22, 23], 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 [23], 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 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 [17].

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 exemplarily express a query in some fictive SQL-like query language using the sliding window expressions from CQL [2].

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 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 [6]. Let $\mathbb{I} := \{[t_s, t_e) \in T \times T \mid t_s < t_e\}$ be the set of time intervals.

Definition 2. (*Physical Stream*) A pair $S^p = (M, \leq_{t_s,t_e})$ 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,
- $\leq_{t_s,t_e}$ 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 3. (*Value-equivalence*) Let $S^p = (M, \leq_{t_s,t_e}) \in \mathbb{S}^p$ 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 $\leq_{t_s,t_e}$ 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 [24] 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 [13]:

- 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.

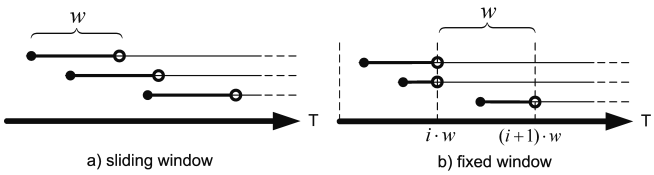

Figure 1: Windowing constructs

- 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 $S^p = (M, \leq_{t_s,t_e}) \in \mathbb{S}^p$ 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 [25], we divide the time domain $\mathbb{T}$ in sections of fixed size $w \in T$. Hence, each section 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. 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 [11] towards a temporal semantics and windowing constructs.

3.1 Logical Streams

Definition 4. (*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 [11]. 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 $S^p = (M, \leq_{t_s,t_e}) \in \mathbb{S}^p$ 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 [23]: filter(σ), map (μ), Cartesian product ($\times$), duplicate elimination (δ), difference ($-$), group (γ), aggregation (α), union ($\cup$) and window (ω).
Appendix A 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 $\sigma : \mathbb{S}^l \times \mathbb{P} \to \mathbb{S}^l$ returns all elements of a logical stream $S^l \in \mathbb{S}^l$ that fulfill the predicate $p \in \mathbb{P}$ with $p : (\Omega \times T) \to \{true, false\}$. 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 $\mu_f : \mathbb{S}^l \times \mathbb{F}_{map} \to \mathbb{S}^l$ 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 $f : \Omega \to \Omega$. Note, that f can also express an n-ary function due to the definition of Ω as a universe of all elements.

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

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 $\times : \mathbb{S}^l \times \mathbb{S}^l \to \mathbb{S}^l$ of two logical streams $S_1^l, S_2^l \in \mathbb{S}^l$ is defined by:

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

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 $\circ : \Omega \times \Omega \to \Omega$.
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 $\delta : \mathbb{S}^l \to \mathbb{S}^l$ 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 $- : \mathbb{S}^l \times \mathbb{S}^l \to \mathbb{S}^l$ 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.

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

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 [23] 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

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

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 $f : \Omega \times T \to \{1, \ldots, k\}$. Each group S_j^l represents a new logical stream for $j \in \{1, \ldots, k\}$ having the same schema as S^l.

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

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:

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

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 $S_2^l \in \mathbb{S}^l$:

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

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.

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

   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.

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

   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 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.

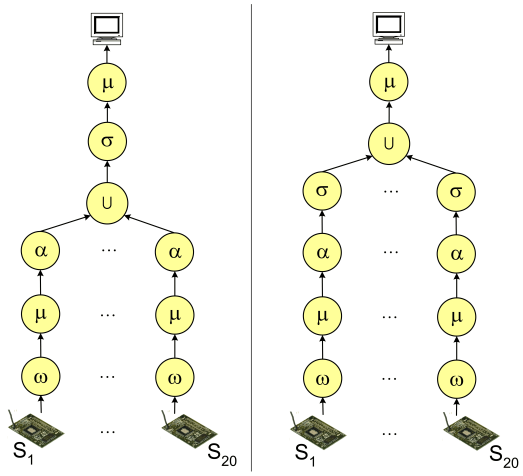

Figure 2: Query plans composed of our operations

4. PHYSICAL OPERATOR ALGEBRA

From an implementation point of view, it is not sufficient 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 the physical operator algebra in PIPES, our infrastructure for data stream processing.

To the best of our knowledge, there is no other approach to stream processing which uses time intervals to express the validity of stream elements. There are other approaches [3, 14] 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 generates a positive element by decorating the new element with a '+' which is sent through the query plan afterwards. The window operator stores all incoming elements and creates a negative element if the validity of an element in its buffer expires according to the window. The negative element, i. e. the element decorated with a '-', is pushed through the query plan and processed. 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 [22].
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:

$$M \triangleright M' :\Leftrightarrow (\exists\, m := (e, [t_s, t_e), n) \in M,$$
$$\tilde{m} := (\tilde{e}, [\tilde{t}_s, \tilde{t}_e), \tilde{n}) \in M.\ e = \tilde{e}$$
$$\wedge\ t_e = \tilde{t}_s\ \wedge\ (\exists\, M'' \in \wp(\Omega \times \mathbb{I} \times \mathbb{N}).$$
$$M'' = (M - \{m, \tilde{m}\})\ \cup \{(e, [t_s, \tilde{t}_e), 1)\}$$
$$\wedge\ M'' \triangleright M'))\ \vee\ M = M'$$
$$\text{where } M - \{m, \tilde{m}\} := (M\ \setminus\ \{m, \tilde{m}\}) \cup$$
$$(\{(e, [t_s, t_e), n-1), (\tilde{e}, [\tilde{t}_s, \tilde{t}_e), \tilde{n}-1)\}\ \setminus\ \Omega \times \mathbb{I} \times \{0\}).$$

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 $\zeta : \wp(\Omega \times \mathbb{I} \times \mathbb{N}) \rightarrow \wp(\Omega \times \mathbb{I} \times \mathbb{N})$ produces a set.

$$\zeta(M) := \{M' \in \wp(\Omega \times \mathbb{I} \times \mathbb{N}) \mid M \triangleright M' \wedge$$
$$(\forall\, M'' \in \wp(\Omega \times \mathbb{I} \times \mathbb{N}).\ (M' \not\triangleright M'')\ \vee\ (M' = M''))\} \quad (11)$$

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 $\leq_{t_s, t_e}$ 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 Implementation

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 $\leq_{t_s,t_e}$ (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 as stateful.

This order invariant seems to be very expensive to satisfy. However, it is commonly assumed in stream processing that raw input streams arrive temporally ordered [4, 13] or mechanisms exist that ensure such a temporal ordering [24]. Moreover, our efficient algorithms for the reorganization of the internal data structures of stateful operators rely on this ordering invariant.

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. For example, in a binary join 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 of physical streams since it generally leads to early results. Our stateful operators additionally link the elements in their internal data structures according to the lexicographical order $\leq_{t_s,t_e}$. This helps to efficiently run the reorganization phase by simply following these links. The reorganization phase can be stopped if a linked element is accessed whose end timestamp is larger than $min\{t_{s_j} \mid j \in \{1, \ldots, n\}\}$. Keeping this implementation detail in mind, the secondary order helps to purge expired elements earlier.
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 [24]. A similar problem arises if the delays between subsequent elements within a physical stream are relatively long. In this case, the reorganization phase is triggered seldom, 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 Aggregation

We want to sketch the implementation of the aggregation as a non-trivial stateful operator. Let $S^p = (M, \leq_{t_s,t_e})$ be the physical input stream of the aggregation operator. Since we implemented an incremental aggregation [15], we need a binary aggregation function $f : \Omega \cup \{\bot\} \times M \to \Omega$ that is applied successively to the current aggregation value $\tilde{s} \in \Omega \cup \{\bot\}$ and an incoming element $s = (e, [t_s, t_e)) \in M$ (see Figure 3). $\bot$ is solely used to initialize the aggregate.
We first probe the internal data structure for elements whose time interval overlaps with $[t_s, t_e)$. For the case of a partial overlap we split the element into maximum subintervals with either no or full overlap, while keeping the aggregation value for each of them. Then, we update the aggregation value of all overlapping elements $\tilde{s}$ by invoking f on $(\tilde{s}, s)$. For each maximum subinterval r of $[t_s, t_e)$ for which no intersection is found in the internal data structure, we finally insert an element consisting of an initialized aggregation value $f(\bot, s)$ and the time interval r.
Thereafter, we perform the reorganization phase by removing all elements from the internal data structure whose end timestamp is smaller than t_s or equal. Those can efficiently be determined by additionally linking the elements in the internal data structure according to $\leq_{t_s,t_e}$, which corresponds to an ordering by end timestamps in this case. As each expired element contains the final aggregation value for the associated time interval, we append it to the physical output stream. Consequently, the aggregation operator produces its results during the reorganization phase.

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 3.3) 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.

4.2.5 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.

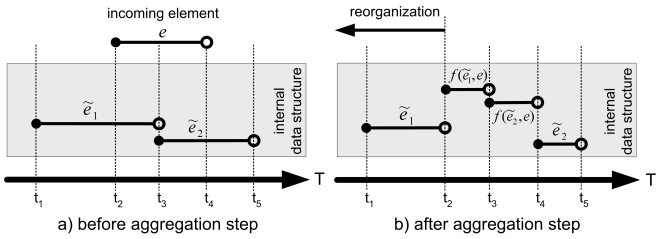

Figure 3: Aggregation operator

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.

However, these 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. Coalesce generally decreases 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 PIPES

PIPES (*Public Infrastructure for Processing and Exploring Streams*) [17] is an infrastructure with fundamental building blocks that allow the construction of a fully functional DSMS tailored to a specific application scenario. The core of PIPES is a powerful and generic physical operator algebra whose semantics and implementation concepts have been presented in this paper. In addition, PIPES provides 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 [5] 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, PIPES inherently offers to run queries over streams and relations [3].

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.

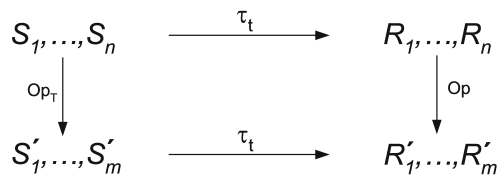

Figure 4: Snapshot reducibility

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

$$\tau_t(S^l) := \{(e, n) \in \Omega \times \mathbb{N} \mid (e, t, n) \in S^l\} \qquad (12)$$

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 6. (*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 4 gives a commuting diagram that illustrates snapshot-reducibility.

Snapshot-reducibility is a well-known concept from the temporal database community [22, 7] 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. [3], 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 7. (*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.

$$S_1^l \doteq S_2^l :\Leftrightarrow \forall\ t \in T.\ \tau_t(S_1^l) = \tau_t(S_2^l) \qquad (13)$$

Definition 8. (*Physical stream equivalence*) Let $S_1^p = (M_1, \leq_{t_s,t_e})$, $S_2^p = (M_2, \leq_{t_s,t_e}) \in \mathbb{S}^p$ be two physical streams. We denote two physical streams as *snapshot-equivalent iff* their corresponding logical streams are equal.

$$S_1^p \simeq S_2^p :\Leftrightarrow \tau(M_1) \doteq \tau(M_2) \qquad (14)$$

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 [23], the huge set of conventional and temporal transformation rules for snapshot-equivalence over multisets listed in [23] also holds in the stream context. This includes common transformation rules such as join reordering or predicate pushdown, and additional temporal transformation rules for duplicate elimination, coalescing etc.

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 following transformation rule for two logical streams S_1^l, S_2^l:

$$\sigma_p(S_1^l \cup_+ S_2^l) \doteq \sigma_p(S_1^l) \cup_+ \sigma_p(S_2^l)$$

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 aggregation specified in [23] 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. However, we decided to split the group and aggregation operations 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 in 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 [20, 19, 18] 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 $S^l \in \mathbb{S}^l$:

$$\begin{aligned}
\sigma_p(\omega_w(S^l)) &\doteq \omega_w(\sigma_p(S^l)) \\
\mu_f(\omega_w(S^l)) &\doteq \omega_w(\mu_f(S^l)) \\
\gamma_f(\omega_w(S^l)) &\doteq (\omega_w(\pi(\gamma_f(S^l),1)),\ldots,\omega_w(\pi(\gamma_f(S^l),k)))
\end{aligned}$$

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 function. Therefore, the window operator ω_w has to be applied to each logical output stream.

6. RELATED WORK

Our work is closely related to multiset (bag) semantics and algebraic equivalences for the relational algebra [11, 12] as well as their temporal extensions [22, 23]. It basically transfers the approach of [23] to continuous queries over data streams. Thereby, we abstract from relational schemes, introduce windowing constructs and provide adequate non-blocking operator implementations. This leads to a snapshot-equivalent output to the operations presented in [23] and therefore, a plethora of transformation rules is applicable for optimizing continuous queries over streams. Whereas Slivinskas et al. specify their operations from an implementation point of view with the λ-calculus, we present a suitable temporal logical operator algebra as well, which defines the semantics of the operations in a more intuitive way while abstracting from a particular implementation.

In the context of continuous queries over streams, there has also been considerable research. Tribeca [25] introduces fixed and moving window queries over single network streams. TelegraphCQ [9] relies on a declarative language to express a sequence of windows over a stream, whereas Gigascope [10] and [27] try to unblock operations by using stream constraints instead of windows. Aurora [8, 1] 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. Contrary to PIPES, 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. The Tapestry system [26] 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. [2, 3] propose an abstract semantics for a concrete query language over streams and relations supporting only sliding windows. From a semantical point of view, our approach is at least as expressive as [2] due to snapshot-reducibility, whereas our implementation significantly benefits from our stream-to-stream operators incorporating time intervals. This avoids the drawback of higher stream rates that arise due to sending positive and negative tuples through a query plan, in order to incrementally maintain the internal relations in the STREAM system correctly. [14] also prefer the positive-negative tuple approach and focus on particular operator implementations.

In a broader context, our approach is related to sequence databases [21] 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 [16] 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 also refer the interested reader to [3, 13] for a broader overview on data stream processing.

7. CONCLUSIONS

Due to lack of a formal specification of the semantics of continuous queries over data streams, we first proposed a sound temporal logical operator algebra by exploiting and extending the well-known semantics of the extended relational algebra as well as existing work in temporal databases. Second, we described the main implementation issues of our physical operator algebra that relies on 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 modeling their validity independent from the granularity of time. Due to snapshot-reducibility our approach is logically compliant to related approaches [3], while it does not suffer from higher stream rates arising from positive-negative elements used in related approaches to indicate element expiration. We explained why the physical operations produce sound results in terms of a snapshot-equivalent output to the logical operations. We further showed how to effectively reorganize stateful operators based on the time intervals of incoming elements and the ordering invariant assumed for streams. Third, we derived appropriate stream equivalences based on snapshot-multiset equivalence which allows 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 continuous query processing over data streams, while it relies on the common, well-known steps from query formulation to query execution established in DBMS.

We already proved the feasibility of our approach during the development of PIPES [17], 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.

8. REFERENCES

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

[2] A. Arasu, S. Babu, and J. Widom. 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)*, 2003.

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

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

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

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

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

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

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

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

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

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

[13] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

APPENDIX

A. DERIVED OPERATIONS

In this appendix, 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 Section 3.2. Let S_1^l, S_2^l be logical streams.

A.1 Theta-Join

A theta join is a map $\bowtie_{p,f}\colon \mathbb{S}^l \times \mathbb{P} \times \mathbb{F}_{map} \to \mathbb{S}^l$. 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:

$$\bowtie_{p,f}(S_1^l, S_2^l) := \mu_f(\sigma_p(S_1^l \times S_2^l)) \tag{15}$$

A.2 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.

$$\ltimes_p(S_1^l, S_2^l) := S_1^l \bowtie_{p,\mu_{\pi(S_1^l)}} \delta(S_2^l) \tag{16}$$

A.3 Intersection

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

$$\cap(S_1^l, S_2^l) := S_1^l - (S_1^l - S_2^l) \tag{17}$$

A.4 Max-Union

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

$$\begin{aligned} \cup_{max}(S_1^l, S_2^l) := \{(e,t,n) \mid (\exists\ n_1.\ (e,t,n_1) \in S_1^l \\ \wedge\ \exists\ n_2.\ (e,t,n_2) \in S_2^l\ \wedge\ n = max\{n_1, n_2\}) \\ \vee\ (\exists\ n_j.\ (e,t,n_j) \in S_j^l\ \wedge\ \nexists\ n_k.\ (e,t,n_k) \in S_k^l \\ \wedge\ n = n_j \text{ for } j,k \in \{1,2\}\ \wedge\ j \neq k)\} \\ = (S_1^l - S_2^l) \cup_+ (S_2^l - S_1^l) \cup_+ (S_1^l \cap S_2^l) \end{aligned} \tag{18}$$

This definition complies with the one proposed by [23].

A.5 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:

$$-_{strict}(S_1^l, S_2^l) := S_1^l - (S_1^l \ltimes_= S_2^l) \tag{19}$$

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

However, from a 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, PIPES provides specific implementations in addition.