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

Peter M. Fischer
Systems Group
ETH Zurich
Switzerland
peter.fischer@inf.ethz.ch

Kyumars Sheykh Esmaili
Systems Group
ETH Zurich
Switzerland
kyumarss@inf.ethz.ch

Renée J. Miller
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 relational and XML databases, the same is not true for data streams. The existing work 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.

## Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*; H.2.3 [**Database Management**]: Languages—*Data description languages (DDL)*; H.2.4 [**Database Management**]: Systems—*Query Processing*

## Keywords

Stream Databases, Stream Constraints, Semantic Optimization

## 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 [35]. However, beyond a few limited proposals (including K-Constraints [10] and Gigascope [15]), 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 are explicitly specified, can be used to check data consistency, improve application modeling, 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 [26], 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

To illustrate our approach, consider Linear Road which is a popular benchmark for data stream management systems [8]. 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 contain 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's 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.

The explicit specification of constraints in Stream Schema will enable a stream system to automatically exploit this semantic and structural 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.

### 1.2 Contributions

The modeling of common structural constraints permits automated optimization, facilitates human understanding of streaming applications (which are notoriously hard to understand), and simplifies query writing as checks for integrity constraints do not have to be hand-coded into queries.

In addition to our model, Stream Schema, the main contributions of this work are:

- a discussion of how Stream Schema specification can be exploited in a wide range of stream processing models;
- an analysis of how Stream Schema can be used in the static analysis of queries to simplify (or minimize) the queries;
- a suite of runtime optimizations enabled by Stream schema;
- two case studies (on Linear Road and on an RFID-driven supply chain application) showing how the separation of queries from data constraints changes (and simplifies) how applications using stream queries are modeled and implemented.

### 1.3 Structure of the paper

The rest of this paper is structured as follows. We discuss the background and related work in Section 2. In Section 3 we present Stream Schema and discuss validation of streams in Section 4. We then discuss how Stream Schema can be exploited in a wide range of stream processing models (Section 5). Finally, we present several applications of Stream Schema in Section 6. Two cases studies (Sections 7, 8) and their relevant experiments show the applicability and benefits of Stream Schema on different models, workloads and implementations.

## 2. RELATED WORK

In developing Stream Schema, we have chosen to focus on common structural constraints that are useful in semantic checking of continuous queries. Before presenting our model (Section 3), we consider a few other approaches to designing and using metadata in stream processing. Our decision to focus on structural constraints was largely influenced by this related work.

Several approaches [35, and others] have considered how to specify and exploit dynamic behavior (such as arrival rates or dynamic delays) in stream optimization. These solutions are strongly dependent on the processing model. A well-known example is the work of Tucker et al. [34] on the use of *punctuation semantics* to optimize stream operators. A punctuation can be used to unblock operators or output partial results. There are also many proposals for stream constraints that are based on a specific processing model or the language of a specific system, the most common of which are types of window specifications [24, 16]. Such constraints are orthogonal to properties of the data stream itself, properties we call *static* to distinguish them from *dynamic* or processing-model dependent properties. In Stream Schema, we focus on the specifation of static data behavior.

Since the common wisdom is that streams evolve over time, we need to clarify what we mean by *static*. For many applications, there will be semantic constraints that hold over time (either forever, or for a significant enough period of time to make them useful in optimization). Many of these applications have domain or integrity constraints that can be encoded in Stream Schema as structural constraints or as relative constraints on a portion of a stream. These constraints can then be used to optimize query processing. Examples include business domain constraints, government regulatory constraints, or physical constraints on the way a manufacturing line has been constructed. In contrast, other work has looked at capturing constraints for sensor data and other applications where the data may be error-ridden or have a highly dynamic structure. Such data may not conform to any strict constraints. To handle such applications, the use of soft constraints has been proposed [28]. We do not consider such approximate constraints in our work.

Finally, we have chosen to focus on structural constraints within a single stream since this is already a very rich (and under-studied) area. Constraints between streams have been studied by others [16;, 10;, 24, and others] although these solutions also tend to depend on specific processing models, join algorithms and dynamic properties. For example, Golab et al. [24] 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 joins between streams using join elimination and anti-join elimination.

## 3. STREAM SCHEMA

We begin with an overview of our design choices to express static stream metadata, then present our schema model. As with any schema formalism, there is a clear trade-off between the expressive power of the schema and the cost of validation (checking whether a stream conforms to a schema). In our model, we have chosen to include a rich set of descriptions that (as we show in Section 6) have considerable power and can be exploited in optimization. While cost-based optimization of streams is still an active area of research, we give evidence that our descriptions have the potential to be used by emerging optimizers.

### 3.1 Overview

We use a totally ordered sequence of items as our formal stream model. We do not assume any specific value ordering, in particular, we do not designate any special meaning to particular attributes, e.g., a timestamp attribute that must be present in every item, since our constraints are powerful enough to express them if they are required for an application.

When exploring the design space to describe these item sequences, two main classes of formalism are available: 1) sequence constraints to describe the relationship of items; and 2) structuring to decompose into subsequences (according to various criteria).

Since item sequences (under names such as "strings" or "traces") are commonly used to describe the behavior of complex systems, several formalisms exist that capture the item relationships:

1. *value (order) relationships between items*, such as the total order of the values of certain attributes [15];
2. *temporal logic*, such as LTL, to express that particular properties will hold always, will hold at some point, or will hold until some point; this is often used in specification and validation of complex systems [31]
3. *grammars*, that describe permitted sequences of items (e.g., the child nodes in an XML document) [3]. To deal with infinite sequences, $\omega$-grammars [33] have been defined.

These three approaches share a certain amount of overlap (and in certain cases equivalence, e.g., LTL and $\omega$-regular grammars [33]). Since temporal logic and grammars, in particular, can be quite expressive (and thus expensive to verify), our design choice is not so much which to use, but rather how expressive should our constraints be.

Sequence structuring, in turn, has not received a great deal of attention. In Stream Schema, sequence structuring permits the scoping of constraints. This enables a more human-understandable model than one based on constraints alone.

Generally speaking, a data stream can be structured along two dimensions in order to provide substreams:

1. *item values*, resulting in substreams with particular value ranges and relationships between values (logical partioning);
2. *item ranges*, resulting in contiguous substreams with particular properties (e.g., login sessions with a specific structure).

It should be noted that in nearly all sequence or stream-oriented query processing languages, operators with the corresponding semantics are available, since *item value* structuring has a correspondence to *group by*, whereas item range structuring corresponds to *window* or *pattern* operators.

Having explored the design space of structure and constraints of streams, the important design choice is to determine which combination provides a good balance between expressiveness and complexity. We allow the nesting of item value structuring without limitation, since this is simple to validate even in the presence of other constraints. We also permit value (order) constraints and the combination of these at any level of nesting. For item range structures, we are more restrictive: they may not overlap, and they are only allowed as terminals in the nesting of the logical constraints. This design avoids the complexity of having to combine the substreams in validation, reducing the verification effort. As we will show in Section 6.3, this restricted form, nonetheless, allows important optimizations for queries containing patterns and semantic windows.

In order to express item sequence constraints, we use a powerful, but simple regular expression language. We have chosen to exclude the full expressive power of modern pattern matching languages (found in many stream and CEP query languages [6;, 17, and others]). Such languages would permit the expression of intricate constraints on how portions of a stream can overlap or be recursively nested. While central to query languages, the *a priori* validation of such constraints could be expensive in both space and time. Furthermore, optimizations based on such constraints are not yet known, and it is not yet clear whether an effective cost-model for such constraints could be developed and used in practice in an optimizer.

On the reasons of validation complexity, we excluded general integrity constraints: uniqueness or functional dependencies among element values in a single stream would require possibly extremely large amounts of state, as all different occurences would have to be recorded. Foreign key relationships are only possible among multiple streams, which we have postponed for future work, as it involves model-specific join algorithms.

As a result, Stream Schema includes the following types of constraints on streams:

- Specification of relationships (value and structural relationships) over (sub)sequences within a stream.
- Logical Partitioning of streams (by value and by structure).
- Grouping of (sub)sequences of streams. Notably, however, we restrict this grouping to form a tree structure.
- Relative constraints between portions of a stream.

These descriptions are formalized and summarized in Table 1 and explained below. We give examples of each description using the Linear Road benchmark just to keep the examples simple. However, we stress that our model is much more general than Linear Road as we illustrate in Sections 5, 7 and 8.

## 3.2 Item Schema

A data stream is composed of items (also called tuples or events). Items can be specified in any data model, e.g., relational [5, 9, 7, 15, 17, 1] , XML [11], object-based models [2]. Given this heterogeneity, Stream Schema is designed to work with any item model that supports access functions, denoted by $A$, (be they relational attributes, elements, XPath expressions, methods, etc.) For sake of simplicity, we use the term attribute instead of these access function in rest of the paper.

We will use $\mathit{IS}$ to refer to an item schema (which will have a name $N$ and a set of attributes $\mathcal{A}$. $I$ refers to an item. We say that $I \models \mathit{IS}$ if $I$ conforms to the schema $\mathit{IS}$. We will use $V$ (or sometimes $V_i$) to refer to the domain of an attribute. The value $\mathit{NULL}$ is a special value where $\mathit{NULL} \notin V$ for all domains $V$. Domains may be infinite or finite. Additionally, a set of comparison functions $C$ is defined over each domain, more formally

$$C : V \times V \to \mathbb{B} \cup \mathit{NULL}$$

As an example, for Linear Road, the position reports have a relational item schema that contains the attributes that include Time (TIME), Vehicle ID (VID), and Speed (SPD). A simple application of item schemas is to ensure item structure and domain integrity, e.g. checking that all attributes are present, and the observed values are in the required domain, as to represent application-related domain constraints. Item schemas also play an important role in optimizing the storage of items and in optimizing predicate evaluation on item values. We refer readers to the existing literature on how to do this [13, 25, 22, 32].

## 3.3 Logical Partitioning

Two logical partitioning constraints are considered in Stream Schema: 1) partitioning by item structure; and 2) partitioning by attribute value.

Formally, partitioning by item structure splits the original stream $S$ into two substreams $S_1$ and $S_2$, using a set of attributes $\mathcal{A}'$. Items on which all attributes of $\mathcal{A}'$ are defined (i.e., non-null) go into $S_1$, all other items go into $S_2$.

As an example of partitioning by item structure, Linear Road's input stream is a combination of four different streams (one containing only position reports, and the other three query streams). To support DSMSs 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. Attributes for not needed for a particular type are given the value NULL. For example, for the position report stream the attributes Type, TIME, VID, SPD, XWay, LANE, DIR, SEQ, POS are non-null. Each of these substreams has its own item schema, and possibly also other constraints, e.g., pattern or next-constraints (see below). In a DSMS that actually support heterogeneous item schemas, this could have been expressed explicitly using different item schemas.

Partitioning by attribute value is defined using an attribute $A_p$ (with finite domain $V_p$) and a partitioning bound $n$. This constraint creates at most $n$ substreams with the same schema, one for each value of the $A_p$ attributes. In general case, instead of $A_p$, we can have sequence of attributes $\mathcal{A}' = (A_1, ..., A_k)$ with domain of $V_1 \times V_2 ... \times V_k$. Since $\mathcal{A}'$ serves as a unique identifier/key of this subsequence, one can express a global stream key by choosing an appropriate $\mathcal{A}'$ and specifying $n$ as 1.

As an example, the position reports stream can be partitioned

| Const. | | Formal Definition | Example from LR |
|---|---|---|---|
| Item Schema | | $\mathbb{IS} : (N, \mathcal{A})$<br>$\mathcal{A} : \{A_i\}$<br>$A_i(\mathbb{IS}) = V_i$<br>$A_i(I) = v \in V_i \cup \mathit{NULL}$ | For Item Schema of position report stream (P):<br>$N$: $\mathrm{P}_{\mathbb{IS}}$<br>$\mathcal{A}$: {TIME,VID,SPD,XWay,LANE,DIR,SEG,POS} |
| Partitioning | By Str. | $S \xrightarrow{\mathcal{A}'} S_1, S_2$<br>$\forall I \in S \begin{cases} I \in S_1 & \text{if } \forall A_i \in \mathcal{A}', A_i(I) \neq \mathit{NULL} \\ I \in S_2 & \text{otherwise} \end{cases}$ | The input stream (S) along :<br>$\mathcal{A}'$: {TIME,VID,SPD,XWay,LANE,DIR,SEG,POS}<br>$\mathrm{S}_1$: position report stream (better known as P)<br>$\mathrm{S}_2$: rest (mixture of three query streams) |
| | By Val. | $S \xrightarrow{A_p,\ n} S_1, S_2, ..., S_n$<br>$\forall I \in S, I \in S_i$ if $A_p(I) = v_i$<br>where $V_p$=$\{v_1, ..., v_n\}$ is the finite domain of $A_p$ | Position report stream (P) along:<br>$A_p$: VID and $n$ =$\vert$VID$\vert$<br>$\mathrm{S}_i$: the position report stream of the vehicle with<br>VID(I) = $\mathrm{vid}_i$ |
| Pattern/ Repetition | | $P ::= F \mid F'^{**'}$<br>$F ::= FF \mid F' \vert' F \mid F'^{*'} \mid F'^{+'} \mid '('F')' \mid E \mid \epsilon$<br><br>and element $E$ is defined by restricting the Item Schema<br>$E$: $\mathbb{IS}_{(V_i \leftarrow \{v_i\})}$ | Vehicle trip pattern for a particular vehicle (vid)<br>$(\mathrm{L}_0(\mathrm{L}_1 \vert \mathrm{L}_2 \vert \mathrm{L}_3)^*\mathrm{L}_4)^{**}$<br>Where $\mathrm{L}_j$ is defined by restricting $\mathrm{P}_{\mathbb{IS}}$<br>$\mathrm{P}_{\mathbb{IS}_{(\mathrm{VID} \leftarrow \{\mathrm{vid}\}, \mathrm{LANE} \leftarrow \{\mathrm{j}\})}}$ |
| Next Const. | | $c(A_n(current), A_n(next))$<br>where $current$ and $next$ are any two adjacent items:<br>$current = I_i \leftrightarrows next = I_{i+1}$<br>and comparison function $c$ is defined over domain of $A_n$ | On the main stream:<br>$\leq$ (TIME(*current*), TIME(*next*)) |
| Disord. | | $\forall i, j \in \mathcal{N}$:<br>$< (i + k, j) \Rightarrow\ < (A_o(I_i), A_o(I_j))$<br>where stream is in *ascending* order on accessor $A_o$<br>and $k$ is the upper bound of disorderedness | On the main stream:<br>$A_o$: TIME<br>$k = 0$ |
| Combination | | $Tree \rightarrowtail (Tree \vert Leaf)^+$<br>$Tree$: nodes with *Partitioning* constraint<br>$Leaf$: nodes with *Pattern* constraint<br><br>Having this tree, stream-level *Next* constraints can be placed at **any** node, pattern-level *Next* constraints at **leaf** nodes, and *Disorder* constraint at **root** node. | Combination of all important constraints on input stream of LR is depicted in Figure 1. |

**Table 1: Formal Definition of Stream Schema Constraints**

based on the VID value ($A_p = VID$). If there are at most $|VID|$ different vehicles, then $|VID|$ substreams are created.

Multiple alternative partitionings of the same stream are often possible. For example, we could split the position report not 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.

## 3.4 Subsequences and Patterns

In many cases, a stream (or a logical partition of stream) can be broken into item ranges (also called subsequences), which in turn 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, patterns are of finite length. They can repeat infinitely often, but the instances may not overlap. Each of these repetitions therefore defines a subsequence, and their repetition defines a possibly infinite stream. We use a regular language $F$ for patterns, and designate the repetitions as $P$. The regular language is defined over items that may satisfy an item schema or restrictions (selection conditions) on an item schema ($E$). Compared to languages in pattern queries (e.g., SASE [6] or Cayuga [17]), this language has two simplifications that stem from our need for a description language, not a query language: 1) Only contiguous patterns can be specified, to ensure that the whole stream is described and no items can be ignored 2) All language constructs related to *matches* (e.g., length, next start, result generation) are not needed, since we do not allow overlapping matches, and want to describe the whole stream.

As an example, in the Linear Road benchmark, there are some patterns in the input streams, including the following taken from the specification [8].

> 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., permitting a blocking operator on infinite data to produce results (see Section 6.3).

## 3.5 Item Value Relationship

In many streams, attribute values in different items have a well-defined relationship. For example, we may 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. In Stream Schema, these orderings between attribute values are specified using *next-constraint*. It is defined based on a comparison function $c$ over an attribute $A_n$ of two adjacent items namely *current* and *next*.

The scope of validity for such a constraint may be the whole stream, a logical substream, or a single repetition of a pattern. For the sake of simplicity, we illustrate the pair of adjacent items in latter with *pnext* and *pcurrent*.

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 unblock operators (similar to punctuations[34]), rewrite semantic windows or patterns, and also for semantic correctness checks (to determine if semantic windows close).

## 3.6 Disorder

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 delays or the lack of strict time synchronization between different sources, items may arrive out of order. In Stream Schema, similar to Ordered-Arrival Constraint[10], we use a parameter $k$ to express the 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, thus creating disorder in the stream.

Knowing a bound on the amount of disorder 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 [30, 29].

## 3.7 Combining Constraints

The combination of the different types of constraints in stream schema results in a tree-like structure. In this tree, every node represents a stream (or a substream) and every edge represents a *partitioning* constraint. The parent node of this edge is the stream which the partitioning constraint holds on and the child stream is result of the partitioning. In case of partition *by structure*, this is the substream with non-null values, and in case of partition *by value*, this substream is the representative of the whole class of partitions). Additionally, in this tree, only the leaf nodes have *pattern/repetition* constraints.

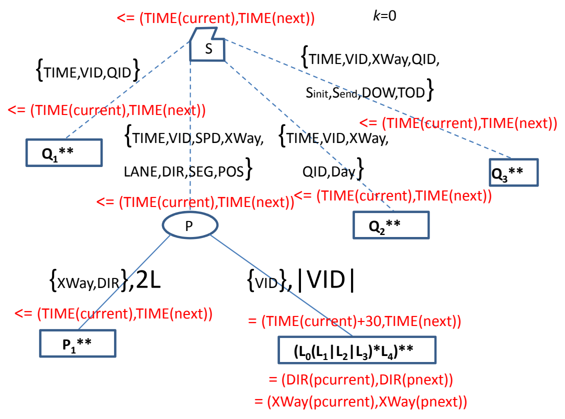
To position other Stream Schema constraints, we can have:

1. the stream-level *next-constraints* at any node,
2. pattern-level *next-constraints* at leaf nodes, and
3. a single *disorder constraint* at the root node.

Since the partitioning might place adjacent pairs of items into different partitions, the next-constraint will not always hold over the partitioning, thus a new next-constraint is needed to express the relationships after the partitioning. This could be the same relationship, or a more specific relationships if the substream can be described in more detail.

As an example, figure 1 depicts the constraint tree on Linear Road data stream. Reading the figure top-down, there is a next-constraint 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. The position report stream $P$ can further be partitioned by either VID values, or by XWay and DIR. The root stream has a non-decreasing TIME attribute which also holds for many of its substreams. After partitioning along the VID attribute, the resulted stream has a more precise next-constraints (stating the fact that a particular vehicle, emits its position report every 30 seconds)

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 an instance of the pattern: during a trip, a vehicle will stay on the same highway and direction, segments and positions will either be non-decreasing or non-increasing, depending on the direction.



<= (TIME(current),TIME(next))
k=0
S
{TIME,VID,QID}
{TIME,VID,XWay,QID, Sinit,Send,DOW,TOD}
<= (TIME(current),TIME(next))
<= (TIME(current),TIME(next))
Q1**
{TIME,VID,SPD,XWay, LANE,DIR,SEG,POS}
{TIME,VID,XWay, QID,Day}
Q3**
<= (TIME(current),TIME(next))
<= (TIME(current),TIME(next))
P
Q2**
{XWay,DIR},2L
{VID},|VID|
<= (TIME(current),TIME(next))
= (TIME(current)+30,TIME(next))
P1**
(L0(L1|L2|L3)*L4)**
= (DIR(pcurrent),DIR(pnext))
= (XWay(pcurrent),XWay(pnext))

**Figure 1: Linear Road Stream Schema**

## 4. CORRECTNESS AND VALIDATION

In this section, we first explain how the correctness of a Stream Schema constraint should be checked. Then validation is formally described using prefix validation. We include a description of the runtime state needed in evaluation of the prefix. An analysis of the space and time complexity completes this section.

## 4.1 Checking Correctness of Constraints

### 4.1.1 Item Schema

Given an item schema $\mathbb{IS}$, an item $I$ in a (sub)stream satisfies the item schema constraint $\mathbb{IS}$ if for all attributes $A_i$ of the item schema, $A_i(I) \in V_i$, in particular $A_i(I) \neq NULL$.

### 4.1.2 Partitionability

A (sub)stream satisfies a partioning constraint if the constraint assigns every item to a partition (so there are no items omitted). Hence, a partitioning by structure constraint is, by definition, valid since every item is assigned to either $S_1$ or $S_2$. A partitioning by value constraint is valid if the size of the domain of $A_p$ is at most $n$ and if for all $I$ in the (sub)stream on which the constraint is applied, $A_p(I) \neq NULL$.

### 4.1.3 Pattern and Repetitions

A stream satisfies a pattern constraint if an automaton representing the pattern accepts the item stream (prefix). Such an automaton can be created from the pattern specification by translating the finite part of the pattern (represented by $F$ in Table 1) into a finite state machine. For the repetitions ($F^{**}$), additional edges are added from accepting states to states reachable from the starting state, marked with the pattern starting symbols.

It should be noted that the pattern validation formalism closely corresponds to Büchi-automata[33], the default implementation of $\omega$-grammars over infinite sequences: It handles infinite iterations over all well-defined set of accepting states (i.e. the representation of $F$). The difference is in the interpretation of correctness: we detect incorrectness also over finite sequences, while a Büchi-automaton only works with infinite sequences.

#### 4.1.4 Next Constraint

A stream satisfies a next constraint if for every consecutive pair of items $I_1$, $I_2$, $c(A_n(I_1), A_n(I_2))$ is true. Of course a next-constraint specified within a pattern $F$ only needs to hold for items within the same pattern instance, and a next-constraint specified with a partition only needs to hold in this partition.

#### 4.1.5 Disorderedness

Stream Schema does not require an ordering relation. However, one may be specified by a next-constraint. In the presence of a total order specified over a particular attribute (e.g., a timestamp attribute), a disorder constraint of $k$ may also be specified. A stream is valid (satisfies the disorder constraint) if an ordered sequence can be generated by sorting the items inside a sliding window of size $k$.

#### 4.1.6 Combination

Constraints can be combined recursively to form a tree of constraints as illustrated in Figure 1. Constraint checking can be done recursively by checking constraints bottom-up through the tree.

### 4.2 Stream Validation

Since any newly arriving item could violate a given schema, complete validation of an infinite stream is not possible. Therefore stream validation is based on validating the current item using the prefix validation result and prefix validation state. Since disorder is orthogonal to all aspects of validation, we first define the validation algorithm for stream data without a disorder constraint, and then extend the definition and analysis for disordered streams.

In order not to 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.

- Partitioning: we define a recursive data structure containing the nested stream data; in addition we need to store the information on the partition decision. The number of substreams can be derived from the number of (nested) states.
- Pattern: the automaton state for a single repetition of a pattern, e.g., all active states in an NFA.
- 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.

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 related to the complexity of stream schema validation can be established:

- for a finite Stream Schema specification, the space needed for validation is finite, for both finite and infinite streams;
- the cost of validating a new item without recursion is polynomial; and
- the cost of validating a new item with recursion is $O(n^m)$, where $m$ is the nesting depth.

Relevant proofs can be found in a tech report [21].

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

The first variant can be implemented by checking/restoring order according to the ordering relation, then performing the ordered variant of validation. The additional space required is linear to $k$, and the additional cost is the cost of 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. More details can be found in our tech report [21]. This is similar to the solution of Liu et al. [30].

## 5. PROCESSING MODEL INTEGRATION

The next step after defining stream schema is to embed it into the specific data and processing models of data stream management systems (DSMS). Each of these systems uses a somewhat different model, but for all of the systems we have evaluated, a straightforward integration is possible (with one exception that we highlight at the end of the section).

To perform this embedding, the following steps are needed:
1) The abstractions of *item*, *item schema* and *attribute* are mapped to their concrete counterpart in each DSMS. 2) The *stream data model* needs to be checked for compatibility. 3) *Implicit schema constraints* of a DSMS need be expressed in Stream Schema.
4) *Existing Schema-like capabilities* of a DSMS need to be checked against the capabilities of Stream Schema.

Now we discuss details of each of these steps for a number of well-known processing models.

1. In relational systems (Aurora/Streambase [5], CQL [9], the SQL pattern extension [7], Gigascope [15], Cayuga [17], CCL [1]), a stream of homogeneous items are used, where *items* are flat relational tuples with attributes. For XQuery streaming [11], a stream can be heterogeneous (individual items validate against different XML schema definitions), where items are atomic values or XML nodes, accessible by XPath expressions. For such systems, we would need to define an accessor function (or attribute) for each valid XPath expression. SASE [6] 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.

2. For the *stream data model*, most models assume a totally ordered sequence of items as a basis (which can be defined by next-constraints in Stream Schema) with some relaxations to this ordering: CQL uses a sequences of batches [9] as its 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. This can be defined in Stream Schema using a next-constraint with the $\leq$ relation,

instead of the $<$ relation used for defining a total order. Other approaches use k-constraints [10] or Slack (Aurora) to give a bound to the degree of out-of-orderness (with the same semantics as our disorder constraint).

3. Many Stream Processing Models define *implicit timestamp attributes*. In Stream Schema, these implicit constraints can be expressed using an item schema and next constraints capturing the appropriate order of values. Since some systems (SASE, Esper, XQuery) do not require timestamps, we chose not to make timestamps a required part of Stream Schema.

4. While most DSMS provide some *schema-like definitions*, the stream-oriented aspects of these schemas are often restricted to some ordering properties and several dynamic properties (e.g., arrival delays). A somewhat closer match is the possibility to define a stream using a query (Esper and Coral8). In this approach, the query specification (filter, pattern) would imply similar schema constraints as our Stream Schema.
   *StreamBase/Aurora* provides a schema-like operator specification $(OnA, slack, GroupByB_1, B_N)$, expressing an order on an attribute $A$, limited disorder $slack$ and the possibility to group by the attributes specified in the $Group\ By$ clause. All of these constructs can be mapped to Stream Schema (next-value, disorder, partition by value). *Gigascope* uses ordered attributes (representing timestamps, sequence numbers, etc.) of three different types: 1. strictly/monotonically increasing/decreasing; 2. monotone non-repeating; and 3. increasing in group. Ordering 1 and 3 are expressible as next-constraint (with partitioning for 3). The precise definition of 2) cannot be derived from the informal definition in the Gigascope paper. If the purpose is to express non-repetition of values in an infinite sequence, this cannot be validated, since all values need to be kept. Otherwise we could also express it as a $<$ next constraint.

## 6. APPLICATION OF STREAM SCHEMA

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

1. **Validation and (type) annotation** of the data stream, including reactions to *invalid* data.
2. **Changing Query Semantics**, such as allowing queries to run that would not run without a schema or statically determining that a query will produce no results.
3. **Enabling Optimizations** on query plans and query processing, such as reducing cost or response time.
4. **Extending Modeling** streaming applications by separating constraints from queries.

## 6.1 Stream Validation

In many use cases, explicit validation of streams is not needed, since the stream constraints are guaranteed to hold by the source producing the stream. Similarly, in a distributed stream processing setting, only untrusted data needs to be validated, which may be only a portion the 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.

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.

In terms of development, clearly building a stream validator from scratch is always an option. In following subsections, we propose two other alternatives.

### *6.1.1 Validation Using Existing Validation Framework*

We implemented a large part of Stream Schema based on the Xerces XML parser and validator [4], since it already provides most of the operations and data structures needed for the validation of Stream Schema. We extended XML schema with the new Stream Schema 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 schemas using the standard XML schema validation mechanisms. 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 we 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.

### *6.1.2 Validation Using Continuous Queries*

As an alternative, Stream Schema can be translated into a continuous query, since the operations required for stream schema validation match closely the set of commonly available expressions and operators in DSMS (and Complex Event Processing systems). If a matching operator is not available, such a system would 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 be useful.

## 6.2 Impact On Stream Processing Semantics

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

### *6.2.1 Static Check for Non-Executable Expressions*

The system can use Stream Schema to either output a warning or to abort execution in the following situations.

- Non-executable predicate-based windows with aggregates. For example, in the web log example where we have a pattern `logon browse`$^*$ `logoff` specified, if a query defines a window to be closed on the occurence of a `browse` item, this window might not close, since occurrence of the `browse` item is optional in the pattern specification. The system can therefore issue a warning.
- Execution of blocking operators. For example, if a blocking operator (e.g., a sort) is used over a stream and from an analysis of the schema a system can determine that the execution may be infinite, a system can abort the operation (or issue a warning).
- 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.

### 6.2.2 *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 order. A more detailed example is given in Linear Road case study.

## 6.3 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 on item schema specifications are similar to existing schema-based optimizations [13, 25, 22, 32] and will not be discussed here.

Stream Schema provides the metadata to perform optimizations, so an important class of optimizations enabled by stream schema is 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 [11]. The important contribution of stream schema is to formally express if an optimization is applicable.

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

### 6.3.1 *Pipelined Execution*

When strictly following the definition of semantic windows, e.g., FORSEQ [11], such a window could 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 aggregation) 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 to allow this optimization: 1) every open window will be closed at some point, 2) windows have a total order; they will be closed in the same order they were opened. A detailed analysis on this optimization is given in the LR case study.

### 6.3.2 *Stream Data Partitioning*

The volume of data that needs to be processed in real time DSMS can easily exceed the resources available on a centralized server. A well-known approach to tackle this problem that has been used in Distributed DSMSs is data stream partitioning. This approach requires the splitting of resource-intensive query nodes into multiple nodes each working on a subset of the data feed [14].

Johnson et al. [27] propose a solution to partitioning a stream query workload based on a set of queries. Their approach includes two steps: 1) finding a partitioning scheme which is compatible with the queries; and 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 the Johnson et al. work is that the data (not just the queries) are actually partionable by the schemes produced in their first step. The partioning constraints of Stream Schema can be used to determine which (if any) of these schemes can actually be used, thus making this approach fully automatable. In the first case study, we will show how this optimization technique can improve the performance of the LR implementation.

### 6.3.3 *Window/Pattern Optimizations*

- The patterns of Stream Schema can be used to simplify the overlap in the specification of window or pattern instances. Specifically, if we can determine that a new window (or pattern) can only start after the previous one has been completed, then we may be able to optimize the processing of windows (or patterns). To be more concrete, here are two examples from different frameworks:
  - `Sliding` or `Landmark` windows are more expensive compared to `Tumbling` windows, since the number of open windows is potentially much higher, and more checking may be needed. Using the information in Stream Schema, a query written using landmark or sliding windows can be rewritten into a tumbling window. An example of this rewriting is given in the LR case study later on.
  - One of the constructs in the SQL pattern extension[7] 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 should start looking for the next match from the row after the next row or from the last row of the current pattern. Using a pattern constraint over the stream, 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 the original one, but cheaper because avoids performing unsuccessful matches.
- Removing existing structure from window/pattern specification to only check what a schema does not already provide. The following is an example for pattern matching systems: Complex event processing (CEP) systems usually use Finite State Machines (FSMs) for pattern matching [17, 6, 19]. 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 other areas [18]. 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)*`.

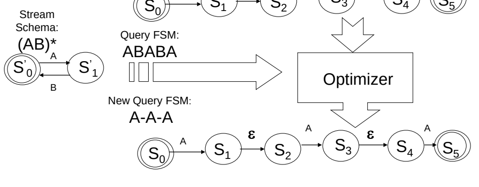


**Figure 2: Pattern FSM decompositon**

### 6.3.4 State Reduction

Many stream operators (e.g., join, group-by, and sort), maintain state in order to generate the correct results [12]. Most of the optimization techniques for stream processing aim at reducing the 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, that is avoid materialization (or discard as early as possible); and 2) purge states after certain evaluation steps.

#### Avoiding materialization.

In the stream processing system, newly arriving items are fed into the open windows to determine if they will contribute to the output results. If there is a way in which 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. Stream schema can help in different ways to ease making such decisions. The following are some examples for a stream join operator:

- differences in partition by value bounds : if two streams involved in the join have a partitioning constraint on the join attributes, and the $n$ is not the same, one can drop the items with the missing values from the respective stream;
- 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; and
- mismatch between next constraints (only items which comply with the next-constraints of both streams have a chance to successfully participate in a join).

#### State purging.

In some cases, stream schema constraints allow an operator to purge many of its states. For example, the join operator needs to keep track of a 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 certain items will never show up again, the join processor can purge the state it has been keeping for those items [20].

## 6.4 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 very explicit manner: all possibly relevant predicates and expressions are directly expressed in the query (to ensure correctness), and also often manually arranged (to achieve good performance). By doing so, predicates from two domains are mixed: 1) predicates to describe the desired behavior; and 2) predicates to capture semantic constraints.

The use of Stream Schema enables a different approach. Queries can be written to extract the desired results only. There is no longer any need to provide constraints regarding data consistency and/or structure as part of the query itself.

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

- Simpler queries: queries express only the data to be retrieved and can thus be more easily reused over different streams
- Simpler domain or semantic constraints: constraints need to be declared once and can be re-used 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 shown in Section 8.

## 7. 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 [8] implementation in Continuous XQuery [11]. 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 exist, so the optimizations are discussed at a formal level and implemented by manually adapting the queries.

## 7.1 Existing MXQuery Implementation

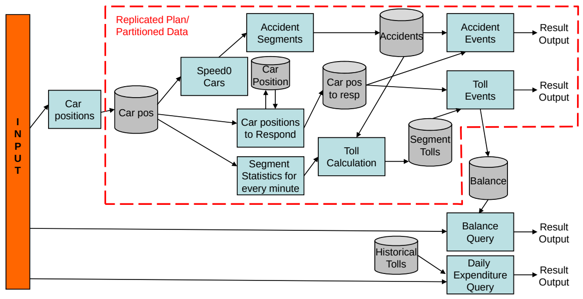


**Figure 3: MXQuery Linear Road Implementation, extended with data partitioning information**

The Linear Road implementation of MXQuery [11] 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.

## 7.2 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 Stream Schema for LR, this can be achieved. For example, the following window expression is part of the query that finds vehicles with speed equal to zero:[1]

```
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 the value of the minute attribute changes in order to show that the window will be closed.

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, consequently closing the window.[2] □

[1]The full query and the analysis for other queries is given in [21].

[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.

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

## 7.3 Query Rewrites for Pipelining Execution

As described in the optimization section, pipelined window execution is an important factor to reduce latency and memory consumption. For semantic windows, an important precondition is that windows will be closed in 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. By definition of sliding windows in XQuery, for each incoming item, at most one window will be opened. Now we show that these windows are strictly ordered with respect to time attribute of their first element. The start condition of the window specificies that a window should be opened if the time attribute is different between two adjacent elements in the stream. In addition, the Stream Schema description states a $\leq$ relationship between the time attributes in the stream, meaning that the only difference of time attribute values can be an increase. As a result of both query and schema constraints, the desired order is achieved. □

**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
$\$s\_curr_{w}/@minute < \$s\_curr_{w'}/@minute$
now we show it's impossible to $w$ to close after $w^{'}$, meaning
$\$e\_next_{w}/@minute > \$e\_next_{w'}/@minute$
but since we know
$\$s\_curr_{w}/@minute + 2 = \$e\_next_{w}/@minute$
$\$s\_curr_{w'}/@minute + 2 = \$e\_next_{w'}/@minute$
this is a contradiction. □

We can use similar reasoning for other continuous queries, details are again given in [21].

## 7.4 Data Partitioning

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

Among the LR continuous queries, the Account Balance query is the only one which does computation over more than one expressway or direction. Therefore, we can easily parallelize the execution of the other queries in the following way:

- *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 responsible for these notifications. For each incoming position report that has fulfilled the notification preconditions, it retrieves the accidents for the *same* expressway and the *same* direction and then notifies the vehicle about accidents in its neighbor segments (if any). *Toll Notification* is analogous.

## 7.5 Experimental Setup

In order to validate the optimizations spelled out in the case study, we re-created the experimental setup of Botan et al. [11]: All experiments were run on dual-CPU AMD system with single-core (pipelining experiment) and dual-core (partition 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 were carefully tuned and chosen to take advantage of the implicit schema knowledge, we created a baseline using semantically equivalent queries that do not use schema knowledge.

## 7.6 Pipelining Experiment

For Linear Road, most window constraints are on minutes, and the resulting computations on the window contents to calculate statistics, accidents and tolls all need to be performed when the value of the minute attribute changes. As a results, without pipelining the response time requirement 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 7.3), 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 of L=2.5, using Stream Schema we were able to scale to L=3.5.

## 7.7 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 on a partition of the key value set, the workload can be distributed over multiple cores, system or data centers. As determined in Section 7.4, 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 to 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.

## 8. CASE STUDY II: SUPPLY CHAIN

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

## 8.1 Item Distribution and Tracking

A typical supply chain system (e.g., a car manufacturing factory) attaches RFID tags to items (e.g., car parts) in order to track their distribution based on their types from the entrance gate towards their specified destinations, e.g., the assembly line depots. For example, all items of type 'gearbox' should 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. These RFID readers are arranged in a tree, as depicted in Figure 4.

A common query is used to detect if items have been misrouted. For example, if an item has ended up at destination 8, but it was supposed to be routed to another destination, this must be detected and reported. Such a query may use patterns to describe valid destinations.

For example, the pattern AB*C where A is the entrance reader, B is any intermediary reader, and C is the correct destination reader for a given car part.

With the help of metadata the performance of such queries can be improved. The idea is that detecting a misrouted item is in many cases possible before it reaches its final destination. Given the reader 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 $R_{11}$], or even higher branches. The structure of this reader tree can be provided by stream schema to the stream engine resulting in a structure-aware item tracking. For example, given the structure of the tree (specified in Stream Schema) and an assignment of part types to destination nodes, as soon as a part that should be going to $R_{14}$ reaches $R_3$, we can report the error.

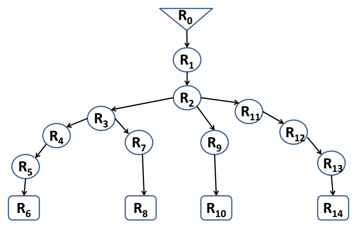


**Figure 4: Supply Chain with RFID readers**

## 8.2 RFID Readings Schema

The item schema for the RFID readings stream (R) is defined as

$$N\text{: R}_{\mathbb{S}}$$
$$\mathcal{A}\text{: \{ReaderID,TagID,ItemType,TIME\}}$$

and combination of constraints on the main stream of RFID readings is depicted in Figure 5. Items in this stream are homogeneous and the TIME values of the readings are non-decreasing (without any disorder). Notice that several readings may arrive at the same TIME (in a batch). The stream can be partitioned along the TagID

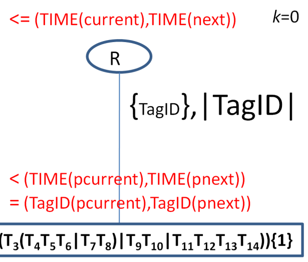

<= (TIME(current),TIME(next))
k=0
R
{TagID}, |TagID|
< (TIME(pcurrent),TIME(pnext))
= (TagID(pcurrent),TagID(pnext))
(T0T1T2(T3(T4T5T6|T7T8)|T9T10|T11T12T13T14)){1}

**Figure 5: RFID Readings Stream Schema**

attribute. Each partition corresponds to a particular TagID (tagid) and has a finite number of readings[3]. Each of these partitions conform to a simple pattern constraint, shown at the only leaf of the constraint tree. In this pattern, $T_i$ is defined as below

$$T_i\text{: R}_{\mathbb{S}(\text{TagID}\leftarrow\{tagid\},\text{ReaderID}\leftarrow\{R_i\})}$$

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

## 8.3 DejaVu

We have used DejaVu [19] to implement this case study. DejaVu is a declarative Pattern Matching System over live and archived streams. It is built on MySQL an open source database system and extends the MySQL language with the MATCH_RECOGNIZE clause [7] to define patterns with semantic windows. The DejaVu implementation uses finite state machines for the execution and internal representation of pattern queries.

## 8.4 Query Rewrites for Early Detection

Assume that the rightmost leaf of the reader tree in Figure 4 (ending in R14) is the destination for 'engines'. A pattern query for detecting misrouted items to this reader is the following.

```
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")
);
```

As depicted in Figure 5, the Stream Schema specification encodes the possible paths as a pattern with repetition of one. Having this knowledge of the reader tree structure, it is straightforward to find the right replacement for readers in route-checking queries. In fact, for each leaf reader, one can replace it with the farthest non-branching ancestor. In the case of the above query, R14 will be replaced by R11 (and similarly, R10 by R9, R8 by R7, R6 by R4).

## 8.5 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 was used to monitor the memory usage.

## 8.6 Early Detection Results

Here, we measure the memory consumed by the windows which mostly maintain partial matches. Early decision making allows us to close the windows earlier, which means less memory consumption. As the results in the table show, the closer we come to the root we branch, the more memory we save off the baseline 1494 KB, which was needed by queries that did not use schema knowledge.

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

[3]At most the depth of the reader tree.

## 9. CONCLUSIONS AND FUTURE WORK

Describing the static properties of a data stream using Stream Schema opens up an important direction toward efficient and 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 for 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. Similarly, relationships between streams would be an important direction, as to investigate if and how foreign key relationships can be suitably expressed. As a related matter, further investigation is needed into alternative ways to react to violations of constraints within a stream.

Stream Schemas may be designed manually, or potentially discovered using stream mining (or pattern mining) [23]. Alternatively, they could be derived from business process or workflow descriptions. Both of these areas deserve further investigation as they will improve the sophistication and usability of stream systems.

*Acknowledgements.*

The work published in this article was partially funded by the MASTER project under the EU 7th Research Framework Programme Information and Communication Technologies Objective (FP7-216917).

We would like to thank Christian Tarnutzer for his contributions to an earlier version of the constraints and optimizations, Kostis Tsoulos for his work on the stream schema implementation on Xerces and Nihal Dindar for her help with the experiments on DejaVu.

## 10. REFERENCES

[1] Coral8 CCL Reference.

[2] Esper reference documentation 3.0.0.

[3] World Wide Web Consortium (W3C). XML Schema. W3C Recommendation, 2004.

[4] Xerces2 Java Parser Project Homepage.

[5] D. J. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] P. M. Fischer, K. Sheykh Esmaili, and R. J. Miller. Stream Schema: Providing and Exploiting Static Metadata for Data Stream Processing. Technical report, ETH Zurich, 2009.

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

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

[24] L. Golab et al. Optimizing Away Joins on Data Streams. In *SSPS '08: 2nd International Workshop on Scalable Stream Processing Systems*, 2008.

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

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

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

[28] F. Korn, S. Muthukrishnan, and Y. Zhu. Checks and Balances: Monitoring Data Quality Problems in Network Traffic Databases. In *VLDB*, pages 536–547, 2003.

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

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

[31] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

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

[33] W. Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, pages 133–191. MIT Press, 1990.

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

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