Stream Schema: Providing and Exploiting Static Metadata for Data Stream Processing

Peter M. Fischer [#], Kyumars Sheykh Esmaili [#], Renee J. Miller [⋄]

[#] *Systems Group, Department of Computer Science*
ETH Zurich, Switzerland
`{peter.fischer,kyumarss}@inf.ethz.ch`
[⋄] *Department of Computer Science*
University of Toronto, Canada
`miller@cs.toronto.edu`

Abstract

Schemas, and more generally metadata specifying structural and semantic constraints, are invaluable in data management. They facilitate conceptual design and enable checking of data consistency. They also play an important role in permitting semantic query optimization, that is, optimization and processing strategies that are often highly effective, but only correct for data conforming to a given schema. While the use of metadata is well-established in relation and XML databases, the same is not true for data streams. What work exists mostly focuses on the specification of dynamic information. In this paper, we consider the specification of static metadata for streams in a model called Stream Schema. We show how Stream Schema can be used to validate the consistency of streams. By explicitly modeling stream constraints, we show that stream queries can be simplified by removing predicates or subqueries that check for consistency. This can greatly enhance programmability of stream processing systems. We also present a set of semantic query optimization strategies that both permit compile-time checking of queries (for example, to detect empty queries) and new runtime processing options, options that would not have been possible without a Stream Schema specification. Case studies on two stream processing platforms (covering different applications and underlying stream models), along with an experimental evaluation, show the benefits of Stream Schema.

1 Introduction

Data stream processing has been a hot topic in the database community for most of this decade, leading to numerous publications, prototype systems, startup companies and commercial products. One area within data stream processing research that has received relatively little attention is the use of metadata, in particular static metadata. Of course dynamic properties of streams, such as constraints on arrival rates, have long been exploited for optimization [30]. However, beyond a few limited proposals (including K-Constraints [9] and Gigascope [14]), structural and semantic constraints on stream data have not been exploited in a systematic way. It is well-known in data management, that such constraints, if they explicitly specified, can be used to check data consistency, improve application modeling, extend query semantics, and permit new forms of semantic query optimization, specifically the application of optimizations that are correct (and potentially highly efficient) over data satisfying a given set of constraints. In this paper, we present a new approach for modeling structural and semantic constraints on data streams called *Stream Schema*.

Since no agreement on data models and processing models for data stream systems exists [24], we present a Stream Schema model that is independent of a specific streaming system. As we will show, Stream Schema can be used with various existing processing models and systems.

1.1 Motivating Example

Linear Road is a popular benchmark for data stream management systems [7]. Notably, Linear Road specifies a schema for its benchmark, albeit in an informal way as no stream schema models were available. Linear Road describes a traffic management scenario in which the toll for a road system is computed based on the utilization of those roads and the presence of accidents. Both toll and accident information are reported to cars; an accident is only reported to cars which are potentially affected by the accident. Furthermore, the benchmark involves a stream of historic queries on account balances and total expenditures per day. The input data stream for Linear Road is constrained to be of only four types of tuples: Position Reports and three different types of historical query requests. Furthermore, position reports are associated to a specific vehicle and the reports for each vehicle are constrained to follow a specific pattern. This information (the constraints on the data within the stream) can be exploited to answer queries more efficiently. For example, it's possible to partition the input stream by type of the tuple. Some of the benchmark continuous queries only use data of a specific type, and we can optimize these queries by only running them on the partition (a subset of the stream) for which they are relevant. Furthermore, the position report stream can be partitioned along the vehicle ID and again processing can be divided, in this case a given query reporting statistics on a per vehicle basis can be run in parallel over each vehicle stream. Stream Schema not only describes this partitioning on the data, but also permits checking the queries (using dataflow analysis) to understand how they can be optimized using possible partitionings.

We will show that Stream Schema can specify these Linear Road constraints. This explicit specification of constraints will enable a stream system to automatically exploit this semantic knowledge in a number of ways for optimization. Some of these optimizations have been *hand-coded* by programmers in previous approaches. The benefit of Stream Schema is that it opens up the possibility of automatically applying these optimizations and systematically considering (and comparing) different possible optimizations within a stream optimizer. Notably, several implementations of Linear Road exist (from low-level to language-based), and we use the a high-level implementation with semantic windows [10] to check the applicability and the impact of optimizations enabled by Stream Schema.

1.2 Contributions

The main contributions of this work are:

- a formal model to describe static stream characteristics, called "Stream Schema";
- a discussion of how Stream Schema specification can be exploited in a wide range of stream processing models;
- the use of Stream Schema in static analysis of queries to simplify (or minimize) the queries;
- a suite of runtime optimizations enabled by Stream schema;
- a case study showing how the separation of queries from data constraints changes (and simplifies) how streaming query applications are modeled and implemented.

1.3 Structure of the paper

The rest of this paper is structured as as follows: After informally introducing the constraints in Section 2, a formal model to express and validate these constraints is established in Section 3. The possibilities of embedding this formal model into specific stream processing environments are evaluated in Section 4, followed by applications of Stream Schema in Section 5. Two cases studies (Sections 6, 7) and their relevant experiments show the applicability and benefits of Stream Schema on different models, workloads and implementations.

2 Stream Schema

In order to develop our model, we considered the following criteria. Stream schema constraints should be:

- useful in semantic checking of continuous queries;
- useful in specifying criteria that can permit new semantic query optimization;
- checkable with a tolerable overhead;
- specify static data behavior (and not depend on runtime behavior); and
- independent of a specific processing model, but compatible to the existing models.

Some existing proposals to exploit stream metadata fulfill some, but not all, of these requirements. To avoid overlap, we chose to not include the following types of constraints.

- Stream constraints based on specific query properties, the most common of which are types of window specifications. Such constraints can be specific to the processing model or language of a specific system and are orthogonal to properties of the data stream itself. In the optimization section, we look at certain cases how to exploit these properties in combination with Stream Schema.
- Constraints related to dynamic properties of a stream (e.g., arrival rates, delays). Such constraints are well-addressed in the literature [30], but again strongly dependent on the processing model.
- Constraints between streams ([22]) which can also depend on specific processing models, join algorithms and dynamic properties.

We will now introduce Stream Schema. Many of the constraints we introduce actually serve two purposes: 1) they specify a particular property that needs to hold on the data (and thus can be checked and used) 2) they provide structure (such as substreams or pattern instances), allowing the assignment of more specific constraints on these parts of the stream

2.1 Item Schema

A data stream is composed "items", "tuples" or "events". We will use the term item for the rest of this paper. A first step in stream schema is to describe the structure of these items; this is also the state-the-art for stream metadata description in existing data stream environments. Items can be specified in any data model, e.g., relational [4, 8, 6, 14, 16, 1] , XML [10], object-based models [2]. Considering the amount of prior work on schemas for items, and the divergence between the item data models, we only make that the assumption that one or multiple schema(s) for items exist and can be validated. We assume there are accessor functions (which we refer to as *attributes*, though of course they can be implemented as methods or XPath expressions) that can be used to access values in an item. As a simple example, for the linear road example, the position reports are have a relational item schema that contains the attributes: TIME, VID, SPD, XWay, LANE, DIR, SEG, POS. Item schemas can be used to optimize the storage of items and simplify predicates on the item values. We refer readers to the existing literature on how to do this [12, 23, 20, 28].

2.2 Partitionability

A data stream can often be partitioned into different substreams, based on different criteria. Two possible ways to perform this partitioning are considered in Stream Schema: 1) partitioning by item structure/item schema; and 2) partitioning by attribute value;

As an example of partitioning by item structure, the Linear Road's input stream is a combination of four different streams (one containing only position reports, and the other three called query streams). To support DSMS that can only handle streams with a single item schema, Linear Road defines a schema with the union of all attributes of all the different substream, fourteen in total, and leaving the attributes for not needed for a particular type as null. For example, in position reports the attributes Type, Time, VID, Spd, XWay, Lane, Dir, Seg, Pos are non-null. In contrast, in items from the first query stream Type, Time, VID, QID are non-null; on the remaining query streams other combinations of attributes are non-null. Implicitly, each of these substreams has its own item schema; in an DSMS that actually support heterogeneous item schemas, this could have been expressed explicitly using different item schema. Nonetheless, each of the substream has now this different item schema and possibly also other constraints, e.g. pattern or next-constraints.

As an example for partitioning by attribute values, the position reports stream can be partitioned based on the VID value. For optimizations, especially resource planning, the maximum number of such partitions may also be specified. For example, the schema may specify that there are at most $|VID|$ different vehicles (partitions) in the Linear Road position stream. In contrast to partitioning by item structure/schema, this partitioning generates uniform substreams.

Multiple alternative partitionings of the same stream are often possible. For example, we could split the position report no only by vehicle id, but also by highway (XWAY) and direction (DIR).

Partitioning is useful for both optimization purposes (to partition data and query plans) and for structuring: the overall stream might not be suitable to express additional constraints, but the partitioned stream might be. The vehicle trip pattern described below can only be specified once the general Linear Road stream has been partitioned into position reports and queries, and the position reports have further been partitioned by vehicle id (VID).

2.3 Patterns

In many cases, a stream (or a partition of stream) can be described as a well-defined sequence of items, e.g., a web session log could be expressed as *login browse* logout*. The name *pattern* has been established in the literature for such structures. For stream schema, we allow finite patterns that are repeating, possibly infinitely often to allow infinite streams. Pattern definitions may use predicates over attributes of an item. In the Linear Road stream, there are some patterns in the input streams, including the following taken from the specification [7].

> The simulator generates a set of vehicles, each of which completes at least one *vehicle trip*: a journey that begins at an entry ramp on some segment and finishes at an exit ramp on some segment on the same expressway.

The constraints in a pattern can be used to optimize pattern queries and semantic windows. As one example, pattern specifications in the query can be simplified by using knowledge of existing patterns in the data. In addition, pattern information can be used to “unblock” operators, i.e. indicating a blocking operator on infinite data that it can produce results (see Section 5.3).

2.4 Item value relationship

In many streams, item attribute values in different items have a well-defined relationship. For example, we many be able to define an ordering on attribute values. In certain streaming models (and systems), an ordering is hard-coded for timestamps, but ordering constraints among other attributes cannot be specified. Stream Schema the specification of ordering between attribute values in two adjacent items using a *next-constraint.* The scope of validity for such a constraint may be the whole stream, a a substream or a single repetition of a pattern. For example, the value of the TIME attribute in Linear Road is always non-decreasing, and this can be expressed with a next-constraint on the whole stream. In addition, in Linear Road, the position of a vehicle is non-decreasing or non-increasing (depending) while it stays on the same highway and direction. Such a constraint can again be specified by a next-constraint, but one whose scope is limited to a single pattern repetition (the pattern of a single vehicle on a highway). Such ordering constraints are useful for query optimization, for example, to unblocking operators (similar to punctuations[29]) or rewriting semantic windows or patterns, and also for semantic correctness checks (to determine if semantic windows close).

2.5 Disorderedness

Even though data stream models assume a total (or at least partial) order in the data stream, real-life data streams often do not conform to this assumption. For example, due to the impact of network transport or the lack of strict time synchronization between different sources, items may arrive out of order. K-constraints specify a limit on the disorderedness, and if they hold, some operations on (partially) disordered streams may be performed more efficiently [9]. In Stream Schema, we use as similar approach to express bounded disorder. This static description is an upper bound, in practice dynamic statistics may provide a more precise bound. In the Linear Road case there is no disorderedness given, but it could be easily envisioned that car position reports from different segments might be delayed by processing, thus creating disorder in the stream. Knowing a bound on the amount of disorderedness has been traditionally used to determine the size of a buffer required to restore the order, but more recent work takes disorder more into the account for specific operators and provides related optimizations [27, 26].

2.6 Combining constraints

The combination of the different types of constraints in stream schema results in a tree-like structure, as shown in Figure 1 for the Linear Road data stream. Reading the figure top-down, there is a next-constraints for non-decreasing time and a zero disorder constraint on the complete stream S This stream can be partitioned into four different substreams (P, $Q1$,$Q2$,$Q3$) for position reports and queries, based on the existence of a set of attribute values. Since the partitioning might place adjacent pairs of items into different partitions, the next-constraint will not always carry over the partitioning, thus a new next-constraint is needed to express the relationships after the partitioning. As we can see, this could be the same relationship (time non-decreasing), but different, more specific relationships might be possible (with the car position report stream L, in which each report for a specific car is exactly every 30 seconds). The position report stream P can further be partitioned by either VID values or XWay and DIR. After all these partitionings, the leaves of the tree contain item and pattern descriptions. In the case of the vehicle trip pattern L, there is a next constraint that only holds within a instance of the pattern: during a trip, a vehicle will stay on the same highway or direction, segments and positions will either be non-decreasing or non-increasing, depending on the direction.

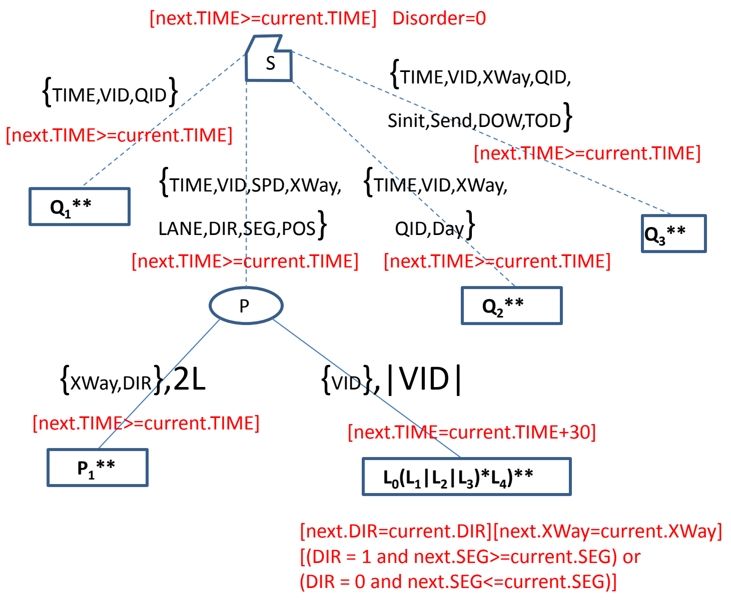
[next.TIME>=current.TIME] Disorder=0
S
{TIME,VID,QID}
[next.TIME>=current.TIME]
{TIME,VID,XWay,QID, Sinit,Send,DOW,TOD}
[next.TIME>=current.TIME]
Q1**
{TIME,VID,SPD,XWay, LANE,DIR,SEG,POS}
[next.TIME>=current.TIME]
{TIME,VID,XWay, QID,Day}
[next.TIME>=current.TIME]
Q3**
Q2**
P
{XWay,DIR},2L
[next.TIME>=current.TIME]
{VID},|VID|
[next.TIME=current.TIME+30]
P1**
L0(L1|L2|L3)*L4)**
[next.DIR=current.DIR][next.XWay=current.XWay]
[(DIR = 1 and next.SEG>=current.SEG) or
(DIR = 0 and next.SEG<=current.SEG)]

Figure 1: Linear Road Stream Schema

3 Stream Schema Formalization

In the previous section, we have introduced, by example, the constraints in Stream Schema. One of the challenges in defining Stream Schema is the fact that there is no established, agreed formalism to express data models and operations (see [24] for a recent preliminary approach to provide some unification). In order to serve a wide range of stream processing environments, we use a schema formalism that not make any assumptions on the specific processing model of the system, or on the item schema. Stream Schema captures the relevant stream constraints, and can be embedded into the processing models of existing stream/complex event processing systems.

3.1 Overview of formalization approach

We define a few simple assumptions about the data model (and item schemas) that are necessary to present our schema formalism. Based on this item definition, we then establish a formalism for streams, again with the necessary operations for validation. The stream constraints are expressed on that model as recursive data structure. Validation is formally described as prefix validation for this stream and schema definition, including a description of the runtime state needed to capture the evaluation of the prefix. A analysis of the space and time complexity completes the formal part.

3.2 Stream Model

Stream of Items A data stream is an *ordered sequence of items.* A stream can be either *finite* or *infinite.*

Given the heterogeneity in the data models for items (and their schemas), Stream Schema is designed to work with any item model that supports access functions (be they attributes, elements, XPath expressions, methods, etc.). To keep our notation simple, we will call such functions *attributes* and assume that each item schema has a set of attributes defined on it.

Item Schema. An item schema, IS, then is a set of attributes and optionally a set of properties that hold over the item schema. We make no assumption about the properties other than there is a function *validate* that given an item and a set of item schemas, it returns the matching schema (in schema type hierarchies like OO or XML schema the most precise match). As an example, the relational schema of position reports in the Linear Road Benchmark can be defined as an item schema P containing the set of attributes {TIME,VID,SPD,XWay,LANE,DIR,SEG,POS}.

3.3 Stream Constraints

We now present the constraints of Stream Schema.

3.3.1 Item Schema

This is the most basic constraint. Given an item schema IS, a stream satisfies the item schema constraint IS if every item in the stream validates against IS. Many stream processing already support item schema constraints.

3.3.2 Partitionability

Partitionability constraints have two forms:

- for partitioning by structure, let A be a finite set of attributes. Then we can partition a stream based on A into two different substreams, one in which all of A are present in every item, and a second in which this is not the case.
  For example, the input stream S in the Linear Road data is a union of four heterogeneous streams. We can partition S into four substreams ($\mathcal{P}$, $\mathcal{Q}1$, $\mathcal{Q}2$, $\mathcal{Q}3$) using the four attribute lists below.

  ```
  [P=(TIME,VID,SPD,XWay,   LANE,DIR,SEG,POS)]
  [Q2=(TIME,VID,XWay,QID,Day)]
  [Q3=(TIME,VID,XWay,QID,   S_init,S_end,DOW,TOD)]
  [Q1=(TIME,VID,QID)]
  ```

  A stream satisfies such a constraint if after all partitions the remainder stream is empty.
- for partitioning by attribute value, let F be a function defined on the items of the stream (a simple attribute or an expression over attributes). Then we can partition the stream by F, where two items i_1 and i_2 are in the same partition (substream) iff $F(i_1) = F(i_2)$. We can (optionally) specify a maximum number of values ($|F|$) for the function F indicating that any stream satisfying this constraints, there may be at most $|F|$ substreams (values of F). As an example, in the LR's position report stream, we can partition the stream by the VID attribute and specify a maximum number of vehicles $|VID|$. A stream satisfies the constraints if attribute values for all parameters of F are available for all items, and the number of partitions does not exceed the given limit.

3.3.3 Pattern and Repetitions

To model these patterns, we use basic regular expressions in which the variables are defined by constraints on the item schema of an item, and/or the attribute values of items.

For example, the vehicle trip pattern (of Linear Road) for a specific vehicle (with VID = vid) is described as follows

```
VT=(L0(L1|L2|L3)*L4)
```

where `L0`, `L1`, `L2`, `L3`, and `L4` are all defined as restrictions of the position report item schema P (recall P was defined above):

```
L0: validate(Item,IS*)=P and Item.Lane = 0
L1: validate(Item,IS*)=P and Item.Lane = 1
...
```

A special aspect of the pattern language of stream schema is the use of repetitions, especially at the level of pattern instances. Inside the pattern, only finite repetitions ($*$,?,+) are allowed, for repetitions of the whole pattern both finite and infinite ($**$) counts are allowed. By doing so, an infinite stream is expressed as a finite pattern (which can be validated more easily) that will repeat infinitely often. To enable reasoning (see constraints below), the following equivalence is defined $(A*) ** = A **$.

For example, the position report stream itself is defined by the pattern $S = (P) **$ in which P is the item schema defined above.

A stream satisfies the pattern constraint if an automaton representing the pattern accepts the item stream (prefix). Such an automaton can be created from the pattern specification by translation the finite part according to regular language/FSM equivalence; for the repetitions additional edges are added from accepting states to states reachable from starting state, carrying the the pattern starting symbols.

3.3.4 Next Constraint

These constraints are used to describe a relationship between consecutive items in a stream. They can be specified by binary comparison operators between *current* and *next* items. We use the notation

$$c(current.p, next.p)$$

in which p is an attribute name and c is comparison predicate defined over the domain of this attribute. As an example, in the Linear Road data, the time attribute is non-decreasing

$$[current.TIME <= next.TIME]$$

A stream satisfies a next constraint if for every consecutive pair of items i_1, i_2, $c(i_1, i_2)$ is true, for a next constraint at the pattern instance level, this condition only needs to hold for i_1,i_2 within the same pattern instance.

3.3.5 Disorderedness

To describe disorderedness, we use an approach that is similar to K-constraints [9]. Specifically, a parameter (positive integer) K is given, indicating that the data may be up to K positions out of order. Stream Schema does not specify an ordering relation, but allows validation approaches with and without a given ordering relation. In the presence of an total order relation over particular attribute values (e.g. timestamp order), a stream is valid if a ordered sequence can be generated by sorting inside sliding window of size K. If no such ordering relation is given, a stream is valid if any of the permutations fulfills all other constraints.

3.3.6 Combining Constraints

Schema constraints (and partitioning) can be nested and expressed in a natural fixed point arithmetic (see the section A of the appendix for more details). In Figure 1, we show the result of our running Linear Road example, expressed in Stream Schema.

Each node of this tree carries a set of next-constraints, and a set of attributes in the item schema of the node. Interior nodes of the tree contain a set of partitioning conditions. Leaves hold a pattern description and a set of next-constraints that hold within a repetition of the pattern. The root node also stores the name of the schema and the disorder parameter.

3.4 Stream Validation

Since any newly arriving item could violate a given schema, complete validation of an infinite stream is not possible. Therefore the mechanism of stream validation is based on validating the current item using the prefix validation result and prefix validation state. Since disorderedness is orthogonal to all aspects of validation, we first define the validation algorithm for ordered stream data, and then extend the definition and analysis for disordered streams.

In order to not store the complete prefix, we define a special data structure that captures only the information necessary for validation. This data structure is also recursive, and mirrors the structure of a stream closely:

- Next-constraints: for all attributes of the constraint, we store the previous value. For pattern-repeating next constraints, values are reset at the end of a pattern instance.
- Partitioning: we define a recursive data structure for the nested stream data; in addition we need to store the information on the partition decision. The number of total substreams can be derived from the number nested states.
- pattern automaton state for a single repetition of pattern, e.g. all active states in a NFA.

Using the prefix validation result, the prefix validation state and a stream schema, the arrival of a new item produces a new validation result and state. The validation is performed by checking the item schema, the next constraints, and then either checking the pattern (leaves) or the partition constraint and then recursively the nested substream definition(s).

Based on the formalization sketched here, a number of properties on the complexity of stream schema validation can be established:

- the space needed for Schema validation is finite, if the set of recursively nested schema definitions is finite, regardless if the validated stream is finite or infinite
- the cost of checking a new item without recursion is polynomial
- the cost of checking a new item with recursion is $O(n^m)$, where m is the nesting depth.

Relevant proofs are provided in subsection B.1 and B.2 of the appendix.

Checking with disorderedness requires additional overhead in terms of space and computational complexity. We define two variants how this validation can be performed: 1) with a known ordering relation (as parameter to validation, expressed like a next-constraint) 2) with no known ordering relation.

The first variant can be implemented by checking/restoring order according to ordering relation, then performing the ordered variant of validation. The additional space required is linear to K, cost to is determined by sorting within sliding window of K. For the second variant, the arrival position of items in the stream needs to be kept to work over partitions. To perform the validation, all permutations allowed by K need to be generated in order to check if at least one matches all constraints specified in the schema, and this enumeration needs to be performed for each newly arriving item. To check next and partitioning constraints, it is sufficient to keep K values around, and the effort for enumeration is $K!$. For pattern specifications, the situation is more complicated, since the permutations might affect the whole pattern instance, thus requiring state to be kept for the full instance of the pattern including all the k-permutations. This is similar to what is discussed in [27].

4 Integrating Stream Schema into Processing Models

The next step after defining stream schema embedding it into the concrete data and processing models of data stream management systems (DSMS). Each of these systems uses a somewhat different model, but for most of the systems we have evaluated, a straightforward integration is possible.

To perform this embedding, the following steps are needed: 1) The abstract *items*, *item schemas* and item operations (which we have called attributes) and *validate* are mapped to their concrete counterpart in the DSMS. 2) The respective *stream data model* needs to be checked on their compatibility. 3) *Implicit schema constraints* of the DSMS need be expressed in Stream Schema. 4) *Existing Schema-like capabilities* of the DSMS need to be checked against the capabilities of Stream Schema. In relational cases (Aurora/Streambase [4], CQL [8], the SQL pattern extension [6], Gigascope [14], Cayuga [16], CCL [1]), a stream of homogeneous items is used, where *items* are flat relational tuples, accessible by attribute name. For query streaming [10], a stream can be heterogeneous (different items validate against different XML schema definitions), where items are atomic values or XML nodes, accessible by XPath expressions. SASE [5] and ESPER [2] also use heterogeneous streams, with flexible item-schema models and access paths. All these item-oriented aspects cleanly map to our formal model. For the *stream data model*, most models assume a totally ordered sequence of items as basis (again corresponding to our formalization), with some relaxations to this ordering: CQL uses a sequences of batches [8] as it stream model, in which the stream has a partial ordering on a timestamp value and the items with the same timestamp do not have any order among them. Other approaches use k-constraints [9] or Slack (Aurora) to give a bound to the degree of out-of-orderness. This again matches the definition in Stream Schema. Many Stream Processing Models define *implicit timestamp attributes*. In Stream Schema, these implicit constraints can be expressed by an extension of the item schema (when an item is defined) and next constraints capturing a relation between timestamps (e.g., strictly increasing or increasing with equal values allowed). Since some systems (SASE, Esper, XQuery) do not define timestamp, we chose not make timestamps part of Stream Schema, but to automatically extend the set of Stream Schema constraints when embedding Stream Schema into processing models defining timestamps. While most DSMS provide *schema-like definitions*, the stream aspects of these schemas are usually restricted to some ordering properties and several dynamic properties (e.g. arrival delays). A somewhat closer match is the possibility to define a stream/schema by a query (Esper,Coral8), where the query specification (filter, pattern) would imply similar schema constraints as our Stream Schema, but without the possibility to provide structure. *StreamBase/Aurora* provides a schema-like operator input specification $(OnA, slack, GroupByB_1, B_N)$, expressing an order on A, limited disorder *slack* and the possibility to group by the attributes specified in a *Group By*. All of these constructs can be mapped to stream schema. *Gigascope* uses a specification of ordered attributes (representing timestamps, sequence numbers etc.) in three different types: 1. Strictly/monotonically increasing/decreasing, 2. Monotone non repeating 3. Increasing in group: Ordering 1 and 3 are expressible as next-constraint (with partitioning in 3). The precise definition of 2) cannot be derived from the informal definition in the Gigascope paper; it is therefore not clear if such a specification can be validated.

5 Application of Stream Schema

Similar to the interaction of XML Schema with the processing model of XQuery, there are four kinds interaction of stream schema with a DSMS:

1. **Validation and (type) annotation** of the data stream, including reactions on *invalid* data.
2. **Changing Query semantics**, such as allowing queries to run that would not run in the absence of schema or statically determining that a query will produce not results.
3. **Enabling Optimizations** on query plans and query processing, such as reducing computational cost or response time.
4. **Extending Modeling** streaming applications by separating constraints from queries.

5.1 Stream Validation

In many use cases, explicit validation of streams is not needed, since the stream constraints are guaranteed to hold by the producing source. Similarly, in a distributed stream processing settings, only untrusted data needs to be validated, which may be only a portion of all of all streams used. Nonetheless, for situations in which validation is required (e.g. untrusted input), it should be possible to perform it without significant overhead. Indeed, we have designed Stream Schema with this goal in mind. The formal definition shows that this is possible, with two elegant options for implementation:

5.1.1 Validation using an Existing Validation Framework

We implemented a large part of stream schema based on the Xerces XML parser and validator [3], since it already provides most of the operations and data structures needed for Stream Schema validation. We extended XML schema with the relevant stream constraints, and changed the XML parser so that it can consume a root sequence instead of a root element. Each item in this sequence is first validated against the set of item schema using the standard XML schema validation mechanisms, providing type information for this item; the existing operators in Xerces were then re-used to express the stream constraints. The current implementation does not support checking nested schema definitions yet; however were will add this capability soon. In terms of validation cost, we expect parsing and item validation to dominate the cost for most scenarios, followed by pattern validation.

5.1.2 Validation using Continuous Queries

As an alternative, Stream Schema can be translated to a continuous query, since the operations required for stream schema validation match closely the set of commonly available expressions/operators in DSMS (and CEP or Complex Event Processing systems). If a matching operator is not available, system would also not benefit from the optimizations in this area (e.g. Aurora does not have pattern matching, so checking and using pattern information does not provide benefit). For such systems, a subset of Stream Schema, without patterns can still provide benefits.

When a validation failure is detected in a stream, we may terminate processing of the stream. Such an approach might not always be desirable, as it limits the ability of a DSMS to deal with unexpected data. One possible alternative is to treat validation as a normal stage in query processing (just as pattern matching) and allow the programmer to capture failures and also drop possible optimizations (in an on-line way) that are based on schema constraints that are not satisfied. Alternatively, the DSMS could relax the schema in the face of data violating a constraint.

5.2 Impact On Stream Processing Semantics

The presence of stream schema can have a profound effect on the semantics of operations

5.2.1 Static check for non-executable expressions

- Warn/Abort the execution of non-executable predicate-based windows with aggregates. For example, if the *end* condition of defined windows over a stream of book information items is <bibliography> element but according to the schema, the <bibliography> is an optional element for book items, a system can generate a warning (based on static analysis of the query) that that there may be some open windows which may never closed.

- Warn/Abort the execution of blocking operators. For example, if a blocking operator (e.g., a sort) is used over a streams and from an analysis of the schema a system can determine that the execution may be infinite, a system can abort the operation
- Warn/Abort about empty results. For example, if the pattern query $AC+B$ is applied over a stream with the schema of $(AB)*$, a 'no-result' warning could be generated.

5.2.2 Extended Set of Runnable expressions

A system may be able to change a blocking operator into a non-blocking one (and in doing so, make the operator *runnable*), if the stream satisfies certain schema constraints. As a simple example, a blocking sort operator may be removed from a query plan, if the stream is known to comply to a schema that guarantees the same sorted order.

5.3 Optimizations

The constraints provided by Stream Schema are applicable to a large range of operators and expressions. In the scope of this paper, we focus on optimizations based on the *stream* aspects of Stream Schema; optimizations based item schema specifications are similar to existing schema-based optimizations [12, 23, 20, 28] and will not be discussed here (Some techniques and relevant examples are provided in section C of the appendix).

Stream Schema provides the metadata to perform optimizations, so an important class of optimizations enabled by stream schema are not *new* in a strict sense, but in fact well-understood in terms of their mechanism and benefit, e.g. rewriting a window type from sliding to tumbling in order to use a simpler evaluation mechanism with less CPU and memory cost[10]. The important contribution of stream schema is to formally express if an optimization is applicable.

In this section, we will therefore show an overview on the classes of optimizations in which stream schema is beneficial Where necessary, we will point to related work for the detailed benefits of existing, but newly enabled optimizations. In the two case studies, we will discuss selected optimizations in more detail and show the benefits experimentally.

5.3.1 Pipelined Execution

When strictly following the definition of semantics windows, e.g. FORSEQ, such a window would be a pipeline breaker: the items bound by such a window can only be processed when the end condition has been successfully evaluated. For many streaming applications, this behavior is undesirable, since all following operations (such as aggregations) can only be started after the window has been closed, and thus an additional amount of latency and memory consumption is incurred. In addition to other preconditions purely decidable at the language level, two important conditions need to be fulfilled in these cases: 1) every open window will be closed at some point, 2) windows have a total order; they will be closed in same order they were opened. A detailed analysis on this optimization is given in the LR case study.

5.3.2 Stream Data Partitioning

The volume of data that needs to be processed in real time for data stream management systems can easily exceed the resources available on a centralized server. An well known approach taken by Distributed DSMSs to tackle this problem is data stream partitioning which is splitting resource-intensive query nodes into multiple nodes working on subset of data feed [13].

A recent work to partition a stream query workload is based on the query side and includes two steps [25]: 1) finding a partitioning scheme which is compatible with the queries 2) using this scheme to transform an unoptimized query plan into a semantically equivalent query plan that takes advantage of existing partitions.

The unspoken assumption of that work is that the data (not just the queries) is actually partionable by of the schemes produced by 1). The partioning information of Stream Schema information can now be used to determine which (if any) of these schemes can actually be used, thus completing this work. In the first case study, we will show how this optimization technique can improve the performance of the LR implementation.

5.3.3 Window/Pattern Optimizations

- Simplifying the overlap specification of window/pattern instances: Only if new window/pattern starts if the previous has been completed. To be more concrete, here are two examples from different frameworks:
  - `Sliding` and specially `Landmark` windows are more expensive compared to `Tumbling` window, since the number of open windows is potentially much higher, and more checking is needed. Using the information in Stream Schema, a query written using landmark or sliding windows can be rewritten into tumbling window. An example of this rewriting is given in the LR case study later on.
  - One of the constructs SQL pattern extension[6] is the `SKIP To` which determines where we should start looking for the next match once the current one has been completed. Two common options are `NEXT ROW` and `PAST LAST ROW` meaning we start looking for the next match from the row after next row or last row of the current pattern. Using the pattern constraint over the stream specified, one can rewrite the query to have more efficient `SKIP TO` options. For example, if the Stream Schema defines a stream as repetition of the pattern `AB*C` and the query matches the pattern `AC`, we can safely replace the

value of the `SKIP To` option with `PAST LAST ROW`. The new query is semantically equivalent with original one but cheaper because avoids starting unsuccessful matches.

- Removing existing structure from window/pattern specification to only check what schema does not already provide. An example below for pattern matching systems:
  - Complex event processing (CEP) systems usually use Finite State Machines (FSMs) for pattern matching [16, 5, 18]. Using the information of the patterns already present in the stream, it is possible to simplify the query FSM by fully or partly decomposing it, a technique commonly used in areas [17]. Such decompositions can improve the performance of the CEP systems by reducing number of states or relaxing the transition rules. For example, if we know that incoming items already comply with some patterns in the stream's schema, rechecking these sub-patterns is unnecessary. Figure2 depicts this optimization for the query `ABABA` over the stream `(AB)*`.

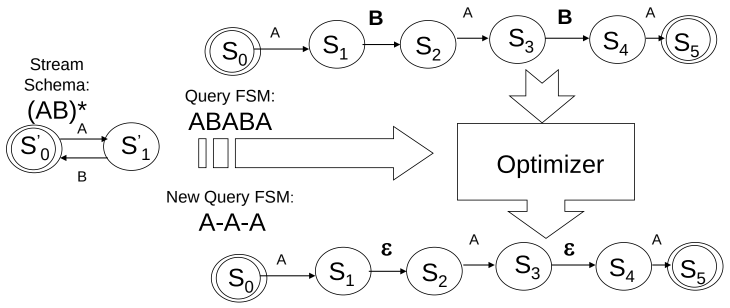

Figure 2: Pattern FSM decompositon

5.3.4 State Reduction

A lot of stream operators (i.e join, group-by, and sort), maintain some states in order to generate the correct results [11]. Most of the optimization technique for stream processing aim at reducing number of maintained states to minimize the memory/disk cost. These techniques in general fall into one of these two categories: 1) avoid keeping items at all (discard as early as possible) 2) purge state after certain evaluation steps.

Avoiding materialization In the stream processing system, newly arriving items are fed into the open windows and they will have their contribution to output results. If there is a way that we can make sure that an incoming item will not contribute to the output result, we can safely drop that item avoiding unnecessary resource allocation. This technique is similar to the semantic load shedding [15] in which the query optimizer intelligently drops some tuples in order to minimize the error of the output of the query, but it does introduce errors at all. Stream schema can help in different ways to ease making such decision. Following are some examples for a stream join operator:

- difference in key size: if two streams involved in the join have the partitionability constraint on the corresponding attributes, and the key set is not the same, one can drop the items with the missing values from either of the streams.
- if any of the streams involved in the join is heterogeneous and some of its item schemas do not include the join attribute, they can be eliminated
- mismatch between next constraints (only items which comply with both sides' next constraints have the chance to successfully participate in a join)

State Purging In some cases, stream schema constraints allow this operators to purge number of its states. For example, the join operator needs to keep track of number of items, as there might be matches for them in the future, if based on for example a monotonic next constraint in the stream schema, one can make sure that the such item would never show up, the join processor can purge the state it has been keeping for those items[19].

5.4 Modeling Streaming Query Applications

The optimizations provided by stream schema can be used to simplify modeling and developing streaming query applications. Looking at the state of the art, one can conclude that streaming queries are written in a a very explicit manner: all possibly relevant predicates and expressions are directly expressed in the query (to ensure correctness), and also often manually arranged in order (to achieve good performance). By doing so, predicates from two domains are mixed: 1) predicates to describe the desired behavior 2) predicates to capture and utilize environmental constraints.

The query rewrites enabled by stream schema lead to a different approach: Queries can be written a high level of abstraction, while the environmental constraints are expressed in stream schema.

This separation of query and structural constraints allows for a significant improvements in the way how streaming applications can be developed:

- Simpler queries: Express only what actually be seen/not seen in the stream, re-use for multiple environments
- Simpler "environment constraints": Express constraints once in Stream Schema, re-use for multiple queries
- Separation of development for queries and schema.

An example of such a change in modeling (including the necessary optimizations and rewrites) is discussed in Section 7.

6 Case Study I: Linear Road

To check the expressiveness of our schema proposal and determine the usefulness of its applications in stream processing, we used the Linear Road Benchmark [7] implementation in Continuous XQuery [10]. Continuous XQuery is an interesting target for Stream Schema, since its data model does not have any stream-oriented implicit constraints, it uses semantic windows, and allows arbitrary nesting of expressions. The schema for Linear Road has already been given in Figure 1, so we will omit it here. Currently, no optimization framework for a data stream system is known to exists, so the optimizations are discussed at a formal level and implemented by manually adapting the queries.

6.1 Existing MXQuery Implementation

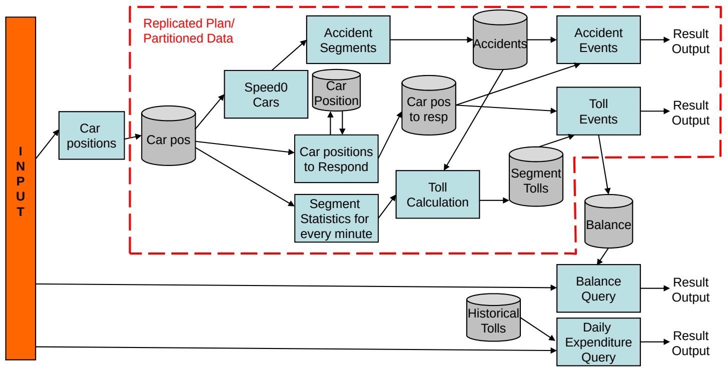

Figure 3: MXQuery Linear Road Implementation, extended with data partitioning information

The Linear Road implementation of MXQuery [10] uses a combination of continuous XQuery expressions and dedicated stream stores to express the streaming queries, as shown in Figure 3. In total, 7 threads were used, 4 driving the output stream, 3 for intermediate results.

6.2 Schema-driven executability of XQuery expressions

Since continuous XQuery uses predicate-based windows, it is in many cases non-trivial to determine statically if an open window will ever be closed or not. This is important, since the output of the window is often consumed by blocking operators, e.g. aggregations. Using the Stream Schema for LR, this can be achieved. For example, the following window expression is part of the query finds cars with speed equal to zero[1]:

[1]The full query and the analysis for the other queries is given in the section D of the appendix

```
   forseq $w in $ReportedCarPos sliding window
start curItem $s_curr, prevItem $s_prev
when $s_curr/@minute ne $s_prev/@minute
end curItem $e_curr, nextItem $e_next
when $e_curr/@minute ne $e_next/@minute
```

This window will stay open as long as the incoming reports have the same value for the minute attribute, thus we need to prove that minutes changes in order to show that the window will close.

Proof The maximum number of reports in a particular minute is $2 * |VID|$, since every vehicle emits two reports per minute. The next report emission (regardless of the source vehicle) belongs to the next minute, closing the window[2].

The argument for other windowing queries is analogous (with different upper bounds).

6.3 Query Rewrites for Pipelining Execution

As laid in the optimization section, pipelined window execution is an important factor to reduce latency and memory consumption. For semantic windows, a important precondition is that windows will be closed at the same order that they were opened, since otherwise the execution would be blocked until the "correct" window is ready. For sliding and landmark windows additional information is required which can be derived from the stream schema. For example, in the continuous query for accident detection for the segments, the FORSEQ part looks as follows:

```
    forseq $w in $ReportedCarPos
sliding window
start curItem $s_curr, prevItem $s_prev
when $s_curr/@minute ne $s_prev/@minute
end curItem $e_curr, nextItem $e_next
when ($s_curr/@minute +2) eq ($e_next/@minute)
```

By using two lemmas, we show that windows are ordered and hence we can pipeline the results.

Lemma 1. Windows will be opened in a strictly increasing time order.

Proof. Assuming (strictly) increasing time attributes, for every incoming item a new window is opened; these windows are strictly ordered with respect to their first element's time (proof by induction over pair of consecutive windows). By relaxing this assumption to non-decreasing time, only a single window is opened for all items with the same time values (the first), maintaining the desired property. □

Lemma 2. Windows will be closed in the same order as they were opened.

Proof. We prove this by contradiction. Assume we have two arbitrary windows w and $w^{'}$ in which w was opened before $w^{'}$ meaning

[2]Changing the unit of the time attribute from *seconds* to *minutes* (in `minute = ⌈ time/60 ⌉`) preserves the non-decreasing property specified in the schema.

$\texttt{\$s\_curr}_w\texttt{/@minute < \$s\_curr}_{w'}\texttt{/@minute}$

now we show it's impossible if w would be closed after $w^{'}$, meaning

$\texttt{\$e\_next}_w\texttt{/@minute > \$e\_next}_{w'}\texttt{/@minute}$

but this is contradictory, since we know

$\texttt{\$s\_curr}_w\texttt{/@minute + 2 = \$e\_next}_w\texttt{/@minute}$
$\texttt{\$s\_curr}_{w'}\texttt{/@minute + 2 = \$e\_next}_{w'}\texttt{/@minute}$

□

We can make a similar proofs other continuous queries in the implementation, details are again given in section E of the appendix.

6.4 Data Partitioning

As we described before, the position report stream of LR benchmark can be considered as combination of multiple position report sequences from different expressways and different directions. Depending on the nature of the continuous queries over LR input stream, it might be possible to partition this stream along XWay and DIR dimension and process them independently and in parallel.

Regarding the LR continuous queries, the Account Balance query is the only one which does the computations over more than one expressway or direction. Therefore, we can easily parallelize the execution of the other queries as follow:

- *Accident Detection*: the query 'Accident Segment' uses a group-by and stream keys are part of the grouping predicates, so this is trivially parallelizable. *Toll Calculation* is analogous
- *Accident Notification*: the query 'Accident Events' is in charge of these notifications and for each incoming position report -which has fulfilled the notification preconditions- retrieves the accidents for the *same* expressway and the *same* direction and then notifies the vehicle about accidents in his neighbor segments (if any). *Toll Notification* is analogous.

6.5 Experimental Setup

In order to validate the optimizations spelled out in the case study, we re-created the experimental setup given in [10]: All experiments were run on dual-CPU AMD system with single-core (pipelining experiment) and dual-core (partitioning experiment) Opteron 2.2 GHz processors and 6 GB RAM. A Sun 1.6_10 64-bit JVM with a heap size of 3 GB respectively 5 GB was used. Since the queries used in the provided setup were carefully tuned and hinted to take advantage of the implicit schema knowledge, we created a baseline using semantically equivalent queries that would not use schema knowledge.

6.6 Pipelining Experiment

For Linear Road, most window constraints are on minutes, and the resulting computations on the window contents to compute statistics, accidents and tolls all need to be performed at minute changes. As a results, without pipelining the response time requirements of 5 seconds is violated at these minute changes, even though unused processing capacity is available during a minute. Schema information can be used to enable pipelining in window processing (see Section 6.3), and thus alleviate the issue. In the experiments, this effect was clearly visible: While running the queries without the schema information (and thus without pipelining) only allowed a scaling factor L=2.5, using schema improved the results to L=3.5.

6.7 Stream Partitioning Experiment

A second experiment is geared toward partitioning the stream in order to parallelize the processing. When the results of a query or a set of queries can be computed relying only a partition of the key value set, the workload can be distributed over multiple cores, system or data centers. As determined in Section 6.4, the workload of linear road can be partitioned along the XWay and Direction attributes. Since the level of parallelism present in the original setup was only enough saturate 2 cores on the experimental platform (and 4 cores being available, the stream and the query plan were split into two substreams with the equivalent query plans, sharing only the balance store. On this 4-core machines, L=5.0 was reached with partitioning, while L=6.0 was missed, since the maximum observed response time was 8 seconds.

7 Case Study II: Supply Chain

The second case study focuses on application of Stream Schema's pattern specification feature in query optimization and application design. It addresses the problem of detecting misrouted items within an RFID-driven supply chain.

7.1 Item Distribution and Tracking

A typical supply chain system (e.g. in a car factory) attaches RFID tags to items (e.g. car parts) in order to track their distribution from the entrance gate towards their specified destinations, e.g. assembly line depots based on their types. For example all items of type 'gearbox' go to destination number 6. For each item, there is a path from the entrance to its destination which includes a number of RFID readers, also the entrance and the destinations. These RFID readers express the distributions a tree, as depicted in Figure 4.

A common query is the detection of misrouted items. For example if an item has ended up at destination 8, but it was supposed to be routed to another destination it must be detected and reported. A possible, high-level way of solving this issues is to specify a pattern query at each destination that checks on the item type. For example, as the pattern AB*C in which A is the entrance reader, B is any intermediary reader, and C is the destination reader.

With the help of metadata the performance of such queries by improved. The idea is that detecting a misrouted item is in many cases possible before reaching the actual destination. Given the readers tree in figure 4, instead of doing the item type checking at leaves [R_6, R_8, R_{10}, R_{14}], the check can be done after branches [R_4, R_7, R_9, and R_{11} respectively]. The structure of this readers tree can be provided by stream schema to the stream engine resulting in a structure-aware item tracking.

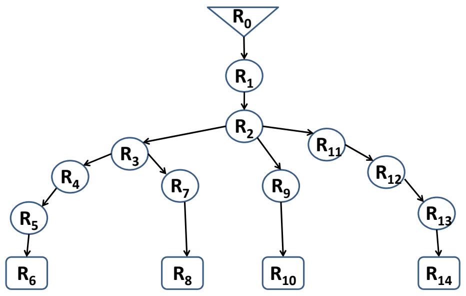

Figure 4: Supply Chain with RFID readers

7.2 RFID Readings Schema

Having specified item schema for the RFID readings as below

```
R(ReaderID,TagID,ItemType,TIME)
```

the main stream of RFID reading is depicted in figure 5, stating that items in this stream are homogeneous and in a non-decreasing fashion over their TIME attribute (without any disorderedness). Moreover, it can be partitioned along the TagID attribute. Each partition corresponds to a particular TagID (tagid) and has a finite number of readings[3] and these readings comply with a pattern shown at the leaf. In this pattern, T_i is defined as

T_i: `validate(Item,IS*)=R and Item.ReaderID =` R_i

For each partition, TIME values are strictly increasing since a particular item is sensed by only one reader at any point in time

[3]At most depth of the readers tree

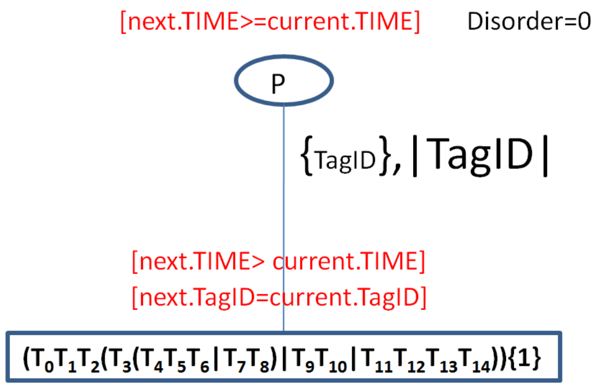
[next.TIME>=current.TIME]
Disorder=0
P
{TagID}, |TagID|
[next.TIME> current.TIME]
[next.TagID=current.TagID]
$(T_0T_1T_2(T_3(T_4T_5T_6|T_7T_8)|T_9T_{10}|T_{11}T_{12}T_{13}T_{14})){1}$

Figure 5: RFID Readings Stream Schema

7.3 DejaVu

We have used DejaVu [18] for this use case. DejaVu is a declarative Pattern Matching System over live and archived streams. It's built on MySQL open source database system and extends MySQL language with MATCH_RECOGNIZE clause [6] which defines pattern with semantic windows. The DejaVu implementation uses finite state machines for the execution and internal representation of the pattern queries.

7.4 Query Rewrites for Early Decision Making

Assume the rightmost leaf of the readers tree in figure 4 (R14) is the depot for 'engines', the pattern query for detecting misrouted items to this depot is

```
SELECT InitialS, EngineS, RoutingTime, MatchNo
FROM Readings
  MATCH_RECOGNIZE (
    PARTITION BY TagID
    MEASURES A.ReaderID AS InitialS,
             C.ReaderID AS EngineS,
             C.Timestamp - A.Timestamp AS RoutingTime,
     MATCH_NUMBER AS MatchNo
    AFTER MATCH SKIP PAST LAST ROW
    ALL MATCH
    PATTERN(A B* C)
    DEFINE A AS (A.ReaderID  = "R0")
           B AS (B.ReaderID != "R14")
           C AS (C.ReaderID  = "R14" AND
           C.ItemType != "engine")
);
```

Having knowledge of the readers tree structure, it's possible to optimize the query by replacing R14 with R11.

7.5 Experiment Setup

In our experiments, we used the query described in the previous section. The length of a path was fixed at 30. We have generated readings for 1000 Tags which end up in the 'engine' leaf. Misrouting probability is set to be 0.02. In our experiments we have changed the branching position (position where the ancestor of the 'engine' leaf has sibling). The open source memory measurement tool Valgrind has been used for monitoring the memory usage.

7.6 Early Decision Making Results

Here, we measure the memory consumed by the windows which mostly maintain partial matches. Early decision making allows us close the windows as soon as possible which means less memory consumption. As the result in the table shows the closer we come to the root we branch, the more memory we save off the baseline 1494 KB, which was measured without using schema knowledge.

Branching Pos.	Memory (KB)	Saving (%)
2	8	99.4
5	189	87.3
15	711	52.4
25	1234	17.4

8 Related Work

Besides the pre-existing schema-like constraints in current DSMS, which we discussed in Section 4, there is wide range of related research:

Tucker et al [29] describe in their work how to use *punctuation semantics* to optimize query operators: Every operator, that gets a punctuation over the stream instead of a normal element, knows that no more elements matching that punctuation will arrive. With this information it can unblock, output some partial result or reduce its state.

Punctuations are well-suited for cases which we have dynamic behavior and as such orthogonal to our work.

Golab et al [22] have extended the SQL DDL to define three stream integrity constraints (Stream Key , Foreign Stream Key, and Co-occurrence), which are defined across streams for particular time windows. The main goal of these constraints is to reduce the cost of join between streams by query transformations such as 'join elimination' and 'anti-join elimination'[22].

These constraints are again orthogonal to our work, since we have focused on detailed description of single streams (and possibly treat them as a union of substream sequences)

Research on schema-based optimizations dates back into the early '80s, recent work on relational [12] and object-oriented [23] query languages focuses on goals like predicate addition and removal, join removal and empty result detections. For XPath/XQuery, there has been work not only work for persistent

XML [20], but also for streaming XML[28]. These approaches are a subset of what is expressible in Stream Schema, since they focus on the contents a single item or document, not a possibly infinite stream.

9 Conclusions and Future Work

Describing the static properties of a data stream using stream schema opens up an important direction toward efficient declarative stream processing. Existing approaches to use dynamic properties and statistics of streams are complemented by this new information. Both lines of work provide important foundational results necessary to the development of systematic cost-based optimizers for stream data management.

Stream Schema provides many avenues for future work. Additional optimizations based on Stream Schema should be investigated, possibly also leading to additional stream constraints. An integration of *window constraints* can provide more information for stream joins and capture schema change over time, but there are semantic issues and processing model dependencies. As a related matter, further investigation is needed into alternative ways to react to violations of constraints within a stream.

Furthermore, investigation into methods for generating Stream Schema holds a lot of promises. Besides manually designing Stream Schemas using informal applications semantics, we see three approaches: 1) stream mining: pattern mining and subgroup mining well-defined problems [21], so existing algorithms in this area can be used to create Stream Schema; 2) other formal descriptions of systems can be translated into Stream Schema, such as business process descriptions or workflows; and 3) the output of continuous queries also implies a schema with particular constraints (which could be automatically inferred and used in subsequent processing). Each of these three approaches provides promising directions for future work.

References

[1] Coral8 CCL Reference.

[2] Esper reference documentation 3.0.0.

[3] Xerces2 Java Parser Project Homepage.

[4] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2), August 2003.

[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient Pattern Matching over Event Streams. In *SIGMOD*, 2008.

[6] Anonymous. Pattern Matching in Sequences of Rows. *SQL Standard Change Proposal*, 2007.

[7] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.

[8] B. Babcock et al. Models and Issues in Data Stream Systems. In *PODS*, 2002.

[9] S. Babu, U. Srivastava, and J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries Over Data Streams. *TODS*, 29(3), 2004.

[10] I. Botan et al. Extending XQuery with Window Functions. In *VLDB*, 2007.

[11] I. Botan et al. Flexible and Scalable Storage Management for Data-Intensive Stream Processing. In *EDBT*, 2009.

[12] Q. Cheng et al. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB*, 1999.

[13] M. Cherniack et al. Scalable Distributed Stream Processing, 2003.

[14] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a Stream Database for Network Applications. In *SIGMOD*, 2003.

[15] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing over Data Streams. In *SIGMOD*, 2003.

[16] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards Expressive Publish/Subscribe Systems. In *EDBT*, 2006.

[17] S. Devadas and A. R. Newton. Decomposition and Factorization of Sequential Finite State Machines. *IEEE Trans. Computer-Aided Design*, 8(11), 1989.

[18] N. Dindar et al. DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events. In *SIGMOD*, 2009.

[19] L. Ding, E. A. Rundensteiner, and G. T. Heineman. MJoin: A Metadata-Aware Stream Join Operator. In *DEBS*, 2003.

[20] D. Florescu et al. The BEA Streaming XQuery Processor. *VLDB Journal*, 2004.

[21] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining Data Streams: A Review. *SIGMOD Record*, 34(2), 2005.

[22] L. Golab et al. Optimizing Away Joins on Data Streams. In *SSPS '08: 2nd international workshop on Scalable stream processing system*, 2008.

[23] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic Query Optimization for Object Databases. *ICDE*, 1997.

[24] N. Jain et al. Towards a Streaming SQL Standard. *VLDB*, 2008.

[25] T. Johnson et al. Query-Aware Partitioning for Monitoring Massive Network Data Streams. In *ICDE*, 2008.

[26] J. Li et al. Out-of-Order Processing: a New Architecture for High-Performance Stream Systems. In *VLDB*, 2008.

[27] M. Liu et al. Sequence Pattern Query Processing over Out-of-Order Event Streams. In *ICDE*, 2009.

[28] H. Su, E. A. Rundensteiner, and M. Mani. Semantic Query Optimization for XQuery over XML streams. In *VLDB*, 2005.

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

[30] S. D. Viglas and J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *SIGMOD*, 2002.

Appendices

A Formal Stream Schema Constraint Combination

We define a Stream schema M as a combination of a name and a definition:

$$\mathcal{M} = \mathcal{N} \times \mathcal{D}$$

We furthermore define

$$F(\chi) = (\mathcal{C} \times \mathcal{P})^* \times (\mathcal{T}_{NFA} \cup (\mathcal{P}^* \times \mathbb{N} \times \chi)^*)$$

F defines a set of next-constraints, and either a pattern definition or a partitioning and a nested stream definition. (N.B. For notational simplicity, we use $A^* = \bigcup_{n \in \mathbb{N}} A^n$.)
Using F, Def can now be defined as the smallest set so that $Def = F(Def)$, in an explicit form $Def = \bigcup_{n \in \mathbb{N}} F^n(\varnothing)$
For convenience, we define the following accessors on Def

- $c_i : Def \to \mathcal{C}$: i-th next-constraint, comparison function
- $p_i : Def \to \mathcal{P}$: i-th next-constraint, access path function
- $type : Def \to \{'pattern', 'partition'\}$: Does $\mathcal{M}$ specify a pattern or a partitioning
- $t : Def \to \mathcal{T}_{NFA}$: Pattern, if $type(Def) =' pattern'$
- $q_{k,l} : Def \to \mathcal{P}$: k-th partitioning, l-th access path/key, if $typeDef =' partition'$
- $n_k : Def \to \mathbb{N}$: k-th partition, key space definition, if $typeDef =' partition'$
- $d_k : Def \to Def$: nested Schema definition for the k-th partitioning, if $typeDef =' partition'$

B Formal Stream Schema Validation

B.1 Validation Mechanism

To do so, we need to establish the contents and the structure of the runtime state. It mirrors the schema definitions, since each definition has a particular amount of state, and all partitions requite nested state.

$$G(\eta) = \mathcal{V}^* \times (\mathcal{S}_{NFA} \cup (\mathbb{N}^* \times \eta)^*)$$

and $\mathcal{ST}$ as the smallest set so that $\mathcal{ST} = G(ST)$.

Again, we define accessors of $\mathcal{ST}$ for convenience:

- $p_i \mathcal{ST} \to \mathcal{V}$: i-th next-constraint, value for $access(item, p_i(Def)$
- $type\mathcal{ST} \to \{'pattern', 'partition'\}$: Does $\mathcal{S}$ contain state for a pattern or a partitioning
- $t\mathcal{ST} \to \mathcal{S}_{NFA}$: automaton state for Pattern, if $type(St) =' pattern'$
- $n_k\mathcal{ST} \to \mathbb{V}^*$: set of observed key values in k-th partition, if $type(ST) =' partition'$
- $d_k\mathcal{ST} \to St$: nested State definition for the k-th partitioning, if $type(ST) =' partition'$

The validation function for a an item and previous state can now be defined as:

$$sval : \mathbb{B} \times \mathcal{I} \times \mathcal{ST} \times \mathcal{DEF} \to \mathbb{B} \times \mathcal{ST}$$

$$sval(b, next, st, def) \mapsto (b', st') :$$
$$if type(def) =' pattern' : (\forall i, b_{1i} \wedge b_2, ((p_i)_i, t))$$
$$\text{where for each } i\ (b_{1i}, (p_i)) := nval(next, c_i(def), p_i(def), p_i(st))$$
$$(b_2, t) := pval(next, t(def), t(st))$$
$$if type(def) =' partition' : \forall i, b_{1i} \wedge \forall k, b_{2k}, ((p_i)_i, (V_k)_k, (d_{kn})_{kn}))$$
$$\text{where for each } i\ (b_{1i}, (p_i)) := nval(next, c_i(def), p_i(def), p_i(st))$$
$$\text{where for each } k\ (b_{2i}, Vk, (d_{kn})n)) :=$$
$$part - val(next, (q_{kl}(st))_l, n_k, d_k(def), d_{k,n}(st))$$

where n describes to sequence in to which next belongs to according to $q_{k,l}$

For $type(Def) = pattern$, all next constraints c_i/p_i need to hold, as well as the pattern t_i. For $type(Def) = partition$ all next constraints need to hold, all possible partitioning lternatives, and the recursive checks for each of the partitions.

The prefix validation of a stream can now be defined by recursion:

$$b_0, St_0 = true, ((), init)$$

$$b_i, St_i = sval(b_{i-1}, \mathcal{S}_i, St_{i-1}, Def)$$

An empty stream is valid and has a pattern state with no next values and initial state of pattern. The validation a stream with a new item is determined as the validity of the prefix (with a computed state), and the validity of the new item given the state.

A finite stream is valid if the complete prefix is (stream-)valid and the all elements of the pattern state are in an accepting state.

$$|S| = N \wedge sval(b, S_N, Def, V) \mapsto (true, V') \wedge \bigcup_{t \in t(Def)} t(V') \in t.accept$$

Validating a next constraint is defined as follows:

$$nval : \mathcal{I} \times \mathcal{C} \times \mathcal{D} \times \mathcal{V} \to \mathbb{B} \times \mathcal{ST} :$$
$$nval(next, c, d, st) \mapsto (b, st')$$
$$(b1, access(next, p))$$
$$\text{where } b1 := (\text{if } (st = \bot) \; true \text{ else } c(st, access(next, p)))$$

If the result b is either *true* (no previous value in the state) or the comparsion with the previous value. The new state is the value of *next*

$$part - val : \mathcal{I} \times \mathcal{P}^n \times \mathbb{N} \times \mathcal{D} \times \mathcal{ST} \to \mathbb{B} \times \mathcal{ST}) :$$

$$part - val(next, L, n, def, st) \mapsto (b, st')$$
$$\text{if}(n = 0)\text{then}(b_n \wedge (\forall x \in KEYVALS : x = \bot \wedge k = 2 \vee$$
$$\forall x \in KEYVALS : x \neq \bot \wedge k = 1), st_n) \vee$$
$$\text{else}$$
$$((b_n \wedge (a \to \bot) \notin KEYVALS) \wedge \|newstate\| \leq l.N, newstate)$$
$$\text{where} KEYVALS := \{(p \to access(next, p)), \forall p \in L\}$$
$$(b_n, st_n) := sval(b_2, next, d_{k,n}(st), d_k(def))$$
$$k := \text{selected partition/partition index}$$
$$NEWSTATE := st \cup KEYVALS$$

For partitioning, the key paths need to be checked. For heterogeneous partitioning (expressed n=0), all values need to be $\bot$ or none. For homogeneous partitioning, no $\bot$ is allowed, a matching key (and thus nested definition) is found, and they size of the seen key space is checked.

$$pval : \mathcal{I} \times \mathcal{T}_{NFA} \times \mathcal{S}_{NFA} \to \mathbb{B} \times \mathcal{S}_{NFA} :$$
$$pval(next, t, st) \mapsto (b, st')$$
$$(CAND \neq \; \varnothing, st \setminus \{sp, \exists a, b : (sp, a, b) \in CAND\}$$
$$\cup \{to, \exists a, b : (a, b, to) \in CAND\})$$
$$\text{where } SYM := \{s \in \Sigma : validate(next, s)\}$$
$$CAND := \{(sp, sym, to) \subset t.E : \exists to, sp \in st, sym \in SYM\}$$

The set $CAND$ is computed that contains all the edges that can be reached from the current states in V_{pat} following the symbol s (which is schema). s is computed by validating all the schema definition in *Sigma*. If $CAND$ is not empty, the pattern is valid. The state is updated by removing the source of these edges and adding the targets.

B.2 Validation Complexity

Based on the formalization in the previous section, a number of properties on the complexity of stream schema validation can be established

Theorem B.1. *The space needed for Schema validation is finite, if the set of recursively nested schema definitions is finite, regardless if the validated stream is finite or infinite*

Proof. By definition of $\mathcal{ST}$, $\mathcal{ST}$ without considering the recursive state contains a finite number of values. If $type(St) =' partition'$, a finite number of nested states is contained, otherwise there are no nested state. The number of nested schema definitions (and thus states) is limited (by requirement of the theorem and also in practice). □

Theorem B.2. *cost of checking a new item without recursion is polynomial,* $O(|NC|) + O(|activestates|)$ *or* $O(|NC|) + O(|Part| * |keys|)$. *Since* $|keys|$ *will be small in practice (in most cases 1 or 2 paths, most likely less than 5, we can treat this is a constant, reducing the total cost to O(n)*

Proof. By definition of $\mathcal{D}$, there is a linear number of rules for each type. The cost of performing *access* can be considered a constant, similarly also choosing the relevant edges for state in the pattern automaton. □

Theorem B.3. *The cost of checking a new item with recursion is* $O(n^m)$, *where* $m \approx log_n(|\bigcup Def|))$ *(nesting depth)*

Proof. The cost of check one level of recursion is $O(n)$ (previous theorem), and at each partitioning a single partition is chosen (by definition of Ψ). Thus for m levels of nesting, the cost will be $\prod_m O(n) = O(n^m)$ □

C Optimization of Continuous Queries Based on Item Schema

The item-level schema (traditional schema) knowledge can contribute to optimization of continuous queries. Here we focus on Streaming XQuery as the query language and present some optimization techniques followed by an example for each of them.

C.1 Window Type Rewriting

If we can derive from the schema information that that there can only be one window at a time, we can rewrite the 'sliding' window to 'tumbling'. For example:

Context: start and end conditions of a sliding window looking for specific elements:

```
forseq $w in $seq sliding window
start curItem $s when $s[self::A]
end curItem $e when $e[self::Z]
return $w
```

Used information: Sequence always has an A, then some other letters (but no A or Z) and then a Z . Sequence is iterating but not interleaved. Therefore, each A is eventually followed by a Z, without another A in-between.

Action: Rewrite sliding to tumbling.

```
forseq $w in $seq tumbling window
start curItem $s when $s[self::A]
end curItem $e when $e[self::Z]
return $w
```

Watch out that the events really are unique as the query wants. As example look at the entry gates: <in> and <out> seems to be ok. But as soon as you compare persons as well, meaning you only take a window from an <in> to an <out> of the same person, this will produce wrong results if the stream is not prefiltered, since it will be interleaved. This is because we would ignore the <in> of B if the window is open after an <in> of A already.

C.2 Copy (Parts of) the End into the Start Clause

To reduce the number of open windows it makes sense to restrict the start of a window as much as possible. Therefore, if something is checked in the end condition and we know that this can be done in the start clause, we should do it there too (or maybe only there). We propose here to copy the condition. But even in the following example it would make sense to move it instead of copying, since it will be fulfilled at both places. The advantage of having it at both places is when a system evaluates end conditions in some order, e.g. looking first only at parts about the current element and only if they are fulfilled the rest will be checked to reduce the number of accesses on older elements. For example:

Context: query checks in end condition that the end element has some specified value and that this value is the same as in the start element:

```
forseq $w in $seq landmark window
start when fn:true()
end curItem $e when $e/@name eq $w[1]/@name
                    and $e/@name eq ''Fischer''
return $w
```

Used information: none from schema

Action: Since @name is the same in the start and in the end element, check it already in the start to keep the number of open windows as small as possible.

```
forseq $w in $seq landmark window
start curItem $s when $s/@name eq ''Fischer
end curItem $e when $e/@name eq $w[1]/@name
                    and $e/@name eq ''Fischer''
return $w
```

C.3 Remove (Parts of) the WHERE Clause

Some where clauses become superfluous if the schema can tell that the cases where the where would not be fulfilled are impossible to occur. Note, that in case of matches on the element using to close the window this optimisation is only applied if the stream is infinite or force is used, since only then there is no need to check for "partial windows" at the end of the stream. For example:

Context: Where clause contains conditions on other than start/end elements, but they are guaranteed by the schema to be fulfilled.

```
forseq $w in $seq sliding window
start curItem $s when $s[self::start]
end curItem $e when $e[self::end]
where count($w[self::middle]) eq 1
return $w
```

Used information: sequence consists of <start>, some other elements, <middle>, some other elements and <end>. It is repeating and infinite but not interleaved, all elements of the sequence are to appear every iteration.

Action: Since there will be one <middle> between <start> and <end> and only once anyway, we do not have to check for it: Remove the part of the where checking for the <middle> appearance. Note, that this is only possible for sliding or tumbling windows. Landmark windows stay open after the first end and therefore would have another middle in them. We show what to do in the above case if landmark windows are used in Section **??**.

```
forseq $w in $seq sliding window
start curItem $s when $s[self::start]
end curItem $e when $e[self::end]
return $w
```

C.4 Remove Hopeless Windows

Sometimes after all these rewrites there will still be windows that stay open forever, like the one in the next example. Since they stay open forever, all elements contributing to these windows can not be garbage-collected since the system assumes they will be needed as soon as the window is closed, not knowing that this is impossible. This will exceed the available memory at some point in time. With our stream descriptions we have now the possibilities to detect these windows and remove them. For example:

Context: Stream of deliveries to clients. System should sum up prices of sold products per client sold in the first six months. First delivery to a client in a year is detected by having @seqid=1, after 6 months a <midyear> is injected. Each element has date information integrated. Note, that @seqid=1 is possible after <midyear>.

```
forseq $w in $seq sliding window
start curItem $s when $s[self::delivery]/@seqid eq 1
force end curItem $e when $e[self::midyear]
                     and $e/@year eq $s/@year
return <total @client={$s/@client}>
         {sum($w[@client eq $s/@client]/totalprice)}
       </total>
```

Used information: Repeating, non-interleaving sequence with four children: (sequence of deliveries, <midyear/>, sequence of deliveries, <endyear/>). A streamNext concerning midyear/@year says that ”next > current”. A second one concerns all items having a year attribute telling ”next >= current”. Therefore, only one midyear will have the same year. And as soon as any element has a higher year attribute value the next midyear will have the same or a higher year value.

Action: Windows with start element having @seqid=1 appearing between <midyear/> and <endyear/> will never be produced, since after an endyear all elements will have a higher year. The system does not know that, but can derive the impossibility of closing this window from the facts in the stream description. As soon as a higher year arrives in the <midyear/> as the window starting element has, the system knows that the window will never be closed because of the next rule concerning <midyear/>. There are at least the following possibilities:

- Check for every window on each step if the end condition will still be satisfiable. In the above query, after each <endyear/> the items will have a greater @year value. From the schema we know that this value is only getting larger, not smaller. Therefore, the end condition can never be reached and we can close these windows. This seems straight ahead but will impose a very large overhead.
- The overhead mentioned in the previous variant could be reduced by only checking if parts of the end condition are not fulfilled. In the above case we would therefore only check all windows if the element is a <midyear/> but has a higher year value as the item starting the window.
- Rewrite the query by moving the @year clause from the end condition into an if-then-else, with return if condition results to true and the empty sequence if not. Like this we again close all hopeless windows, but do not produce them. This helps memory-management, since if all windows get closed, the elements they refer to can be garbage-collected. If there remain hopeless windows, memory blows up. Note that to do this the system must be able to detect the impossibility of closing some windows before execution. For the above noted stream description this variant is not usable since it is not clear that the year attribute value will only increase after an <endyear/> element.
- Skip events from <midyear/> until the first element having a year attribute value greater than the one of the <midyear/>, since it makes no sense to open a window before the increment of @year.

As one can see we reach again some limit and will not be able to describe all possible inter-dependencies between elements. Missing descriptions will cause some less restrictive optimisations to be used causing some more overhead. But this is still better than having all hopeless windows staying around forever. So we decide on a trade-off between having less overhead but remove some of the windows instead of not removing any windows and having no overhead (but suffering from memory problems). It might be subject to future work to find some more fine grained descriptions to enable the missing descriptions, but it remains to prove if this will bring a big enough benefit to justify the act of complicating describing streams again.

C.5 Efficient Item Storage

Having a well specified schema, its possible to design very optimized storages to store the item information. For example, if the domain type of elements or attributes are known, it would be possible to use array-like structures for accessing them.

D Queries in MXQuery Implementation of LR Benchmark

As shown in figure 3, nine continuous queries have been used to realize the LR Benchmark queries.

D.1 Q1: Car Positions

```
  declare variable $InputSeq external;

  for $w in $InputSeq
  where $w/@Type eq 0 return $w
```

D.2 Q2: Accident Segments

```
declare variable $ReportedCarPositionsSeq external;

forseq $w in $ReportedCarPositionsSeq early sliding window
start curItem $s_curr, prevItem $s_prev when $s_curr/@minute ne
  $s_prev/@minute
end curItem $e_curr, nextItem $e_next when ($s_curr/@minute +2) eq
  ($e_next/@minute)
let $currMin := fn:ceiling($e_curr/@minute)
let $stopedCars :=
for $rep in $w
group $rep as $r-group by $rep/@VID as $vid_s, $rep/@XWay as
  $xway_s, $rep/@Seg as $seg_s, $rep/@Dir as $dir_s, $rep/@Lane
  as $lane_s, $rep/@Pos as $pos_s
where count($r-group) ge 4
return <stopped_car VID="{$vid_s}" XWay="{$xway_s}" Seg="{$seg_s}"
  Dir="{$dir_s}" Lane="{$lane_s}" Pos="{$pos_s}"></stopped_car>
let $accidents :=
for $car in $stopedCars
group $car as $c-group by $car/@XWay as $xway_a, $car/@Seg as
  $seg_a, $car/@Dir as $dir_a, $car/@Lane as $lane_a, $car/@Pos as $pos_a
where count($c-group) ge 2
return <accident minute="{$currMin}" XWay="{$xway_a}" Seg="{$seg_a}"
  Dir="{$dir_a}"></accident> (: Pos="{$pos_a}" :)
let $accidentsRes := if ( count($accidents) gt 0 ) then <accidents>
                       {$accidents} </accidents>
                 else <accidents><accident minute="{$currMin}" XWay="-1"
                   Seg="-1" Dir="-1"></accident></accidents>
return $accidentsRes
```

D.3 Q3: Car Positions to Respond

```
declare variable $ReportedCarPositionsSeq external;
declare variable $CAR_POS_STORAGE external;

for $item in $ReportedCarPositionsSeq

let $prevCarRep := lr:store("CAR_POS_STORAGE", (@VID eq $item/@VID) )

return
( if ( count($prevCarRep) eq 0 or ($prevCarRep/@Seg ne $item/@Seg and
   $item/@Lane ne 4) ) then $item
```

```
 else (),
  lr:store-update("CAR_POS_STORAGE", $item, (@VID eq $item/@VID))
)
```

D.4 Q4: Segment Statistics for Every Minute

```
declare variable $ReportedCarPositionsSeq external;

forseq $w in $ReportedCarPositionsSeq early tumbling window
start curItem $s_curr, prevItem $s_prev when ( fn:ceiling($s_curr/@minute)
  ne fn:ceiling($s_prev/@minute))
end nextItem $e_next when ( $s_curr/@minute +1) eq $e_next/@minute
let $currMin := fn:ceiling($s_curr/@minute)
let $avgCarSpeed :=
for $rep in $w
group $rep as $r-group by $rep/@VID as $vid_a, $rep/@XWay as $xway_
  a, $rep/@Seg as $seg_a, $rep/@Dir as $dir_a
return
  <res XWay="{$xway_a}" Seg="{$seg_a}" Dir="{$dir_a}" VID="{$vid_a}"
  vAvgSpeed="{avg($r-group/@Speed)}" ></res>
 let $segStatistics := (<res endMark="1" minute="{$currMin}"></res>,
for $car in $avgCarSpeed
group $car as $c-group by $car/@XWay as $xway, $car/@Seg as $seg,
  $car/@Dir as $dir
return
  <res endMark="0" minute="{$currMin}" XWay="{$xway}" Seg="{$seg}"
  Dir="{$dir}" avgSpeed="{avg($c-group/@vAvgSpeed)}" carCount=
  "{count($c-group)}"></res>, <res endMark="3" minute="{$currMin}">
  </res>,<res endMark="3" minute="{$currMin}"></res> )
return $segStatistics
```

D.5 Q5: Toll Calculation

```
declare variable $SegmentStatSeq external;
declare variable $ACCIDENT_STORAGE external;

forseq $w in $SegmentStatSeq sliding window
start prevItem $s_prev when $s_prev/@endMark eq 1
force end nextItem $e_next when ($e_next/@endMark eq 3) and
  ( ($s_prev/@minute + 4) eq $e_next/@minute )
let $resMin := $e_next/@minute
let $allAccSeg := lr:store("ACCIDENT_STORAGE", (@minute eq $resMin) )
let $segData :=
for $s in $w
where $s/@endMark eq 0
group $s as $s-group by $s/@XWay as $xway, $s/@Seg as $seg,
  $s/@Dir as $dir
return
<res XWay="{$xway}" Seg="{$seg}" Dir="{$dir}" avgSpeed=
"{avg($s-group/@avgSpeed)}" carCount="{mxq:empty-to-zero(
$s-group[@minute eq $resMin]/@carCount)}"></res>
let $allAffectedSeg :=
for $segmCurr in $allAccSeg
let $segm := $segmCurr/@Seg
return
if ($segmCurr/@Dir eq 0)
(: eastbound direction :)
then <res> <XWay>{data($segmCurr/@XWay)}</XWay><Dir>
  {data($segmCurr/@Dir)}</Dir><startSeg>{data($segm) - 4}
  </startSeg><endSeg>{data($segm)}</endSeg> </res>
(: westbound direction :)
else <res> <XWay>{data($segmCurr/@XWay)}</XWay><Dir>
  {data($segmCurr/@Dir)}</Dir><startSeg>{data($segm)}</startSeg>
  <endSeg>{data($segm) + 4}</endSeg> </res>
let $tollResults :=
for $sData in $segData
let $affSeg :=
for $sCurr in $allAffectedSeg
where $sCurr/XWay eq $sData/@XWay and $sCurr/Dir eq $sData/@Dir
  and $sCurr/startSeg le $sData/@Seg and $sCurr/endSeg ge
  $sData/@Seg
return $sCurr
let $notInAccidentZone := count($affSeg) eq 0

let $lastMinCarCount := $sData/@carCount - 50
let $t := if ( $notInAccidentZone and $sData/@avgSpeed < 40 and
            $lastMinCarCount > 0 )
        then $lastMinCarCount * $lastMinCarCount * 2
        else 0
  return <res minute="{$resMin + 1}" XWay="{data($sData/@XWay)}"
    Seg="{data($sData/@Seg)}" Dir="{data($sData/@Dir)}" avgSpeed=
    "{data($sData/@avgSpeed)}" ccount="{data($sData/@carCount)}"
    toll="{$t}"> </res>
return <tolls>{$tollResults}</tolls>
```

D.6 Q6: Accident Events

```
declare variable $ACCIDENT_STORAGE external;
declare variable $CAR_POSITIONS_TO_RESPOND external;

for $s_curr in $CAR_POSITIONS_TO_RESPOND
let $prevMin := $s_curr/@Time idiv 60
let $allAccSegOnWay := lr:store("ACCIDENT_STORAGE", (@XWay eq
  $s_curr/@XWay and @Dir eq $s_curr/@Dir and @minute eq $prevMin) )
let $checkAcc :=
for $s in $allAccSegOnWay/@Seg
let $accOnWay := if ($s_curr/@Dir eq 0)
(: eastbound direction :)
then if ( ($s -5) lt $s_curr/@Seg and $s_curr/@Seg le $s)
     then data($s) else()
     (: westbound direction :)
     else if ($s +5 gt  $s_curr/@Seg and $s_curr/@Seg ge $s)
          then data($s) else()
return $accOnWay
let $accidentAlert := if ( count($checkAcc) gt 0 )
                    then <alert Type="1" Time="{$s_curr/@Time}"
                      Emit="" VID="{$s_curr/@VID}" Seg=
                      "{$checkAcc[1]}"></alert>
                    else ()
return $accidentAlert
```

D.7 Q7: Toll Events

```
declare variable $TOLL_STORAGE external;
declare variable $BALANCE_STORAGE external;
declare variable $CAR_POSITIONS_TO_RESPOND external;

for $s_curr in $CAR_POSITIONS_TO_RESPOND
let $prevMin := ($s_curr/@Time idiv 60) + 1
let $segToll := lr:store("TOLL_STORAGE", (@Seg eq $s_curr/@Seg
  and @XWay eq $s_curr/@XWay and @Dir eq $s_curr/@Dir and @minute
  eq $prevMin) )
let $newBal := <res VID="{$s_curr/@VID}" Time="{$s_curr/@Time}"
  Bal="0" Toll="{ mxq:empty-to-zero($segToll/@toll) }"></res>
return
  (
  lr:store-update("BALANCE_STORAGE", $newBal, (@VID eq $s_curr/@VID)),
  <res Type="0" VID="{$s_curr/@VID}" Time="{$s_curr/@Time}" Emit=""
  Speed="{ mxq:empty-to-zero($segToll/@avgSpeed) }" Toll="
  { mxq:empty-to-zero($segToll/@toll) }"></res>
  )
```

D.8 Q8: Balance Query

```
declare variable $InputSeq external;
declare variable $BALANCE_STORAGE external;

for $w in $InputSeq
where $w/@Type eq 2
return
let $carBal := lr:store("BALANCE_STORAGE", (@VID eq $w/@VID) )
return
<res Type="2" Time="{$w/@Time}" Emit="" ResultTime="
{data($carBal/@Time)}" QID="{$w/@Qid}" Bal="{$carBal/@Bal}">
</res>
```

D.9 Q9: Daily Expenditure Query

```
declare variable $InputSeq external;
```

```
for $w in $InputSeq
where ($w/@Type eq 3) return $w
```

E Guaranteed Window Orderedness

In following we show that the sliding windows opened by the Toll Calculation query are ordered. Other queries either have no usage of windows or use tumbling windows which are naturally ordered.

```
forseq $w in $SegmentStatSeq sliding window
start prevItem $s_prev when $s_prev/@endMark eq 1
force end nextItem $e_next when ($e_next/@endMark eq 3) and
(($s_prev/@minute + 4) eq $e_next/@minute )
...
```

Since this query is consuming the output of the Semgment Statistics for Every Minute (Q4), first we should know that how the outut stream of that query would look like. According to the definition of the Q4, it generates a stream which looks like <res ... endMark="1"> (<res ... endMark="0"><res ... endMark="3">)*. It also preserves the non-decreasing ordering property over the 'minute' attribute.

Now, in the query Q5, by definition of the sliding window type at any point there is at most one new window being opened [or can be shown by taking a very similar approach for the example in the paper], meaning windows are opened regarding their start clause. The end close has two conditions and proving that closing order of the windows is same as their opening based on only one of these two conditions is sufficient, since the conjunction between them is 'and'. Applying the same argument from the text (Lemma 2), with a little modification (here the windows are of size 5) will easily prove the orderedness property of windows.