Streaming XQueryP

Christian Tarnutzer

Master’s Thesis

Databases and Information Systems Group
Department of Computer Science
ETH Zurich
http://www.dbis.ethz.ch/

4th September 2007 - 3rd March 2008

Supervised by:
Prof. Dr. Donald Kossmann
Dr. Peter Fischer
Tim Kraska
Abstract

Data Stream Processing is one of the hottest topics not only in the database community, but in many related fields.

One of the main open issues in this research area are declarative languages to specify the operations on data streams such as filtering, aggregation, sequence matching or more complex expressions.

The DBIS group proposed to use XQuery as a basis for such a stream query language, since it provides a good balance of expressive power and optimisability. In the concept of this proposal the DBIS group has developed an extension of the XQuery Data Model to express infinite data and an extension of the XQuery language to express so-called "windows" over infinite data streams.

The purpose of this master thesis is to investigate the possibilities and the usefulness of approaches to describe the format of the data stream, both in terms of structural (e.g., event sequence) as well as dynamic (e.g., rates) aspects. The information derived from these descriptions should be used to enable optimisations in a declarative stream processor. These optimisations are targeted to reduce response times, memory overhead and processing cost.

To reach the target of stream descriptions, we worked out the following contributions:

1) Streams are to be described. Only if a system has some information about its input, it can optimise its work on it. For streams there is not much description possible at the moment. To describe a stream we will use - depending on the processing model - information about (possibly repeating) patterns and follow-up values (possible ordering, limited values and others). We set up the requirements for such stream descriptions and give a solution proposal on a general stream description model.

2) Since our work started focused on XQueryP we decided to describe XML streams based on XML Schema and extended it by constructs to describe streams.

3) Two different processing models were analysed to see how they use streams, what they assume about streams and what descriptions from our model they could use.

4) Information on streams may change the behaviour of the processing model, so we analysed changes to semantics the stream information can cause in the processing models we analysed in 3).

5) We worked out optimisations for the processing models of 3) that are possible if these information are given.
Contents

1 Introduction .......................................................... 7
  1.1 Motivation ........................................................ 7
  1.2 Objectives ....................................................... 7
  1.3 Outline .......................................................... 8

2 Fundamentals ......................................................... 9
  2.1 XML ............................................................... 9
  2.2 XML Schema ..................................................... 10
    2.2.1 XML validation with XML Schema ...................... 13
  2.3 The XQuery history ............................................ 13
    2.3.1 XQuery .................................................. 13
    2.3.2 XQuery Update ......................................... 15
    2.3.3 XQueryP ................................................ 15
  2.4 Streaming ....................................................... 16
  2.5 XML and streaming ............................................ 17
    2.5.1 FORSEQ .................................................. 17
    2.5.2 Infinite XDM .............................................. 19
    2.5.3 Discussion of XQueryP semantics - update visibility 19
      2.5.3.1 The answer ....................................... 21

3 Describing streams ................................................... 23
  3.1 Streams, problems and existing work ......................... 23
    3.1.1 What is a stream? ....................................... 23
    3.1.2 Common problems with streams ......................... 24
    3.1.3 Current work on problems .............................. 24
      3.1.3.1 Windowing functions .......................... 24
      3.1.3.2 Punctuated streams ............................ 24
      3.1.3.3 k-Constrain ts ................................ 25
  3.2 Why go further? ............................................... 25
  3.3 Use cases for streams and where extensions may help .... 26
    3.3.1 Entry Gates ............................................. 26
    3.3.2 RFID ..................................................... 26
    3.3.3 Financial Data .......................................... 27
    3.3.4 Toolbox .................................................. 27
    3.3.5 Positional grouping .................................... 27
    3.3.6 Miscellaneous .......................................... 28
    3.3.7 Other ..................................................... 28
  3.4 Requirements for stream annotations ........................ 28
    3.4.1 Patterns .................................................. 29

3
## CONTENTS

3.4.2 Information about values of events ........................................... 29

4 Solution proposal for stream descriptions ....................................... 31
  4.1 Patterns .................................................................................. 32
  4.2 Item value information ........................................................... 33
    4.2.1 Restrictions on values of following items ............................ 33
    4.2.2 Information about parallel sequence values ....................... 34
  4.3 Sequence repetition .............................................................. 36
    4.3.1 Number of iterations ..................................................... 36
  4.4 Stream combination .................................................................. 37
  4.5 Interleaving sequences ................................................................ 39
  4.6 Examples ................................................................................ 40
  4.7 Sequence infinity ...................................................................... 40
  4.8 Summary ............................................................................... 41

5 An extension for XML Schema .......................................................... 45
  5.1 Why an XML Schema extension? .............................................. 45
  5.2 The XML Schema extension ....................................................... 45
    5.2.1 From root node to root model group .................................. 46
      5.2.1.1 Root node or Root sequence? .................................... 46
    5.2.2 Stream model groups ....................................................... 47
      5.2.2.1 Nesting of stream model groups ................................. 47
    5.2.3 StreamInfo ...................................................................... 48
    5.2.4 Stream-constraint Definitions ......................................... 49
    5.2.5 StreamNext Schema Component ..................................... 50
    5.2.6 The extensions of XML Schema 1.1 ................................. 51

6 Integration into processing models .................................................... 69
  6.1 Short introduction ................................................................... 69
    6.1.1 XQuery(P) ...................................................................... 69
      6.1.1.1 XQuery(P) and streams ........................................... 70
      6.1.1.2 Refinement of streaming semantics .......................... 71
    6.1.2 Aurora ........................................................................... 72
      6.1.2.1 Aurora and streams ............................................... 73
      6.1.2.2 Optimisation strategy ............................................. 74
      6.1.2.3 QoS and load shedding ......................................... 74
  6.2 Changes in semantics ............................................................... 75
    6.2.1 XQuery(P) ...................................................................... 76
      6.2.1.1 Warnings and aborts ............................................. 76
      6.2.1.2 Operators ............................................................. 76
      6.2.1.3 Scheduling / planning / threading possibilities ........... 77
    6.2.2 Aurora ........................................................................... 78
      6.2.2.1 Warnings and aborts ............................................. 78
      6.2.2.2 Operators ............................................................. 78
  6.3 Optimisations made possible ..................................................... 78
    6.3.1 XQuery(P) ...................................................................... 79
      6.3.1.1 Query rewritings ................................................... 79
      6.3.1.2 Operator optimisations ......................................... 86
      6.3.1.3 Memory optimisations .......................................... 87
      6.3.1.4 Scheduling / planning / threading possibilities ........... 89
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 More on integration into processing models</td>
<td>90</td>
</tr>
<tr>
<td>7 Information on dynamics and their usefulness</td>
<td>91</td>
</tr>
<tr>
<td>7.1 Use case examples</td>
<td>92</td>
</tr>
<tr>
<td>7.2 Optimisation example</td>
<td>92</td>
</tr>
<tr>
<td>8 Semantic analysis of FORSEQ</td>
<td>95</td>
</tr>
<tr>
<td>8.1 Extended Basic Concepts</td>
<td>95</td>
</tr>
<tr>
<td>8.2 Analysis type</td>
<td>95</td>
</tr>
<tr>
<td>8.3 FORSEQ</td>
<td>96</td>
</tr>
<tr>
<td>8.3.1 Normalisation</td>
<td>96</td>
</tr>
<tr>
<td>8.3.2 Static infinity analysis</td>
<td>96</td>
</tr>
<tr>
<td>8.3.2.1 Without schema extension</td>
<td>97</td>
</tr>
<tr>
<td>8.3.2.2 With schema extension</td>
<td>98</td>
</tr>
<tr>
<td>9 Related work</td>
<td>107</td>
</tr>
<tr>
<td>9.1 Loss pattern</td>
<td>107</td>
</tr>
<tr>
<td>9.2 Punctuated streams</td>
<td>107</td>
</tr>
<tr>
<td>9.3 k-Constraints</td>
<td>108</td>
</tr>
<tr>
<td>10 Conclusion and future work</td>
<td>109</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Since the beginning of XML the range of applications does not stop growing. All around the world people of all imaginable domains are steadily discovering XML as a very potent language for their domain specific applications. In the same time, data stream processing has become a hot topic as well. Both come with some problems for people using it: XML imposes an impedance mismatch, since commonly used programming languages do not have a type for handling XML data. Therefore, handling (event) models such as SAX or DOM are used as intermediate format. The same problem exists for common databases. They either store XML as blobs or split it up into the single elements and store their interconnections as relations. Streaming on the other hand suffers from the fact, that it is by nature dealing with infinite data sources and in most cases you can see a streamed element only once, afterwards it is gone.

Solutions for the problems of both are subject to ongoing research and for both there are some results. With XQueryP, the databases and information systems group (DBIS) of ETH Zurich has developed a language extension of XQuery, that allows to write complete programs handling XML, instead of having to use XQuery inside of a sequential programming language and again de- and re-factor XML into and from programming language specific data types. XQuery and XQueryP will be introduced more detailed in Section 2.3.

The next step was to combine these two topics, to allow XQueryP to handle streams. First work on this topic has been done by Kraska [1] by introducing a window construct to execute queries on subsections of a stream and an extension to the XML Schema Data Model (XDM) to allow and mark possibly infinite elements. Since window work on streams is not everything of the combination of XQueryP and streams, this master thesis will go further and make XQueryP fully aware of streams by extending XML Schema to enable optimisations and early detection of problems arising with infinite data.

1.2 Objectives

The objective of this master thesis is to enable XQueryP to process (possibly infinite) streams. [1] introduced both an extension to the XQuery Data Model (XDM) to allow and to indicate possibly infinite sequences and a syntactic extension of XQuery to use windows on streams. But just marking sequences as possibly infinite is not enough. This work wishes to make XQueryP fully aware of and able to handle streams. To achieve this goal, we first develop a general model to describe streams and then propose extensions to XML Schema to describe streams more detailed with the aim that processors can optimise XQuery queries based on this Schema information according to memory
management, threading and query rewrites. To show that these extensions are useful we analyse their impact on different processing models, including XQueryP.

1.3 Outline

The rest of this thesis is structured as follows: In Chapter 2 we will introduce to the basic concepts of XML handling and stream processing.

In Chapter 3 we will have a closer look at stream processing and show why we need means to describe streams more detailed, giving information about their structure and content as well as forming the requirements. In Chapter 4 we present our solution proposal for stream description, while the implementation using XML Schema definitions and syntax will be described in chapter 5. In Chapter 6 we will introduce and analyse some processing models that can use our stream descriptions regarding possible semantic changes caused by our extension, as well as the optimisations that are made possible by our extension.

 Possible descriptions of dynamics and their applications in other processing models than XQuery or its extensions are discussed in Chapter 7. To prove that our claimed advantages of such XML extensions are true, we give in Chapter 8 an infinity analysis of the semantic of FORSEQ, which was - until now - known to be undecidable in terms of infinite evaluation in most cases. Chapter 9 describes related work, whereas in Chapter 10 we will conclude and give a short overview about future work and possible applications.
Chapter 2

Fundamentals

To understand the functionality of XQueryP on streams, we will introduce the reader to the basic concepts of the underlying principles. As a first overview, XML is a standardised format to model data, XQuery is used to run queries on XML, XQuery update modifies XML documents and XQueryP enables the user to write programs using XML completely by XQuery, without a single transformation from XML to another data format or vice versa. Streaming is an extension to documents to be infinite as well as data flowing by in front of a reading/processing user or machine instead of being finite and static.

2.1 XML

XML [3] is a simple very flexible text format derived from SGML. It was originally developed to meet the challenges of large-scale electronic publishing. The main difference to HTML is that XML only stores structured content, telling nothing about the potential presentation and that XML tags can be named completely free, whereas HTML has a fixed set of elements. Basically, XML models data as text-based tree structure of elements. Saved as text, XML looks very similar to HTML, showing text interleaved with markup elements (tags) describing the hierarchical structure of the data. Because of this text based representation the format is well readable by humans and by computers. As example we will use in all parts of the fundamentals section a collection of publications. For the XML part, we present the XML document containing the information about the collected publications:

```xml
<?xml version="1.0" encoding="utf8" ?>
<publications>
  <publication type='book' ISBN='345-5804736233'>
    <title>XML in the wild</title>
    <author aid='12'/>
  </publication>
  <publication type='master thesis' ISBN='234-4563638564'>
    <title>Streaming XQueryP</title>
    <abstract>Data Stream Processing is one of the hottest...</abstract>
    <author aid='28'/>
  </publication>
  <authors>
    <author aid='28'/>
  </authors>
</publications>
```

1Hyper Text Markup Language, http://www.w3.org/TR/html401/
All concepts of XML are visible here. There are freely named elements (publication, title, author, abstract), attributes describing the elements (type, ISBN) and a human and machine visible hierarchy (Title, Author and Abstract are parts of the publication). This simple form of data representation has several advantages: All XML documents have the same structure, although the elements have different names, are therefore platform independent and needed parsing algorithms are relatively simple to write. Since many data in our world has some structure, the hierarchical structuring used by XML should fit many application domains, if not all.

Internally, once a computer has parsed an XML document, it is modelled as a tree of elements, as shown in Figure 2.1 for the publications document from above. To handle the data internally, XML uses XML Information Set [4], its own data model that stores all information to describe an item completely.

Figure 2.1: XML instance tree

2.2 XML Schema

"The purpose of a schema is to define a class of XML documents"[5]. XML Schema [5] defines the needed constructs to describe all parts of an XML document. There are:

- Type definitions
- Attribute declarations
2.2. XML SCHEMA

- Element declarations
- Annotations

Type definitions are either made globally or inside element declarations. They allow the user to define what an element of this type is composed of. Does it have attributes? Does it have child elements (complex types)? May it be empty? Or does it just contain character data (simple types)? If it is a complex type: Are there only elements as children or are they mixed with text? If it is a simple type: Is the character data constrained?

Declarations (element and attribute) declare where elements or attributes do appear. They are either used globally to be referenced inside or directly in type definitions. They mainly contain the following information: name, type, and namespace. For attributes, the usage is listed also to tell the user if this attribute can (optional), must (required) or must not (prohibited) be used in the element containing the declaration or reference. For elements, there may be additional information like its minimal (minOccurs) and maximal (maxOccurs) occurrence.

In addition, elements and types can have some restrictions on their extendibility. Types are all derived from an ur-type. Or more understandable, all types are a derivation of another. By following this derivation chain backwards, you will be able to reach the ur-type from any existing type. Derivations can be done in two ways:

- by restriction:
  - limiting some value to be one of a defined subset (enumeration)
  - by enforcing some pattern the value has to follow
  - removing some elements from the possible children
- by extension:
  - adding attributes or children
  - relaxing value restrictions

If a type can be used as base for restriction or extension can be controlled by the attributes block and final in its definition.

Annotations can be placed in any schema construct to describe it more deeply in human understandable words. The following example shows the XML Schema for the above publications document.

```xml
<x:schema xmlns:x="http://www.w3.org/2001/XMLSchema">
  <annotation>
    <documentation xml:lang="en">
      Schema for publications
    </documentation>
  </annotation>
  <xs:element name="publications">
    <xs:complexType>
      <xs:element name="publication" type="pubType" maxOccurs="unbounded"/>
      <xs:element name="authors">
        <xs:complexType>
          <xs:element name="author" type="authorType"/>
        </xs:complexType>
      </xs:element>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
The above XML Schema makes use of all basic concepts:

- The root node is one of the globally defined ones, here only `<publications>`.
- There are global type definitions (`isbnType`, `pubType`), as well as inside (anonymous) type declarations (for publications and publication type attribute).
- The `isbnType` and the publication type attribute type show derivations by restriction (the latter is a shorthand for restriction since neither `<restriction>` nor `<extension>` are given).
- Inside the author element we use a `<publications/>` element by reference. This means, inside the author element, can be another publications element as the one described globally to report the books the author has published. In the case of this example this is not used in a very good way, since it allows infinite recursions and every publication stated inside an author must be described again completely. It is just used to show how `ref` works.
- In the publications one can see the definition of a unique value: All `<publication>` inside `<publications>` must have a unique value for the ISBN attribute.
- There exists also `<xs:key>` which can be referenced by `<keyref refer="keyname">`. Keys as well must be unique. Keys are used in our example to reference the authors inside of a publication. Like this, each author has to be written only once in the authors element and can then be referenced by its aid attribute. This would also be the right approach to list the publications of an author (e.g. reference by ISBN).

2.2.1 XML validation with XML Schema

If an XML document and its XML Schema are present it is possible to validate the document. This means the parsed Infoset is checked if its items conform to the declaration rules in the given schema. In the same step, the Infoset is enriched by additional information like types and validation results. The output of the validation is named a PSVI, a Post-schema-validation Infoset. The XML Schema documents from W3C\(^2\) define for each schema component its validation rules. They tell what has to be fulfilled for an item to conform to its schema definition. The outcome of the validation will be stored in the PSVI. All information added during the validation step of an item is declared in the according schema component definition’s PSVI Contributions section in the W3C documents.

2.3 The XQuery history

2.3.1 XQuery

XQuery [8] is a declarative, high-level, lazy evaluated query language for XML documents, created to exploit the flexibility of XML to represent many different kinds of information. It accesses the data with a syntax similar to SQL but guarantees most of the time to deliver the results in document-order. It is side-effect free and read-only.

Every query consists of a possible prolog to declare functions and variables and one expression, which can then again contain subqueries to return the operators of the surrounding expression. To

\(^2\)W3C is the World Wide Web Consortium. To be found under http://www.w3.org
refer to the fields of the queried XML file(s), XQuery uses XPath [10, 11]. To handle data internally, XQuery uses XDM [12] - its own data model "XQuery Data Model". Unlike in XML itself, in XDM all values are a sequence of items. Nested sequences are flattened. Query output is not guaranteed to be serialisable to XML, because XDM is a superset of XML Infoset. Nevertheless, all outputs from a query can be used as input for a next XQuery query. Variables, function parameters and return values can be typed using the XML Schema types [7].

Most queries are written using a small number of constructs: FLWOR, if-then-else, XPath expressions and element constructors. These few constructs are enough for many needs. We will demonstrate them shortly by some small examples, for all other possibilities of XQuery, please refer to [8].

We will start with the simplest of the constructs, the element constructor:

```xml
<hello>Hello World</hello>
```

This is already an XQuery, it just returns an element with name "hello" and text content "Hello World".

The next step is to use path expressions to use some values of another XML document:

```xml
<hello>{doc('hello.xml')/helloworld[1]}</hello>
```

This will still create an element with the name "hello", but as content it uses the content of the first "helloworld" element of the document "hello.xml". The reader can verify that this is just standard XPath usage. The curly brackets denote, that the result of the contained sub-expression should be inserted instead of the expression itself.

Let us go a step further and use some conditionals:

```xml
if doc('test.xml')/isTrue eq 'true'
then <hello>{doc('hello.xml')/helloworld[1]}</hello>
else ()
```

As with all functional languages the last function defines the return value. Or in other words, the last returned value that is not consumed by some other function is the return value of the overall query. Therefore, the above query outputs the same as the one before, but only if the content of the <isTrue> element of document "test.xml" is the string "true". It is important, that the if-then-else in XQuery always needs an else-part. If there is nothing to do in the if- or else-expression, one must give the empty sequence.

Our last step will show the FLWOR expressions, the most powerful construct of XQuery:

```xml
let $persons := doc('persons')/person
for $who in $persons
where $who/name eq 'Peter'
order by $who/surname ascending
return $who
```

Every FLWOR expression must have at least one for or let and a return expression. let binds the returned sequence of its expression to the given variable. Note, that since XQuery is a declarative, functional language, variables can only be bound once. There is no possibility to update variable values, except for for. for iterates through the sequence returned by the expression after in and binds each element of the sequence iteratively to the given variable. This is done in the order in which the in-expression returns the elements unless an order by expression is given. order by denotes at least one field which has to be used to re-order the element stream coming from the
2.3. THE XQUERY HISTORY

in-expression. order by has many options which will be skipped here for simplicity but can be found in [8]. If needed, a where-expression can filter out a subset of the sequence elements. If for an assignment the where clause does not evaluate to true, the for immediately restarts and takes the next element of the sequence. For each element passing the where, the return-expression is evaluated. Therefore, the above query returns every <person> element of the "persons.xml" file whose <name> element has the content "Peter" sorted ascending by the content of its <surname> element.

2.3.2 XQuery Update

XQuery Update (Facility) "defines an update facility that extends the XML Query language, XQuery. The XQuery Update Facility provides expressions that can be used to make persistent changes to instances of the XQuery 1.0 and XPath 2.0 Data Model" [13, Abstract]. XQuery Update allows to insert, delete and modify nodes. When modifying, there are two possibilities: modify some of its properties while preserving its identity or create a modified copy of the node with a new identity.

XQuery Update introduces its own set of expressions, namely insert, delete, rename, replace and transform. We will not explain them here, all needed information is found in [13]. The important point of XQuery Update is that updates are applied as a bundle at the end of the query: An XQuery result now consists of an XDM instance and a "pending update list" [13, Section 2.1]. Pending update lists are an "unordered collection of update primitives, which represent node state changes that have not yet been applied" [13, Section 2.1]. If the outermost expression of an XQuery query returns a pending update list, the update primitives are applied implicitly. Note that an expression can only either return updates or values. The other return part has to be empty (or the empty sequence has to be an XDM instance). Like that, all update expressions operate on the initial data, called a snapshot. This snapshot ends with the application of the updates.

2.3.3 XQueryP

XQueryP [14, 15] is an extension of XQuery Update and XQuery to extend them with sequential execution and instant update application. With this extension it is possible to develop XML applications without any host programming language and therefore without wrapping XML data to objects or other host-programming language conform constructs. In other words, XQueryP completes the XML development tools with application logic. Like that, an application can change its state during execution, update variables and apply updates immediately. To combine all these advantages together, there must be a well-defined order on the "side-effecting expressions to have a deterministic result" [14, Section 2].

The extensions are:

- *sequential execution mode*, to be enabled in the application prolog. If it is activated, updates are applied after each expression. For example, they get applied after each iteration of the for construct.

- *blocks*, denoted by {...}. A block contains a sequence of expressions separated by semicolons. Like that, there is now the possibility to call various functions before the last which will define the return value. Variables defined inside a block are only visible within this node. Therefore, every block has its own scope.

- *assignments* to assign values to variables and therefore maintain and update the state (state is defined as "all the variables that are in scope during the evaluation of a given expression" [14, Section 2]).
• allow mixing of updating and non-updating expressions. Note that there are still some con-
straints as to where updating expressions can appear (e.g. not in predicates).

• Additional control flow constructs: while with test and body for another loop variant, break to
leave the nearest enclosing FLWOR or while expression, continue to advance to the next iter-
ation of the nearest enclosing FLWOR or while expression, return to terminate the execution
of a function body early and return a result.

• try-catch expression for exception handling.
• explicit atomicity to define parts of the code which can be rolled back in the event of an error.
• Web service handling to create lightweight web service clients and servers.

For details on the extension please refer to [14, 15]. The important point about this development of
XQuery is that with XQueryP it has now all tools to develop and run XML applications with one
single programming language.

2.4 Streaming

"A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by time-
stamp) sequence of items. It is impossible to control the order in which items arrive, nor is it
feasible to locally store a stream in its entirety" [17, Section 1]. The problem setting in streaming
is to work on possibly very fast arriving data, keep low latency, use small amounts of memory and
at the same time produce correct or appropriate results.

As an example, think about network traffic controlling. The proxy machine, where all traffic
of a company leaves the building, reports each packet to a log&analyse server. This server tries to
detect abnormalities in the network traffic by looking at each and every network packet going in
or out the company network and may be grouping them by IP, user or application. Since network
traffic never ends, it is clear that the server cannot wait for all input to arrive before computing
something on it. It has to analyse the packages and delete them soon to have enough memory for
the next bunch arriving. For this reason, it is mostly able to see each packet once and therefore has
to work on-line.

This introduces some problems to solve [17]:

• Blocking operators cannot be used.
• No backtracking possibility.
• Must react quickly to unforeseen data.
• Long running queries may encounter different system environment, rate changing streams.
• React on increasing amounts of data (scalability).

There exist many attempts to solve these problems with streams or at least to work on them as
good as possible:

• Look only at the last x packages (window based queries).
• Use approximate summary structures (e.g. synopses [18] or digests [19]) - The answers become
approximate and may not be exact.
2.5 XML and streaming

Section 2.4 shows the problems that are to be solved by stream applications or stream-enabling applications. In combination with XQuery there are two main problems. First, there is no window or sub-sequence selection in XQuery and second, the XDM does not allow any part of itself to be infinite.

Kraska defines in [1] two more extensions to make XQuery able to handle streams:

- Forseq to select sub-sequences based on values in a stream
- infinite XDM instances

In the following we will give a short introduction to both extensions.

2.5.1 FORSEQ

This new construct selects sub-sequences from a sequence. In general usage it creates all possible sub-sequences of the elements in the source sequence and binds them iteratively to the given variable like the for construct. For every binding, if the where expression (if given) evaluates to true, it evaluates the return expression.

```
forseq $seq in $stream
  where $seq[1]/@action eq 'start'
    and $seq[length($seq)]/@action eq 'end'
    and length($seq)-1 eq 4
  return $seq
```

In general forseq, all sub-sequences are built, meaning that they do not have to be continuous. However, they must be in document order. From the source sequence (1,2,3,4), forseq would bind the following sub-sequences: (1), (2), (1,2), (3), (1,3), (2,3), (1,2,3), (4), (1,4), (2,4), (1,2,4), (3,4), (1,3,4), (2,3,4), (1,2,3,4). The ordering of the bindings is not defined.

There is a second usage, the windowed forseq, which binds only continuous sequences. One can use three types of windows, each one restricting the previous one a bit more (shown also in Figure 2.2):

- landmark windows can be "defined freely". The only condition is that there is only one binding for every start-end position combination. Once the start condition matches, the window is opened and evaluated every time the end condition matches.
- sliding windows are constrained in the way that for every start position there is only one binding. The first time the end condition matches, the window is evaluated and removed.
- tumbling windows are constrained even further. There is only one open tumbling window at any point in time. If a window is opened, the start condition is not checked until that window got evaluated and removed.

For windowed forseq, the start and end conditions have to be given explicitly. They can be set to fn:true() and be checked in the where clause, but by setting them at their intended places instead of in the where reduces the number of open windows. This is because then they are only bound if they match these conditions, whereas if the conditions are checked in the where, every continuous sub-sequence is bound and only evaluation is constrained. A smaller number of open windows helps to reduce memory needs. Every window is evaluated if the end condition matches or if the source sequence ends and it is still open.
forseq $win in $stream landmark window
start when $win[1]/@action eq 'start'
end when $win[length($win)]/@action eq 'end'
where length($win)-1 eq 4
return $win

There are multiple shorthand notations defined for windowed forseq:

- **curItem**, **prevItem** and **nextItem**, to use after start/end and before when and to be given a variable name. They bind the current, previous or next item respectively to the given variable.

- **position**, to be used like the -Item, but binds the position of the current item to the given variable.

- **force**, to be used before the end. If given, windows are only evaluated if the end condition matches and no more at the end of the stream.

The following example uses all this syntactic sugar:

forseq $win in $stream landmark window
start position $sp when $stream[$sp]/@action eq 'start'
force end curItem $c, nextItem $n when $c/name ne $n/name
return $win

Figure 2.2: The three window types - and one of them using force.
2.5. XML AND STREAMING

2.5.2 Infinite XDM

Every instance of the XDM is now allowed to be an infinite sequence. This is limited to the instance to make things not too complicated. Like that, only the outermost element can be infinite. Still this introduces some problems with blocking operators, if they really have to wait with outputting until they have read the end of the input, which will never arrive. With this extension, XQuery operators have to be revised to guarantee output before having read all data if they are non-blocking or stream-dependent non-blocking operators (they are blocking if one of the input streams is infinite), whereas information-dependent non-blocking operators (non-blocking if they know something about their input stream(s), e.g. that the inputs are ordered) do not have to output early and can be treated as blocking. It is left to the responsibility of the user to detect and circumvent combinations of blocking operators to enable early outputs in stream applications.

As an additional extension a mark for functions that return infinite sequences is proposed. \( \ast \) already denotes possibly infinite return sequences, but the new \( \ast \ast \) tells that the return sequence will be infinite. Like that a data flow analysis can throw errors during static analysis. Therefore, a function that returns an infinite sequence is marked like this:

\[
\text{fn:fibonacci($arg as xs:integer, $arg2 as xs:integer) as xs:integer}\ast\ast.
\]

2.5.3 Discussion of XQueryP semantics - update visibility

A last question about the surrounding settings concerns the update semantics of XQueryP. XQueryP handles updates instantly, as to say after each expression the updates are applied. There is no waiting until the end of the query and finally applying the collected update set as in XQuery update.

Now regarding windows in streams, there arises the question about visibility of instant updates to other windows. If we handle XQueryP as usual, meaning applying the generated updates instantly, should other windows containing the same elements see these updates or not? If we enable this, the elements seen by the windows depend on the order in which the windows are produced. Since this order is not predictable, making changes viewable to other windows, changes become non-deterministic.

One might now wonder why the order is not predictable, since in [1] it is said that windows are produced when their end condition is met and if there are more than one for the same end condition, they are ordered by starting element. For single window production this is ok, but as soon as we have early productions or parallel windows at all, there will be an uncertainty about which windows updates appear at which instant.

If we say that windows do not see each others updates, what is the output of the windows and what of the query? If each window produces its output, does the query merge the outputs after all windows have closed? What about infinite queries? If the outputs are not merged, there will be duplicate outputs for the same element, so we will have no idea how to rearrange the stream again.

These problems lead to the discussion about the definition of the output of a stream query itself:

- The most convenient answer would be that stream queries produce a new stream as output, independent from the input stream. For this version it would be no problem if the output contains multiple variants of the same input elements. The problem with an independent output stream becomes obvious as soon as object identity is used. Since the output is a newly generated stream, the object ids will not be the same anymore.

- If we say on the other hand, that we operate directly on the input stream, parallel as well as early producing windows will be a problem if they process updating queries:
XQuery update arises problems at the end of the query: After all windows are closed, the update sets must be applied. But if we have more than one window updating the same element properties, we have again the race condition discussed before.

XQueryP applies updates instantly. That brings another problem: Assume the stream `<A>1</A> `<A>2</A> `<A>3</A> `<A>4</A>` and the following query:

```xml
for $w in $seq sliding window
start curItem $s when $s eq 1 or $s eq 2
end curItem $e when $e-$s eq 2
return do replace value of $w[2] with 3
```

Step 1 opens a window because `<A>1</A>` eq 1. Step 2 opens a next window because `<A>2</A>` eq 2 and adds `<A>2</A>` to the first window. Step 3 adds `<A>3</A>` to both windows, the first gets closed and evaluated. Before, the windows are (only values shown) `{1,2,3}` and `{2,3}`. After the production of window 1 the second window is `{3,3}`. But since the first element of this window is neither 1 nor 2, it should have never been opened! This problem is shown in Figure 2.3.

![Figure 2.3: The problem of update visibility](image)

Regarding only multiple windows of one query, the following outputs are possible:

- An output stream for each window. With this variant we lose the view of the input stream
- One output stream for all windows together. This single stream can be again arranged in different ways:
  - An interleaved output stream, where every window just injects its next output event as it is ready. This would be the easiest one of the single output stream variants, since every window can work on its own. It is just like having an output for every window and bundling them together. As a possible drawback, we loose object identity and may get duplicate elements since more than one window might output it.
  - An output stream in the order the windows would have had in non-parallel production mode. This variant again looses object-identity and has the problem that we will have to cache all produced output for every window until it really gets closed.
- a merged output stream, meaning that updates of a specific element are done to one and the same element that goes finally to the output. This is the most difficult version, since it requires object-identity to be kept and we have to cache the whole output stream, releasing each element as soon as all windows it is part of got closed. Note, that this is the only variant where we could speak about modifying and outputting the input stream. Modifying means we update the input elements directly and may be (by normal XQuery) inject newly created elements.

For a concise recapitulation of the above listed characteristics of the different versions, see table 2.1.

<table>
<thead>
<tr>
<th>mode</th>
<th>object-identity</th>
<th>cached</th>
<th>input order</th>
<th>output stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple windows</td>
<td>lost (duplicated)</td>
<td>no</td>
<td>lost</td>
<td>newly created</td>
</tr>
<tr>
<td>interleaved</td>
<td>lost (duplicated)</td>
<td>no</td>
<td>lost</td>
<td>newly created</td>
</tr>
<tr>
<td>ordered</td>
<td>lost (duplicated)</td>
<td>per window</td>
<td>lost</td>
<td>newly created</td>
</tr>
<tr>
<td>merged</td>
<td>kept</td>
<td>per element</td>
<td>kept</td>
<td>is input stream</td>
</tr>
</tbody>
</table>

Table 2.1: different window production modes

The discussion will also be spread not only over parallel windows but gains a further dimension regarding multiple queries on the same stream. If the output streams of these queries should be compared, element identity would help but then we would have to operate on the same stream at the same instant.

2.5.3.1 The answer

The XQuery working draft says that XQuery uses snapshot semantics. Therefore, the moment a query or window gets evaluated, its input were already bound and no changes to the original input is visible inside the query. In addition, the result is generated completely, causing a new identity for the changed or output elements.
Chapter 3

Describing streams

Problems that arise when using streams reside in memory management and concurrency. Meaning, it is no more possible to materialise all items passing by during the execution, since if a stream is infinite, there are infinite items and therefore infinite memory needs. So we have to use pipelining, produce - if possible - results while still feeding input, materialise only where really needed. The problem with concurrency is, that if we select sequences from the stream to work on, there may be some infinite sequences that prevent all following ones from being seen or processed. So we should enable parallel handling. But how? Using thread pools, launch new threads for new sequences? Or just pre-detect infinite sequences and delay them until the finite ones have been handled? Note that there may also be an infinite number of finite sequences.

So we need to have some methods to detect such problem-queries and adapt our way of handling them or even abort the execution if there is no possible way of handling the query.

3.1 Streams, problems and existing work

This section analyses how one defines a stream intuitively and what are common problems as well as how these problems are handled today. Later on, in Chapter 6, we will have a look at how different processing models define their streams and how they extend or change this first definition.

3.1.1 What is a stream?

A stream is a series of items carrying information. The items are coming either from a storage or are freshly generated. Storage might be a database or simply a hard disk residing in any computer around the world. If the items are generated, the generating "device" might be some sensor measuring and reporting e.g. temperature in a room every second or a feed author writing e.g. articles for an RSS-Feed or blog. The series might be infinite. Generated streams like measurements from a weather sensor are usually infinite. Finite streams are usually coming from storage e.g. the result of a database query.

Streams are accessed by long-during or even infinite and ad hoc queries. A long-during query often uses window functions or aggregates over all tuples arriving from the moment it is asked. Therefore, every incoming item might trigger a new output or is simply integrated to the aggregate.

3.1.2 Common problems with streams

One common problem on working with streams is that, at least with infinite streams, it is obviously impossible to store all arriving items to be able to answer queries after "the last" item has arrived. Therefore, only aggregating queries that can be calculated cumulative or so-called "rolling aggregates" are possible to answer. This means e.g. for the calculation of an average value only
two values are stored, the current sum and the number of arrived items. For every item, its value is added to the sum and the counter is incremented. After that, the current average might be calculated and output. This imposes a big problem for blocking operators like e.g. sort, which has to see all input before being able to release all items with the lowest value (they have to make sure e.g. that the assumed lowest value really is the lowest).

A similar problem like the one for sort will show up when regarding windows: If we have conditions for a function about when to return or do something, it is not possible to say if this condition will ever be met and therefore if a function will terminate.

3.1.3 Current work on problems

3.1.3.1 Windowing functions

Windowing functions handle the problem that one can not store all incoming tuples forever. They solve the problem by calculating functions only on a selected subset of the stream input. There are two sorts of windows: time based and count based.

Time based windows memorise arriving tuples for a defined timespan, e.g. "of the same hour", "past five minutes" or "of the hour value '7'". The first would calculate its result and drop all memorised tuples as soon as the first tuple of the next hour arrives, and begins memorising again, while the second calculates its result over the memorised tuples each time an arriving tuple has a new minute value, drops all tuples with the oldest minute value and adds the newly arrived tuple to have again five minute values stored. The last one is self explaining. The problem with all these windows is that they only work if exactly defined by the programmer as if you observe a change in hour/minute field, assume that the previous value will not arrive anymore. Or in other words, the input has already to be sorted by these fields.

Count based windows keep a defined number of items. So as soon as the defined number is reached, the result is calculated. For each following item the oldest item is dropped, the new one is added and the next result calculated.

Another grouping for windows divides them into moving and fixed window. While the former are evaluated again and again for new tuples and older ones are dropped, the latter are calculated as soon as some condition is fulfilled and then closed and removed.

3.1.3.2 Punctuated streams

The work of Tucker et al. on stream punctuations [23, 24] wants to solve the problem with blocking operators, state reduction and information about sorted streams. Instead of only receiving items by the stream, one will also get so called punctuations which differ from normal items only by some mark. While punctuations look the same as normal items, their semantics is another: They do not deliver a new value to be processed by the operators, but tell that after a punctuation p no item will contain the values matched by those of p. Therefore, punctuations can contain the same values as items but also ranges to match more values in one step.

To show how punctuations work, assume a sort operator receives the items with values 3, 7, 1, 4, 9, 10, 15, 5, followed by a punctuation with content [4,7], the item 2 and the punctuation [,3]. The operator can output the sorted sequence 1, 2, 3, 4, 5, 7 without possibly missing the item 0 or 6, since the punctuation tells us that no item can arrive containing the values from 4 to 7 and from the minimum to 3. In addition, the operator can drop all memorised items with matching values, since they are output and not used for sorting anymore. Internally, the two punctuations can be composed to [,7]. After sorting all received items with values in this range, the operator will output the sequence and the two or possibly the composed punctuation to share its information with the following operator(s).
3.2. WHY GO FURTHER?

Since punctuations have the same fields as the normal streamed items, the patterns can be combined over field boarders, therefore a punctuation could state e.g. that no more item of which field "x" matches \[.,100\] and field "y" matches \[50,70\] will arrive.

Using punctuations, elsewise blocking operators can output parts of the results early, whereas another (possibly intersecting) group of operators can reduce state.

3.1.3.3 k-Constrain ts

Babu et al. describe in [31] the notation of k-Constrain ts. k-Constrain ts tell a certain behaviour of the stream using constraints within a limited range, denoted by the adherence parameter k. They introduce three types of constraints:

- **referential integrity constraints** are constraints on many-one joins. They define that a referential integrity constraint on a many-one-join on stream S1 and stream S2 with adherence parameter k tells that "for a tuple s1∈S1 and its unique joining tuple s2∈S2, s2 will arrive within k tuple arrivals on S2 after s1 arrives" [31, Section 1.3.1]. If k=0 this is termed strict referential integrity and means that s2 will always arrive before s1. This constraint will allow e.g. to reduce join memory since joins can only happen in some range of tuples.

- An *Ordered-Arrival Constraint* on stream attribute S.A with adherence parameter k tells that "for any tuple s in stream S, all S tuples that arrive at least k + 1 tuples after s have an A value ≥s.A." [31, Section 1.3.2]. This means, a tuple arriving out of order arrives at most k places away from the place it should appear. Ordered-Arrival Constrains allow e.g. unblocking of sort operators.

- A *Clustered-Arrival Constraint* on stream attribute S.A with adherence parameter k tells that "if two tuples in stream S have the same value v for A, then at most k tuples with non-v values for A occur on S between them" [31, Section 1.3.3]. Clustered-Arrival Constrains allow e.g. the unblocking of grouping operators because we know if during k items no more item fitted into a group G we can output its items and know that there will be no more item of this group.

These constrains are used by the "architecture for exploiting k-Constraints", called k-Mon, described in [31, Section 1.5]. k-Mon gets a continuous query from the user, generates an initial query plan based on any current k-Constraints known on the input streams. A parallel running instance, the *Constraint Monitoring*, gets informed about potentially useful constrains on the running query and keeps track of the fitting value for the adherence parameter of every such constraint by monitoring the input streams the constrains are defined on. Whenever this parameter value changes, the monitoring system informs the execution part which adapts its behaviour to make use of the new knowledge. If the value of a parameter grows constantly or just raises over a defined threshold, the according constraint gets unusable and its optimisations will be deactivated until the value drops again.

3.2 Why go further?

As the previous section shows, there are already some solutions to the common streaming problems. This work will go further by describing streams from outside. From outside, one is able to give more general but also more detailed information about the full view of the stream, as well as about item details. By describing we mean giving the information in one package, preparing the "device" to know what it will work on instead of delivering new ways to operate on streams (e.g. windowing functions) or letting the stream itself inform about its future items (e.g. punctuations). Outside,
information can be used by any handler at any time, if needed, not only to unblock operators or reduce their state, but also to optimise queries by knowledge about the tuples to expect or even reenable original operators by knowing that values of certain fields obey some ordering like e.g. "strictly increasing".

If not using for operator unblocking or state reduction, these additional information can be used without having to implement new operators which understand e.g. punctuations or can be told to output result "up to value x".

The information about orderedness, or just how the series of values of certain fields in the items will behave, delivers possible solutions to the problem of termination conditions.

3.3 Use cases for streams and where extensions may help

All these use cases are taken from [1], therefore we will use the window and query types used there, even if we are not narrowing on XQuery.

3.3.1 Entry Gates

This is the most often used scenario, handling entry control systems. Each person allowed to enter or leave the building gets an id card. A card reader at the entry reads the information on the cards passing by and sends them to a logging or controlling instance, therefore generating an infinite stream of events.

Independent of the listed categories, queries on windows on this stream could be optimised if the systems has some more information about the stream instead only that it is infinite. This is because most of the queries on entry gates use windows to get the sequence of events between the entry and exit events of a specified person. Now if the system knows, that in this stream each entry event of a person will be followed by a leaving event before any other entry of the same person, it can assume that the use of a sliding/tumbling window is exactly what the user wants, even if he does not specify it. Or in the other way, if the user queries for a landmark window but uses the WHERE to get only the ones from an entry to the first leaving event, the system could change the query safely to a sliding/tumbling window and therefore remove the WHERE and use all its implemented optimisations for sliding/tumbling windows (generating much less windows than landmark would). The differentiation between sliding and tumbling depends on if the stream is prefiltered and therefore split into parallel streams containing events of one person only (tumbling), or if the user wants one window for each user in one single stream (sliding).

3.3.2 RFID

Using RFID in production lines (each item in production has an RFID tag, every production step reads this tag for monitoring e.g. where an item is in the production line) gives a lot of hints about possible query optimisations. Since RFID are used here to follow an item on the production line, we know that every item will produce the same sequence of events - just interleaved - and that every event will be generated only once for every item as well as that every product leaves the line at some time for sure. So again we know - depending on the query targets - that for each product one sliding or tumbling window may be enough, since each product will go through the packing/finishing stage only once.

The next decision we can make is about early productions. If a window starts/ends per product, we now for sure that it will close and can therefore enable early production, running computations of the return expression even while windows are still open. This needs parallel window production too.
3.3. USE CASES FOR STREAMS AND WHERE EXTENSIONS MAY HELP

3.3.3 Financial Data

For stock exchange events, there is no known pattern in the stream besides a standard layout of messages, since its free market. But annotations about the dynamics are possible. So we can give some information about the time between generated events. Remember the query for windows, which have as start all events where its hour in the timestamp is bigger than the one of the previous and as end all events where it is smaller than the one of the next. Since it is possible to get no event for more than 60 minutes, such a window might cover two hours instead of one (it will be open for two hours but report only about events in one hour). A wrong solution would be to write the query to check for the hours being exactly bigger or smaller by 1 (then the possible two hour window will never be closed) or start and end hour being the same (window will be of appropriate size, but may be processed hours after the actual hour it contains, since again we need the next event to decide).

One possible solution mentioned in [1] is to let the reporting system insert an extra event each 60 minutes. With annotations another solution is given. If the system can find out that there might be more than 60 minutes between the events, it can warn when such queries are asked or activate the injection of extra events itself.

Another information we might gain from additional information is that each financial transaction will end for sure, either with a commit or an abort. So we can maintain a sliding window for each transaction and be sure that it will be closed after some time. Maybe the annotations also tell us how long such windows will be open in average. This information can be used for decisions as how many threads in parallel will be used (does it make sense to keep this many threads or better to let some windows wait? - watch out for real time applications), how much memory will be needed and so on.

3.3.4 Toolbox

Here, many optimisations and rewritings are possible, depending on the query and its information read from the items. The optimisations are the most common, like knowing that windows have a guaranteed end if time is increasing or certain events are listed in the annotation as to be guaranteed to occur in the future (see Section 3.3.7).

On the other side, most of the use cases listed in this category just have to go over the stream once and keep some calculation up to date, like averages or counts for defined events, or filter the stream. So this category is one of the core applications on streams, where it must be possible to continuously produce output while still feeding input, since both, in- and output may be infinite.

3.3.5 Positional grouping

Again we can optimise queries to use tumbling windows, i.e. in the Headings and Paragraphs use case we know that the H2s will be returning and since we use it as start and end with no other H2 in-between, that is exactly what a tumbling window is about. Same fits for the Term Definition Lists. Note, that this optimisation does not solve the problem of blocking with infinite streams, since it may take infinite time until the next H2 arrives.

The possibility to use tumbling windows also tells something about memory, since we only have to keep/materialise events for the current window. After its production, these items can be forgotten.

A second optimisation may be possible for the Page Ranges: If the information say that the list of numbers is ordered, it does not have to remember all or to wait until the end of input to output first ranges, since with the first gap between two input numbers it has the end of the first range. So again, in this special case, the system can use a tumbling window with start condition fn: true and end condition a gap between input page numbers.
3.3.6 Miscellaneous

The night club use case is just another entry gate with additional events about the consummations. So again, all events between two payments can be collected by sliding windows or, if the stream is prefiltered into separate streams for each card id, even by tumbling windows.

Note, that this difference is very important concerning memory. Since the stream contains not only start and end events but consummations as well, memorisation might become a problem. This gets in turn solved by the knowledge that each window has an end for sure, therefore the windows can be produced early and in parallel, so no memorisation for the events but only the results will be needed.

3.3.7 Other

As we have seen, in many use cases the same things can be detected or optimised. Another helping information might be that time is increasing or generally ordering information. For the time option, we refer to the last subsection of [1, Section 3.3.2]. There they say, that they recommend to use sliding windows as a hint for the query processor to reduce the number of open windows, since the query processor is not able to know that time is an increasing value. But exactly this information could be given by the schema.

The same fits for FORCE [1, section 3.4.1], to re-enable again the early production of events if we can tell the processor that time only increases and therefore there will come an ending event. With the knowledge of the additional information on the stream, we should be able to enable pipelining and early productions again, where they got blocked by some possibly infinite end conditions of windows or other constraints.

In most of the above listed use cases, we only presented the possibility to rewrite queries. Another information we might get from the annotations are memory needs that we get from the elements if we want to materialise them. Knowing how many children an element has and also (from the query itself or some output description) which of them will be used by the query it is working on, the engine may decide to drop some of the children that will never be used to save memory (let them be garbage collected out of the middle of the buffer).

The last thing to keep in mind is that annotations can be used for any data. Therefore, also for the output of the query. With this information we can already abort, if we know that the user wants an answer in \( x \) seconds and we have never enough computational power or memory to reach this goal or input might take longer to arrive, or we can decide to start a thread for every window in parallel if it should be fast and we have low load.

3.4 Requirements for stream annotations

Regarding the different use case examples, we see that there are multiple kinds of annotations we could extend stream descriptions to.

- **Known (possibly repeating) patterns** of items inside a stream. This meets for example the entry gates, where we know that for every \text{in} event of a person P, there will come an \text{out} of P before the whole sequence repeats itself.

- **Information about values** of events, such as the information that timestamps do strictly increase or that a certain attribute only has enumerated values.

Keep in mind that all these extensions can also be used to give information about returned elements, as which events should be contained, what output frequency is required and so on, as far as they can not be extracted from the query itself.
3.4. REQUIREMENTS FOR STREAM ANNOTATIONS

In the following, we use the term *sequence* as the descriptor for the repeating part of a stream.

3.4.1 Patterns

To describe patterns several information are needed. We deduce them in four steps:

1. The simplest way of describing the elements of a stream is to list what elements will occur. So a stream can be first described by giving a set of element types to expect.

2. To increase information content the above set of types can be transformed into an ordered list, telling the occurring types in the order of their appearance. Like this one gets the ability to decide if the arriving stream is the described one by comparing to the pattern information. The information should also report the possibility of the individual elements to not appear in every instance of the sequence.

3. The before mentioned ordered list is not able to describe all streams:
   (a) Since streams deliver single parts of complex events there will be the probability to get subevents of several complex events in parallel. Therefore, there must be the possibility to describe multiple patterns that appear interchangeably on the stream.
   (b) Streams can also be composite of other streams, like the output of a feed aggregator. Again this establishes the need to describe multiple patterns which will appear loosely alternating.

4. Since streams are possibly infinite we need some means of telling how these elements will repeat themselves. Which patterns may repeat? Do the individual patterns alternate strictly? Is there a number of repetitions for each single pattern? Or just one total number of repetitions? From the information of this area it should be possible to deduce if the stream in total is infinite or not.

3.4.2 Information about values of events

Having defined the patterns, there may be the need to describe how they will develop, meaning that if they repeat the may describe the same thing again, with some changes. In addition there is the need to have some information about the different values in general, to be able to do some optimisations or calculations about some statistics. As example look again at timestamps of events. The system itself will not know what they mean, they are just numbers, integer fields of the elements. Let's take then a windowed query that has as start and end conditions some comparison of this integer field. It will only be closed if the value of the current element is strictly greater than the first. The engine itself is not able to predict if this will ever be the case without the information that the values will be increasing with every iteration.

There are again different stages of information one can give about values:

1. Information about orderedness of specific data types. This fits the above mentioned integer field defined explicit as of type “timestamp” with the only change to the integer type being the information that each time an element has such a field, its value will be greater than the value of the last seen field with this type.

2. Ordering of specific fields of certain elements in the stream. Instead of defining the orderedness in the type, one can add the information to the field of a certain element in the sequence, telling that it is not the instances of the type itself that will be ordered but the values of a certain field in a specified element of the sequence. Like that the information e.g. only concerns the field “sequence-nr” of reported http events.
In addition, one could not only want to inform about ordering of value instances but even about instance dependencies in general. So the mechanism to inform about ordering must be made more powerful to allow all sorts of dependencies between two instances of a certain field:

- orderedness (in- and decreasing)
- partial orderedness (e.g. each time the third digit from the right changes its only growing, but the first two digits are not sorted)
- other dependencies using only current and next element value (e.g. if the current value is even, the next will be odd)

The last requirement we define on item values concerns uniqueness: In the case of parallel sequences one must be able to decide to which instance an item belongs to. Therefore, the stream description must have a possibility to declare sequence item values as unique. This can be done e.g. by a key defining some fields all items must have and of which the content must be the same for all items of the same sequence.
Chapter 4

Solution proposal for stream descriptions

Having stated the requirements, we give our proposal of how to fulfil them, listing for both categories the information we want to give about streams. For every property we will give a visual example and in case of patterns the proposed notation using regular expressions. Whenever we use regular expressions we will define them using a list notation similar to the ones used in functional programming languages:

Single letters are single elements, multiple letters name lists:

- \((a)\) is the list containing the element \(a\).
- \((as)\) is a list of elements not defined more precisely at the moment.
- \((\)\) is the empty list.

List equality is shown by ‘==’:

‘==’ denotes equality of lists.

A list consisting only of another list is the same as this inner list itself:

\(((as)) == (as)\)

Single elements denote lists consisting of only that single element:

\(a == (a)\)

Output will be given printing every element in single quotes:

- ‘a’ is the output from a stream consisting only of the element \(a\).
- ‘,’ is an append operator, building lists:
  
  \((a,as)\) is a list of which we know that it starts with an ‘a’.
  \((as,bs)\) is a list consisting of two other lists.

‘|’ denotes possibilities. One or the other is to occur, they are not ordered:

\((a)\&(b) => \ 'a', 'b' | 'b', 'a'\)  - Reading a stream combination of single element streams will result in either reading first the element of the first or the second stream. Both possibilities have the same probability. (Stream combination will be defined later on and is only used as an example here.)
4.1 Patterns

A stream is described by the items that will appear on it. This is done by a sequence, which at the same time defines the order in which the items are expected to arrive. For every item one can set if it has to appear in every iteration or not.

There are two problems with that method: First, since a stream is sometimes infinite, one cannot give all the items of the stream. On the other hand a stream could also be allowed to begin differently each time it starts or to have not the same ordering among its elements every time. Therefore, we look for patterns, subsequences from which one can build up all possible variations of the stream.

These patterns can then be used to describe the stream by combining them into groups, ordering by surrounding sequences and building choices to declare that at some points the stream behaviour can be one of some given variants.

Definitions:

`=>' is used as derivation operator showing output when reading the stream defined by the list:

\[(a, as) => 'a', (as) - \text{"when reading the stream } (a, as) \text{ you will first receive the first element of the list, followed by the same output as if when reading the stream } (as) \text{. Therefore } '=>' \text{ is used to define stream output recursively."} \]

When lists are output, inner lists are output recursively, meaning that if the next element of a list is itself a list, it’s first element will be output and not the whole list:

\[(a, (b, c)) => 'a', ((b, c)) = 'a', (b, c) => 'a', 'b', 'c' \]

Single items are described giving their name followed by parenthesis containing the names of the fields they have:

\[a(name, gender) : a \text{ is an item informing about a person, containing the name and gender of this person.} \]

Items between parenthesis are to appear exactly in the ordering they have inside the parenthesis:

\[
\begin{align*}
() & => \\
(a) & => 'a' \\
(a, as) & => 'a', (as)
\end{align*}
\]

`|` can also be used as pattern description to denote choices. Either the left or the right side will appear:

\[
\begin{align*}
(a|b) & => (a) | (b) \\
(as|bs) & => (as) | (bs)
\end{align*}
\]

`*` denotes occurrence between 0 and n times. It can be used directly on elements as well as on lists; using it on single elements assumes a list consisting of a single element:

\[
\begin{align*}
(a)* & => (a), (a)* | () \\
(a, as)* & => (a, as), (a, as)* | ()
\end{align*}
\]
4.2. ITEM VALUE INFORMATION

'+’ denotes occurrence between 1 and n times. It can be used in the same way as ‘*’:

(a)+ => 'a', (a)*
(a,as)+ => 'a', (as), (a,as)*

’?’ denotes occurrence of 0 or 1 times. It can be used in the same way as ‘*’:

(a)? => 'a' | ()
(a,as)? => 'a', (as) | ()

The basic patterns can be used at all places in the stream description. The extensions defined in the following sections are only applicable at top level, outside all standard patterns.

Two simple streams are shown in Figure 4.1.

4.2 Item value information

To give the engine or user some information about values to be expected, we have to extend stream information by some means of value constraints. Other information, such as restricted values (enumerations), are already integrated in most type systems and need no additional extension.

Item value information are only allowed for outermost sequences.

4.2.1 Restrictions on values of following items

To give the system some information about dependencies between two items or the first item of a stream, we introduce next statements. These statements can be one of the following:

- simple facelets to restrict following item values (e.g. "next > current", "next = current+1" or "first = 13"). These rules have to evaluate to true for all cases.
- complex rules about dependencies between following elements in general, such as "current = 'a' ⇒ next ∈ ['a', 'b', 'z']". The semantics of these dependencies are as follows: If the first part (before ⇒) evaluates to true, the second part (after ⇒) will also be true.

A next statement consists of two parts: The list of selector conditions and the list of next rules.

The selector conditions define which elements are constrained by the next statement. In the following examples we use field names to state that the elements to be constrained must have fields addressable by these names, but in this list can be used other conditions like e.g. for XML one could use XPath node tests (e.g. self::elementname to tell that only elements having as name 'elementname' are constrained). The selector conditions must at least test for all fields used in the next rules.

\[ \text{stream instances: } A()\ldots B()\ldots C()\ldots A()\ldots D() \]

\[ \text{schema: } \]

\[ \text{regexp: } (A, B^*, C, A+, D) \]
The next rules consist of the new single keyword first or the two new keywords current and next and the dependency between them. The keyword next stands for the item arriving following up to the one that has just arrived and fulfilling the selector conditions. The element which just arrived is represented by current. first denotes the item to expect at start of the stream.

Since it might be that not all elements on the stream have the same fields the constraint only applies to the elements which do fulfill the selector conditions given in the next statement. Therefore, one can neither say that next always means really the next appearing element nor that it means the next element of the same type. It refers to the next element for which the selector statements evaluate to true.

If someone needs to describe not just the next item but the next item of the same type or the same item in the next iteration, one can define the according next rule more precisely using selectors. If one wants to describe the next item of the same type one writes the statement only to fit on this exact type.

Both types of rules can access fields of the items bound by next and current by appending '.fieldname'. For example, next.time would access the time field of the item bound to next. If no fieldname is appended, the value of the item itself is used.

**Definitions:**

next statements are given in brackets, using the pseudofunction `next(selector conditions, rules)` with two arguments, a list with selector conditions the elements must fulfill to be constrained by the next rules and a list of rules. All fields used in the rules must appear as selector condition.

The selector conditions can be any conditions applicable in the surrounding processing model.

The rules can be any rule applicable for the types of the used fields. They contain either 'next' in combination with 'current' or just 'first'.

- Simple comparisons are defined using '<', '<=', '==', '>=', '!=' and '>'; |

  (as)[next((time),(next.time>current.time))] describes a stream of elements (described inside the list named "as") of which all elements having the field "time" are ordered according to the value of the "time" field. The value of the time field is only getting greater.

- Dependencies are defined using '=>', set membership using '∈', sets are denoted using '{' and '}'

  (as)[next((state),(current.state∈{'s1','s2'} => next.state∈{'s3','s4'}))] describes a stream of elements (described inside the list named "as") of which all elements having the field "state" follow the following dependency: If the state value of the current element is either 's1' or 's2', the state value of the next element will be 's3' or 's4'. If the current state value is neither 's1' nor 's2', the next element is not constrained by this rule.

Figure 4.2 shows an example for facelets on each next item, two examples for facelets for items of the same type are shown in Figures 4.3 and 4.4.

### 4.2.2 Information about parallel sequence values

To fulfill our requirements we need one more information, answering more than one question:
4.2. ITEM VALUE INFORMATION

- How many different values will occur (but not which values, therefore no enumeration)? This requirement is needed e.g. for the entry gates use case that the engine may know the maximum number of people in the building at the same time: the maximum number of in’s without an out and therefore the maximum number of open windows.

- How to decide to which instance of an interleaved sequence an element belongs to?

This information is given as a key constraint, which consists of two part. The first part is a list denoting the fields which form the key and is required. The second is the variety, denoting the size of the key-value-space or in other words the number of possible values in the fields of the key. The variety is optional. If not given, it is assumed to be "infinite". The key construct can be given for any sequence, denoting that all its elements contain the listed fields and for one instance of this sequence, all items will have the same value in these fields. A key is required if a sequence is interleaving.

Definitions:

key constraints are given in brackets, using the pseudofunction key with either one or two arguments: key(fields) or key(fields,variety).

- fields is a list of fields all the elements of the sequence must have and which will have the same content per sequence.
• *variety* is an integer denoting the number of possible value combinations (the size of the key-value-space).

If not given, variety is assumed to be infinite.

\[(as)[\text{key}((\text{name}))]\] is a stream of elements (listed in the list as), constrained by a key on field "name" without given variety.

\[(as)[\text{key}((\text{name}),3)]\] is a stream of elements (listed in the list as), constrained by a key on field "name" with variety 3.

The key property is shown in Figure 4.5.

\[
\text{stream: } \text{in(A)...out(A)}
\]

\[
\text{schema:}
\]

\[
\begin{array}{c}
\text{in(name)} \\
\text{out(name)} \\
\text{key:} \\
\text{fields: (name)} \\
\text{variety: 3}
\end{array}
\]

\[
\text{regexp: } (\text{in(name),out(name))}[\text{key((name),3)]}
\]

Figure 4.5: Describing a sequence with a key constraint

### 4.3 Sequence repetition

This is the core functionality allowing us to describe infinite streams. Instead of giving the same sequence again and again in the schema, one can state that it is repeating. In notation of regular expressions we denote repeating with "\*". Note, that although the "\*" is used for inner repetitions as well as for the outer ones, only the top-level sequences can be infinite. The definition for the regular expression notation will follow in the next section. Figure 4.6 shows an example stream with a repeating sequence.

\[
\text{stream: } \text{in()...out()...in()...out()...in()...out()...}
\]

\[
\text{schema:}
\]

\[
\begin{array}{c}
\text{in()} \\
\text{out()} \\
\text{repeating}
\end{array}
\]

\[
\text{regexp: } (\text{in,out})*
\]

Figure 4.6: Describing a repeating sequence

#### 4.3.1 Number of iterations

This information is depending on the information from Section 4.3 about sequence repetitions. If it repeats, this information will tell the number of *iterations* or whether the number is unbounded. A last possibility will be telling that the sequence repeats infinitely. This information will be represented by one single value. If it is 0, the sequence does not repeat. If it is >1 it iterates the given number of times, "unbounded" means it repeats an unknown but finite number of times and "infinite" means it repeats forever. 1 is a special case since one iteration is the same as no repetition and is therefore prohibited. For the regular expressions we add to the "\*" nothing for "unbounded"
and a second "*" for "infinite". If we have a limited number of iterations we replace the "*" by "\{x\}", 
x being the number of iterations.

Infinite repetitions are only allowed for outermost sequences.

\( \mathbb{N} \) is the set of natural numbers, including 0.
\[
\begin{align*}
\text{(as)}\{0\} & \Rightarrow (\text{as}) \\
\text{(as)}\{x\} & \Rightarrow (\text{as}), (\text{as})\{x-1\} \text{ for } x \in \{(\mathbb{N}\setminus\{0\}) \cup \{\infty\}\}
\end{align*}
\]

'*' has been described before, we will give here a second definition
using the \{x\} notation but with exactly the same meaning:
\[
\begin{align*}
(\text{as})* & \Rightarrow (\text{as})\{y\} \text{ with } y \in \mathbb{N}, y \text{ is only known at runtime.} \\
(\text{as})** & \Rightarrow (\text{as})\{\infty\}
\end{align*}
\]
Figure 4.7 shows all four (the stream for * and ** looks the same in the beginning) variants.

4.4 Stream combination

To aggregate a stream from different other streams, like it is done by RSS-aggregators, we introduce 
the property of combination. It allows to state that a stream consists of multiple sources. Every 
source can be described with all possibilities and then, as last step, they can be combined. For 
the regular expression extension we will use ' & ' while for the visual schema we put the individual 
sequences beside each other inside another stream box.

The combined streams have to consist of elements of disjoint sets. Otherwise no one could 
distinguish which stream an element belongs to, Being disjoint is defined here to be named differently, 
to have different fields (different schema/description) or to be distinguishable by a key to be given 
in addition to the combination. The key is the one defined in Section 4.2.2.

Stream combination is only allowed to be applied to outermost sequences, but there it can even 
be applied on infinite and interleaved sequences. Since stream combination describes only that a 
source will deliver items from different streams, it is not allowed to use the other stream descriptions 
on top of it. Of course, when one of the inner streams is infinite, the combined stream will be infinite 
as well (How infinity on streams is defined is explained in Section 4.7). But the combined stream 
will never be interleaved or repeating, since it is only a bundling of streams and therefore can not 
tell the inner streams to e.g. repeat themselves.

Definitions:
\[
\begin{align*}
(a) & \& (b) \Rightarrow 'a' , (b) | 'b' , (a) \Rightarrow 'a' , 'b' | 'b' , 'a' \\
(a,as) & \& (b) \Rightarrow 'a' , (as) & (b) | 'b' , (a,as) \\
(a,as) & \& (b,bs) \Rightarrow 'a' , (as) & (b,bs) | 'b' , (a,as) & (bs)
\end{align*}
\]

'&' has higher priority than '|' and therefore one does not need to wrap it into parenthesis.

'&' is left ordered:
\[(as) & (bs) & (cs) == ((as) & (bs)) & (cs)\]

'&' is distributive over '|', commutative and associative:

\[(a, b | b, a) & (cs) == (a, b) & (cs) | (b, a) & (cs)\]
\[(as) & (bs) == (bs) & (as)\]
\[((as) & (bs)) & (cs) == (as) & ((bs) & (cs))\]

The last definitions about associativity, commutativity and distributivity result from the fact that ' &' combines (possibly external) streams by bundling their output. The stream just delivers what arrives on one of the separate streams. It cannot influence the ordering and therefore it is not important which streams are "bundled first."

**Proofs:**

distributivity over '|'

\[(a, b | b, a) & (cs)\]

\[=>\]
\[\text{case } a, b:\ (a, b) & (c, cs) \Rightarrow 'a', (b) & (c, cs) | 'c', (a, b) & (cs)\]
\[\text{case } b, a:\ (b, a) & (c, cs) \Rightarrow 'b', (a) & (c, cs) | 'c', (b, a) & (cs)\]

\[(a, b) & (c, cs) | (b, a) & (c, cs)\]

\[=>\]
\[\text{case } (a, b) & (c, cs) : (a, b) & (c, cs) \Rightarrow 'a', (b) & (c, cs) | 'c', (a, b) & (cs)\]
\[\text{case } (b, a) & (c, cs) : (b, a) & (c, cs) \Rightarrow 'b', (a) & (c, cs) | 'c', (b, a) & (cs)\]

commutativity

\[(a) & (b) => 'a', 'b' | 'b', 'a'\]
\[(b) & (a) => 'b', 'a' | 'a', 'b'\]
\[(a, as) & (b, bs) => 'a', (as) & (b, bs) | 'b', (a, as) & (bs)\]
\[(b, bs) & (a, as) => 'b', (b, bs) & (as) | 'a', (as) & (b, bs)\]

associativity

\[((a) & (b)) & (c)\]

\[=> (a, b | b, a) & (c) = (a, b) & (c) | (b, a) & (c)\]

\[=>\]
\[\text{case } (a, b) & (c) : (a, b) & (c) \Rightarrow 'a', (b, c) | 'c', (a, b)\]
\[\text{case } (b, a) & (c) : (b, a) & (c) \Rightarrow 'b', (a, c) | 'c', (b, a)\]

\[\text{compared cases: } ((a) & (b)) & (c) = 'a', 'b', 'c' | 'b', 'a', 'c' | 'b', 'c', 'a'\]

\[(a) & ((b) & (c))\]

\[=> (a) & (b, c | c, b) = (a) & (b, c) | (a) & (c, b)\]

\[=>\]
\[\text{case } (a) & (b, c) : (a) & (b, c) \Rightarrow 'a', (b, c) | 'b', (a) & (c)\]
\[\text{case } (a) & (c, b) : (a) & (c, b) \Rightarrow 'a', (c, b) | 'c', (a) & (b)\]

\[\text{compared cases: } (a) & ((b) & (c)) = 'a', 'b', 'c' | 'b', 'a', 'c' | 'b', 'c', 'a'\]

Stream combination is shown in Figure 4.8, where we show first the two streams independently and then combined.
4.5 Interleaving sequences

So far we can describe streams with possibly repeating sequences. For example \((a,b,c)^*\) describes the stream \(a,b,c,a,b,c,a,b,c,...\) and \("a,a,a,c,b,c"\) can be a possible instance of either \((a,a,a,c,b,c), (a^*,c,b,c), (a{3},c,b,c)\) or \((a^*,(b^*,c)^*)\). There is one thing missing, namely how to allow the streams to not look the same all the times but still follow some requirements. This is needed e.g. to describe entry gates, where it varies how many people will enter a building before someone else leaves it. So there should be the same description for "in,out,in,out,in,out,in,out" and "in,out,in,in,out,out,in,out". Figure 4.9 depicts this behaviour.

A wrong way would be to describe the above with \((in^*,out^*)^+\). This is wrong because the number of out’s occurred at a certain point has to be at most the number of in’s, since no one can leave a building before having entered. To reach this, we allow a sequence to be interleaved, which means that it can appear multiple times in parallel, its items being intermixed. Therefore, one can describe the two entry gate streams above by \((in,out)^*~^\sim, \sim\) denoting interleaving. Since again multiple sequences, in this case multiple occurrences of the same sequence, are combined to one stream, interleaving is simply a special case of stream combination (see Section 4.4).

Still one part is missing: How to decide to which sequence an element belongs to, making it unique. For this we use the key defined in Section 4.2.2. The key-fields defined there are required for an interleaving sequence.

Interleaving can only be applied to outermost sequences, but there even on infinite ones.

Definitions:

\(\mathbb{N}\) is the set of natural numbers, including 0.

Indices denote values of item fields:

\[a_{k=1} : \text{field } 'k' \text{ of item } "a" \text{ has value } "1"\]
\[a_{s_{k=1}} : \text{field } 'k' \text{ of every item in list } "as" \text{ has value } "1"\]
'~' shows that a stream is interleaved:

\[
(a)^-\text{key((k)),x)}] \Rightarrow (a_{k=1}\&(a_{k-2})\&\ldots \&(a_{k-x})\text{key((k),x)}] \text{ for } x \in \mathbb{N}
\]

\[
(a)^-\text{key((k))]} \Rightarrow (a_{k=1}\&(a_{k-2})\&(a_{k-3})\&\ldots \&\text{key((k))})
\]

\[
(a,as)^-\text{key((k)),x)}] \Rightarrow (a_{k=1},as_{k=1}\&(a_{k-2},as_{k-2})\&\ldots \&(a_{k-x},as_{k-x})\text{key((k),x)}] \text{ for } x \in \mathbb{N}
\]

\[
(a,as)^-\text{key((k))]} \Rightarrow (a_{k=1},as_{k=1}\&(a_{k-2},as_{k-2})\&(a_{k-3},as_{k-3})\&\ldots \&\text{key((k))})
\]

Figure 4.10 shows the usage of the added information including the key.

\[
\text{stream: in(A)...in(B)...out(B)...out(A)} \quad \text{schema:}
\]

\[
\text{regexp: } (\text{in(name),out(out(nome)))}^-\text{key((nome),2)])
\]

\[
\text{Figure 4.10: Examples for a non-repeating and a repeating interleaved sequence}
\]

4.6 Examples

- Headings and Paragraphs: The sequence consists of \(P\) and \(H2\) events, is repeating a bounded number of times and not interleaved. \(P\) can occur zero or multiple times every iteration: \((H2,P*)\)*. Therefore, it is finite but it is not known if the last element of the stream is a \(P\) or an \(H2\). If we are using window queries the last conclusion tells us that if we start and end the windows by detecting \(H2\), there may be open windows at the end of the stream: Figure 4.11

\[
\text{stream: H2()...P()...P()...H2()...P()...H2()...}
\]

\[
\text{schema: }
\]

\[
\text{regexp: } (H2,P*)\*
\]

\[
\text{Figure 4.11: An example stream for the headings and paragraphs use case}
\]

- Entry gate: The sequence consists of an \(in\) and an \(out\) event. It is repeating infinitely and there is such a sequence for every person. The latter means, for person \(P\), after each \(in\) will be an \(out\) before the next \(in\), but other person's events may occur between the ones of \(P\). Or in our words, the sequence is marked as interleaved, having a key on the field "name": Figure 4.12

4.7 Sequence infinity

The information, if a sequence will have an end at any point in time, is directly derived from the information about repetition, interleaving and the variety of the interleaving key, if any of them are given. Since all description methods except the stream combination(Section 4.4) are defined on
4.8. SUMMARY

4.8. SUMMARY

stream: \texttt{in(name)...out(name)...in(name)...out(name)...}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
stream pattern qualiers & repeating & interleaved & key - variety & resulting overall stream \\
\hline
\text{infinite} & - & - & \text{infinite} & \\
\text{-} & \text{true} & \text{inf (default)} & \text{infinite} & \\
\text{some number} / \text{unbounded} & \text{false (default)} & - & \text{finite} & \\
\text{some number} / \text{unbounded} & \text{true} & \text{given} & \text{finite} & \\
\text{0 (default)} & \text{false (default)} & - & \text{finite} & \\
\hline
\end{tabular}
\caption{Stream infinity combinations}
\end{table}

Figure 4.12: An example stream for the entry gate use case

single streams, this section describes how to derive the infinity state of a single stream. The different combinations of these three properties and their influence on the infinity of a stream are listed in table 4.1, assuming as default values "0" for repeating and "false" for interleaved. The symbol "." means this value is not important, the values of the other fields imply the infinity state directly.

4.8 Summary

In this section we will shortly repeat all the extensions to summarise how a stream can be described.

The way stream descriptions are given is as follows:

1. One defines the sequence(s) that will be part of the stream using standard regular expressions and numbered iterations, e.g. \( (A,B?,C)^* (D,E,F)^* \)

2. Infinite sequences are declared, e.g. \( (A,B?,C)^* (D,E,F)^* \)**

Figure 4.13: The problem of infinity without repeating
3. Interleaved sequences are flagged, e.g. \((A,B?,C)^* \sim (D,(E+,F)^*)**\)

4. The sequences are combined to a stream, e.g. \((A,B?,C)^* \& (D,(E+,F)^*)**\) or in case of a single sequence the one is output directly as the stream description, e.g. \((A,B?,C)^*\)

The following table (4.2) lists all extensions as a short summary.
<table>
<thead>
<tr>
<th>property</th>
<th>description</th>
<th>regexp</th>
<th>visual</th>
<th>example stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Std. pattern</td>
<td>The patterns known and used so far for sequence descriptions</td>
<td>((A^*,(B?,C)+))</td>
<td>![image]</td>
<td>A, A, A, C, B, C, B, C</td>
</tr>
<tr>
<td>Next</td>
<td>Restrictions on values of following items</td>
<td>((in(time),out(time))^*) [next(((time), (next.time &gt; current.time))]</td>
<td>next: conditions(time), rules (next.time &gt; current.time)</td>
<td>in(5:00), out(5:10), in(6:55), out(11:35)</td>
</tr>
<tr>
<td>Key</td>
<td>Which fields form the key that shows to which instance an item belongs and how many different values this key can have</td>
<td>((in(name),out(name))) [key((name),3)]</td>
<td>key: fields(name), variety: 3</td>
<td>in(A), out(A)</td>
</tr>
<tr>
<td>Repeating</td>
<td>Outermost sequence repeats (possibly infinite)</td>
<td>((A,B)^*)</td>
<td>repeating</td>
<td>A, B, A, B, A, B, A, B..</td>
</tr>
<tr>
<td># iterations</td>
<td>Denotes the number of iterations of the outermost sequence</td>
<td>((A,B)^*) for unbounded, ((A,B)^{x}) for (x) iterations, ((A,B)^**) for infinite repetition</td>
<td>repeating = unb, repeating = x, repeating = inf</td>
<td>((A,B)^2) =&gt; A,B,A,B</td>
</tr>
<tr>
<td>Combination</td>
<td>Combines multiple streams into one. The combined sequences have to be disjoint. Disjointness is given by different item sets (their schema/description differs, by name or by their fields) or by giving a key consisting at least of the fields list.</td>
<td>((A,B)&amp;(C,D))</td>
<td>![image]</td>
<td>A, C, D, B</td>
</tr>
<tr>
<td>Interleaving</td>
<td>There can be multiple instances of this sequence at the same time, its items arriving intermixed. A key containing at least the fields list is required for interleaving sequences. Interleaving is a special case of stream combination, where all combined streams look similar. The sequences are distinguished by the given key.</td>
<td>((in(name),out(name))) [key((name),3)]</td>
<td>interleaved</td>
<td>in(A), in(B), out(B), in(C), out(A), out(C)</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of the stream extensions
Chapter 5

An extension for XML Schema

Having seen the need for stream descriptions and stated the requirements as well as a solution proposal for the general case, we decided to use XML Schema as stream description language and therefore extended it with the required information.

5.1 Why an XML Schema extension?

So far we have seen the basic concepts for XQueryP and streaming. The problems in combining the two is obvious: If an XQuery has to read the first three children of a node and the second of them contains an infinite stream, the query will never be able to produce an answer, because it is not able to read the last element.

Krasko wrote in [1, Section 4] an extension to the XDM to allow an instance to be infinite. Later on he describes a way to mark infinite elements in the XDM, so that one is able to detect them at runtime. Another method to deal with special queries on possible infinite streams described in [1, Section 3] is the FORSEQ keyword to iterate over sequences and therefore use windows on an infinite stream.

But there should be a way to let the query engine detect statically either if there will be problems with a query (e.g. will it block because of infinite input?) or if it could make some optimisations (e.g. change landmark to tumbling windows to decrease the number of open windows, if the schema of the stream shows that this is possible and fits the needs of the query).

In this chapter we will develop extensions for XML Schema to enable stream problem solving and optimisations. Note that we develop these extension with the only purpose to enable optimisations on stream queries. Our goal is not to validate any stream with the extended schema, since it is yet undecided how to validate a possibly infinite stream. This results from the fact that a validation will finish if it has read the whole stream, which is still infinite. Since validation could be done per element, it is possible that our extensions can be used in the future to generate some more PSVI contributions. Again, note that our single goal is to enlarge the Schema domain to cover the extensions to the XDM made in [1].

5.2 The XML Schema extension

In this section we will define the extensions to XML Schema. They will extend the concept of model groups\(^1\). Note that we do not use the term sequence to refer to the XML Schema model group xs:sequence\(^2\) explicitly, but a construct very similar to the model groups. Since it allows the

\(^1\)Model groups are defined as part of [6] in http://www.w3.org/TR/xmlschema11-1/#Model_Groups

\(^2\)The following model groups exist: xs:sequence, xs:all, xs:choice. For details, see [6]
most restrictions on a stream, even sequential ordering, \texttt{xseq} will look most similar to the new construct. In some cases the use of a repeated conjunction or disjunction may be enough. But not to guarantee an end of a sequence: Only with an \texttt{xseq} combined with \texttt{minOccurs=1} you can oblige e.g. an \texttt{<out>} to appear at the end.

Even more restricting, this sequence can only be placed in one point: Since the extension of [1] allows only an XDM instance to be an infinite sequence, the only place to bring infinity into XML Schema is at the top-level, in other words the \textit{root node}. We will vary this root a little bit by saying that not only a root node may be at top level of an XML Schema, but a model group as well. And in exactly this \textit{root sequence} we place our extensions. Therefore, all of the extensions must be made to the root sequence and only the root sequence.

In the following we will give a short explanation about our modifications of XML Schema. Each time we describe it in terms of changes to the \textit{SchemaForSchemas}\textsuperscript{3} - one of the normative documents of the W3C XML Schema definition - as well as in terms of the textual form used in the W3C documents. The complete extension document can be found in Section 5.2.6.

### 5.2.1 From root node to root model group

First, we define that the XML document can begin with a model group instead of just elements.

To enable this, we extend the Schema Component by the property \texttt{stream model group}, which simply contains a stream model group as defined below, if there is one given as direct child of the schema element of the schema document.

In the \textit{SchemaForSchemas} we extend the internal complexType of the \texttt{schema} element to contain our stream model groups (see below). Since the extensions to the model groups should only be usable on top level, we refine the existing model groups and extend the schema type to contain a new group \texttt{streamModelGroup}. (see Section 5.2.2)

It is not possible to extend just the group \texttt{schemaTop}, which is part of the above mentioned type, since this group appears in a \texttt{sequence} of unbounded \texttt{maxOccurs}. But to describe a stream clearly, we must not allow more than one of our root model groups to appear. Therefore, we add our new group on the same level as the \texttt{schemaTop} group and give it an appearance of 0 or 1 times. As you will see, our definition of stream model groups still keeps the possibility to describe a stream as merge of different streams.

#### 5.2.1.1 Root node or Root sequence?

Assuming a schema that declares a streamSequence by referencing two globally defined elements, the engine can not decide in advance if the first incoming element is "the" root element of a document or the first element of a stream. We explicitly do not prohibit one from declaring global elements, since this is a core functionality of XML Schema. We also explicitly allow a schema document to define a document as well as a stream, since like this one can define the elements to store and the ones to be transmitted by a stream with one single schema.

This of course raises one problem: A query engine can not decide in advance if the input will be a stream or an element. The query engine can only decide on this problem in the following two cases:

- After the first "root" element arrives a second one → It is a stream. Otherwise, there is an "end of file" after the first element.
- The query allows derivation of this fact. E.g. if the user uses a \texttt{FORSEQ} on the document variable, it is a stream, since otherwise there would be only one element at this level and as

\textsuperscript{3}The \textit{SchemaForSchemas} can be found in [6] in http://www.w3.org/TR/xmlschema11-1/#normativeschemaSchema and in [7] in http://www.w3.org/TR/xmlschema11-2/#schema
such only one sequence of one element output by the FORSEQ. Since the user knows which
documents he works on, he would not ask such a query.

Since this question is only decidable later on, the engine may evaluate both scenarios and drop the
wrong one as soon as it gets clear. It is not possible to only assume it will be a stream, since with
this assumption the engine will use the additional information and therefore the semantics might
change, as we describe in Section 6.2.

5.2.2 Stream model groups

We define a new Stream Model Group Schema Component. It has one more property in addition to
the ones from the Model Group Component, namely streamInfo.

The property streamInfo contains a streamInfo Schema Component.

The property compositor is changed to contain one of "streamSequence" or "streamChoice". There is no streamAll, since the model group all is just a restriction of sequence with minOccurs = maxOccurs = 1 and its particles can appear in any order. Therefore, it is inappropriate to describe patterns of streams.

Into the SchemaForSchemas we insert a new streamModelGroup element in the top level, consisting of a choice of "streamChoice" and "streamSequence", both of respectively named types: streamChoiceType and streamSequenceType.

streamChoice is allowed to contain streamSequences and all other particles a model group can contain. It is the XML Schema implementation of the stream combinator '& ' (see Section 4.4). It contains the streams to be combined as streamSequence children. In case of single element streams to be part of the combination, streamChoice can contain simple element definitions as well as the original model groups. These definitions are to be handled like usual sequences (potentially consisting of one element), but note that they do not allow any stream annotations (they do not have a streamInfo child). Although the name streamChoice seems to mark a similarity to the known xs:choice model group, it has nothing more in common but the name. Its purpose is stream combination as it is described in Section 4.4. Note that for validation the engine must keep track of the individual states of the combined sequences, since streamChoice only tells that the next arriving item will belong to one of the combined sequences but gives no ordering. Therefore, the validation must be done for all combined sequences separately. The only case when the validation of the streamChoice itself must fail, is if an arriving item does not fit into any of the inner sequences.

streamSequence is the XML Schema implementation of the outermost patterns the streams
contain. They contain normal model groups as inner patterns, as explained in Section 4.1. streamSequence contains the common model group particles and the new stream description element streamInfo. This new element and its Schema Component are defined in Section 5.2.3.

5.2.2.1 Nesting of stream model groups

There is one pitfall to be noticed using stream model groups, which is nesting. Since XDM was
only extended to allow the top element of an instance to be infinite, we allow stream model groups
only at top level of the schema.

StreamSequence in streamSequence is not allowed since the inner must not iterate infinitely.
StreamChoice in StreamChoice is not allowed since stream combination is associative and can
be modelled by unnesting and putting the elements of the inner choice directly into the outer one.

streamSequence in StreamChoice is allowed, but with the meaning of stream combination, not a real choice.

Figure 5.1 shows examples of model groups and gives an example instance for each.
5.2.3 StreamInfo

The streamInfo Component contains all stream-specific information. It has the properties repeating, interleaved, streamNext and streamConstraint.

The property repeating says if the model group repeats by giving the number of repetitions, if any. If there is no repetition, meaning the elements occur once, the value of repeating is '0'. The value '1' has the same meaning as '0', since '1' denotes one occurrence (and therefore no repetition). If this number is dynamic, but finite, the value of repeating is "unbounded". If there are infinite repetitions, the value is "infinite". The default value is '0' for no repetition. The property being absent also means no repetition.

The property interleaved says if the sequence is interleaved with multiple instances of itself. This is possible independently from the status of repetition. If interleaved is true the streamInfo must contain at least one streamConstraint with category key. The default for interleaved is false.

The property streamNext lists conditions on following elements. It contains a streamNext Schema Component which is explained in Section 5.2.5.

The property streamConstraint lists conditions on the whole stream. It contains a streamConstraint Schema Component which is explained in Section 5.2.4.

In the SchemaForSchemas, streamSequenceType allows the according stream model group element to contain the new element streamInfo, that contains the new elements streamNext and streamConstraint as well as the attributes repeating and interleaved.

An example stream schema

Figure 5.2 shows a stream schema for a simple entry gate without constraints (Note that this schema is illegal since if interleaving is set to true there has to be a streamConstraint of type key. We omit it here to only demonstrate the properties of streamInfo itself).
5.2. THE XML SCHEMA EXTENSION

Figure 5.2: Example stream schema with streamSequence and streamInfo (only for demonstration of streamInfo. This schema would be illegal due to lack of a key.

5.2.4 Stream-constraint Definitions

To fulfill the requirements about value information on parallel sequences (3.4.2), we have to define how to declare a sequence as unique, designating each element to exactly one instance of the sequence.

Therefore, our solution is an adaptation of the key constraint to stream model groups: We define the new Stream-constraint definition Component, consisting of the properties name, category, fields and variety. The Stream-constraint Component is part of the streamInfo Component.

Note, that although streamConstraint is an optional component, it must exist with the category key, if the enclosing streamInfo component has its interleaved property set to true.

The property name gives a unique name for the constraint.

The property category is one of \{key\}. We explicitly model category as a selection to be able to add further constraints in the future.

The property fields denotes the fields that are affected by the constraint. They are given by means of an XPath[11], starting with ".", relative from any element child of the surrounding streamSequence. The fields listed in a constraint of category key have to appear in all elements of the sequence and all elements of one instance of a constrained sequence have to have the same values in these fields.

The property variety is only present if the category is key and gives the maximum number of different unique value combinations. Therefore, it denotes the size of the key-value-space. The default value is "infinite".

For the SchemaForSchemas we define a new element called streamConstraint with the attributes name, type and variety and one or more field elements. The element streamConstraint can appear from one to multiple times inside a streamInfo element.

The two attributes name and type are required. type contains the category. The other attributes and elements contain the information for the homonymous properties.

\[\text{see [6], Section 3.11}\]
An example schema for a constrained stream

The example in Figure 5.3 again shows a schema for an entry gate. It is now constrained, such that in each sequence of the stream the persons of the in and the out element are the same and there is only one sequence for every person. As a last information it tells us that there are 50 employees.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <annotation>
    <documentation xml:lang="en">
      Schema for a constrained entry gate.
    </documentation>
  </annotation>
  <xs:streamSequence>
    <xs:element name="in" type="inout"/>
    <xs:element name="out" type="inout"/>
    <xs:streamInfo repeating="infinite" interleaved="true">
      <xs:streamConstraint name="seqKey" type="key" variety="50">
        <xs:field xpath="./@person"/>
      </xs:streamConstraint>
    </xs:streamInfo>
  </xs:streamSequence>
  <xs:complexType name="inout">
    <xs:attribute name="person" type="xs:string"/>
    <xs:attribute name="time" type="xs:dateTime"/>
  </xs:complexType>
</xs:schema>
```

Figure 5.3: Example schema for a constrained stream

5.2.5 StreamNext Schema Component

To give information on following elements (3.4.2), we extend the `streamInfo` by another component, the `streamNext Schema Component` with its properties `selector` and `next`. The `streamNext Component` is part of the `streamInfo Component`.

The property `selector` is given by an XPath[11] and defines which elements are affected by the constraint; elements not being selected by this selector are ignored by the rules of the current `streamNext` component.

The property `next` contains the rules that restrict an instance according to its nearest predecessor matching the `selector`.

The rules for the `next` property follow a strict syntax. They are either algebraic or set-theoretic formulae containing at least two paths, one using `next`, the other using `current` as start point, as well as a comparator or a dependency. To describe the first item of a stream one can also use only `first` and an algebraic formula which the first item value of the named field has to fulfil.

For the `SchemaForSchemas` we define the new element `streamNext`, having an attribute `selector` to define which elements of the surrounding `streamSequence` are constrained. `streamNext` can appear from one to multiple times inside the `streamInfo element` of a stream model group. Each occurrence must have a unique XPath as value of the `selector` attribute. `selector` is an XPath selecting the elements to be constrained. In case of a specific element, one gives the XPath to this element. In case of various elements having some specific fields, one gives an XPath having a node
test at the end checking for those fields. In both cases the path starts with '.' and therefore is relative to the surrounding streamSequence. Compared to our solution proposal, selector is an XPath having at its end most probably all conditions stated in the conditions part of the next rule.

In case of an XPath selecting just one element per sequence, the rule describes the behaviour of this element in each iteration of the containing sequence. In the case of a node test the rule must be true between every element fulfilling the node test and the next following element again fulfilling the node test.

The rules are given as children of the streamNext element and follow a strict syntax, which is put together by a variety of new elements, defined in a separate schema. In this thesis we will use the namespace prefix "nxt" to denote these special elements. Each rule has to start by an element named dependency or comparator.

Dependencies describe rules like the ones using '=>' in Section 4.2.1. Comparators are used in the other cases, they have an attribute type denoting the type of comparison they stand for. Both elements must have two children on which the dependency or the comparison is evaluated.

Inside comparator one can use value elements to give hard coded values, operator elements to calculate some values, the first element to state which constraints the really first element of the sequence matching the selector XPath has to fulfil, or next and current elements standing for the elements to be constrained. operator again has a type attribute denoting the operation it evaluates and two children as its operands. first, current and next have an attribute xpath to denote which field of the elements matching the selector of the streamNext are used in the rule. Therefore, the selector must make sure the elements it selects have at least all the used fields (the ones appearing in the xpath attribute of next, current and first).

dependency has as first child a special current and as second child a special next element building the subexpressions on which the dependency is defined. As speciality these two elements again have one child, either a set element, describing a set the surrounding element has to fall in to evaluate its part of the dependency to true, a hard coded value or a comparator as before. The semantics of dependency is as follows: If the first subexpression evaluates to true (the surrounding current has the hard coded value, the comparator evaluates to true or the surrounding current falls into the set its child describes), the second subexpression has to be true as well. If the first subexpression evaluates to false (the current element does not have the hard coded value, is not in the set it describes or the comparator evaluates to false), the dependency has no constraining meaning on these exact elements.

For a better understanding of the streamNext construct we give some examples in the following.

Examples for streamNext

The first example says that the time attribute will only increase from element to element (fig. 5.4).

The second example uses set-theoretic rules to constrain the future elements. The element is part of a stream describing the work of a non-deterministic state automaton (fig. 5.5). The rules make sure that the automaton starts with state "start" and in the future eventually the content "goal" is reached, as well as that step increases monotonously.

A third example (fig. 5.6) shows how to use an algebraic formula to tell the engine that input will be partly sorted (sorted with respect to hundreds column upwards).

5.2.6 The extensions of XML Schema 1.1

Below are the complete declarations we added to the W3C XML Schema Part 1 [6]. We give them in written text as well as excerpts from and additions to the SchemaForSchemas. The textual parts are written in the same style as the W3C document. Since there are some similarities to model
groups and identity-constraint definitions, some sentences have been directly copied and/or adapted to assure equal style. If not stated differently, the newly introduced type and element definitions are inserted on top level.

From root node to root model group

Since we only extend the Schema Component, we give only the changes here. All other things are the same as given in the Structure Part of the W3C document.

The Schema Itself

The schema itself has the following additional property:

Schema Component: Schema, a kind of Annotated Component
{stream model group definition} A Stream Model Group Definition component.

XML Representation of Schemas

XML Representation Summary: schema Element Information Item
<schema
  attributeFormDefault = (qualified | unqualified) : unqualified
  ...{any attributes with non-schema namespace . . .}>
Figure 5.5: Next value constraint with set-theoretic rules
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:streamSequence>
    <xs:element name="item" type="itemType"/>
    <xs:streamInfo repeating="infinite" interleaved="false">
      ...<ns:streamNext selector="./item/value">
        <ns:comparator type="greaterequal">
          <ns:first xpath="."/>
          <ns:value>0</ns:value>
        </ns:comparator>
        <ns:comparator type="greaterequal">
          <!-- this dependency means next/100 >= current/100
          or in other words: values are sorted with respect
          to their hundreds column upwards-->
          <ns:operator type="division">
            <ns:next xpath="."/>
            <ns:value>100</ns:value>
          </ns:operator>
          <ns:operator type="division">
            <ns:current xpath="."/>
            <ns:value>100</ns:value>
          </ns:operator>
        </ns:comparator>
      </ns:streamNext>
    </xs:streamInfo>
  </xs:streamSequence>
</xs:schema>

Figure 5.6: Next rules for partially orderedness

Content: ((include | import | redefine | annotation)*,
  (defaultOpenContent, annotation*)?, (streamChoice
  | streamSequence)?, (((simpleType | complexType
  | group | attributeGroup) | element | attribute | notation),
  annotation*)*) </schema>

Schema Schema Component
  Property Representation
  ...
  {stream model group definition} A Stream Model Group Schema Component
  according to the <streamChoice> or
  <streamSequence> element among the
  children, if present, else absent.
  ...

SchemaForSchemas schema

The extended internal complexType of the top level element "schema" is shown in Figure 5.7.
Stream Model Groups

The sequence of element information items in a stream are specified with a stream model group. Unlike model groups, stream model groups cannot indirectly contain other stream model groups. The only exception is streamChoice, which can contain streamSequence to allow a stream to be merged of different streams. But there is no recursion.

Definition: A stream model group directly contains the particles in the value of its particles property.

Definition: A stream model group indirectly contains the particles, groups, wildcards, and element declarations which are contained by the particles it directly contains.

Definition: A stream model group contains the components which it either contains directly or indirectly.

Where subtitles and according subsections are missing, they are equal to the subsections of Model Groups. Just rename choice by streamChoice, sequence by streamSequence and remove the all.

The Stream Model Group Schema Component

The stream model group schema component has the following properties:

Schema Component: Stream Model Group
{compositor} One of streamChoice or streamSequence.
{particles} A list of particles.
{streamInfo} If the compositor is streamSequence: A streamInfo Component. Else absent.
Optional. An annotation.

The stream model group schema component specifies a sequential (streamSequence) or disjunctive (streamChoice) interpretation of the {particles}.

**XML Representation of Stream Model Group Schema Components**

The XML representation for a stream model group schema component is either a `<streamChoice>` or a `<streamSequence>` element information item. The correspondences between the properties of these information items and properties of the component they correspond to are as follows:

**XML Representation Summary: streamChoice and streamSequence**

- **Element Information Item**
  - `<streamChoice`  
    - id = ID  
    - {any attributes with non-schema namespace . . .}>  
    - Content: (annotation?, (streamSequence | element | group | choice | sequence | any)*)  
  - `/streamChoice>`
  - `<streamSequence`  
    - id = ID  
    - {any attributes with non-schema namespace . . .}>  
    - Content: (annotation?, streamInfo?, (element | group | choice | sequence | any)*)  
  - `/streamSequence>`

Each of the above items corresponds to a particle containing a stream model group, with properties as follows:

- **Particle Schema Component**
  - **Property Representation**
  - {term}: A stream model group as given below:

- **Stream Model Group Schema Component**
  - **Property Representation**
  - {compositor}: One of streamChoice or streamSequence depending on the element information item.
  - {particles}: A sequence of particles corresponding to all the `<all>`, `<choice>`, `<sequence>`, `<any>`, `<group>` or `<element>` items among the children, in order. If compositor is streamChoice, then among the children may be `<streamSequence>` as well.
  - {streamInfo}: If the compositor is streamSequence: The streamInfo component according to the `<streamInfo>` child, if present. Otherwise absent. Else absent.
  - {annotation}: The annotation corresponding to the `<annotation>` element information item in the children, if present, otherwise absent.
Special comments on validation

For validation the engine must keep track of the individual states of the combined sequences, since streamChoice only tells that the next arriving item will belong to one of the combined sequences but gives no ordering. Therefore, the validation must be done for all combined sequences separately. Only if an arriving item does not fit into any of the inner sequences, the validation of the streamChoice must fail.

SchemaForSchemas stream model group

The new group "streamModelGroup" is shown in Figure 5.8, while Figure 5.9 shows the new types.

```xml
<xs:group name="streamModelGroup">
  <xs:annotation>
    <xs:documentation> This group is for the stream model groups which occur only at the top level of schemas. They are used to describe possibly infinite streams. </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="xs:streamSequence" type="xs:streamSequenceType"/>
    <xs:element name="xs:streamChoice" type="xs:streamChoiceType"/>
  </xs:choice>
</xs:group>
```

Figure 5.8: The new group "streamModelGroup"

StreamInfo Definition

To describe streams, we introduce the streamInfo Schema Component. It appears inside the streamModelGroup Schema Components.

The StreamInfo Definition Schema Component

The streamInfo definition schema component has the following properties:

- **Schema Component**: StreamInfo Definition
- **{annotations}**: A sequence of Annotation components
- **{repeating}**: The number of iterations, unbounded or infinite.
- **{interleaved}**: A boolean.
- **{streamNext}**: A set of streamNext Definition Schema Components.
- **{streamConstraint}**: A set of streamConstraint Definition Schema Components.

XML Representation of StreamInfo Definition Schema Component

The XML representation for a streamInfo definition schema component is a `<streamInfo>` element information item. The correspondences between the properties of these information items and properties of the component they correspond to are as follows:
<xs:complexType name="streamSequenceType">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:attributeGroup ref="xs:defRef"/>
      <xs:sequence>
        <xs:element name="streamInfo" minOccurs="0"
          type="xs:streamInfoType"/>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:nestedParticle" minOccurs="0"
          maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="streamChoiceType">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:attributeGroup ref="xs:defRef"/>
      <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="xs:streamSequence
          type="xs:streamSequenceType"/>
        <xs:group ref="xs:nestedParticle"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

Figure 5.9: The new types for streamModelGroups

XML Representation Summary: streamInfo Element Information Item

<streamInfo
  id = ID
  repeating = (nonNegativeInteger | unbounded | infinite) : 0
  interleaved = boolean : false
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, streamNext*, streamConstraint*)
</streamInfo>

The corresponding schema component is as follows:

StreamInfo Definition Schema Component
Property Representation
{annotation} The annotation corresponding to the
  <annotation> element information item
  in the children, if present, otherwise
  absent.
{repeating} infinite, if the actual value of the
  repeating attribute equals infinite,
  unbounded, if the actual value of the
repeating attribute equals unbounded, otherwise the actual value of the repeating attribute, if present, otherwise 0.

{interleaved} The actual value of the interleaved attribute, if present, otherwise false.

{streamNext} A set of streamNext Definition Schema Components, according to the <streamNext> element information items among the children, if present, otherwise absent.

{streamConstraint} A set of streamNext Definition Schema Components, according to the <streamConstraint> element information items among the children, if present, otherwise absent.

Constraints on XML Representations of StreamInfo Definitions

Schema Representation Constraint: StreamInfo Definition Representation OK

In addition to the conditions imposed on <streamInfo> element information items by the schema for schema documents, the following must be true:

1. The corresponding streamInfo definition must satisfy the conditions set out in Constraints on StreamInfo Definition Schema Components.

Constraints on StreamInfo Definition Schema Components

The following must be true:

- If the actual value of the interleaved attribute evaluates to true, there must be at least one streamConstraint child with its type attribute set to "key".

SchemaForSchemas streamInfo

Figure 5.10 shows the new type "streamInfoType".

Stream-constraint Definitions

Where subtitles and according subsections are missing, they are equal to the subsections of Identity-constraint Definitions. Just make sure to remove selector.

The Stream-constraint Definition Schema Component

The stream-constraint definition schema component has the following properties:

```
Schema Component: Stream-constraint Definition
{annotations} A sequence of Annotation components.
{name} An xs:NCName value. Required.
{target namespace} An xs:anyURI value. Optional.
{stream-constraint category} One of {key}. Default is key. Required.
{variety} An xs:Integer or infinite. Default is infinite. Optional if stream-constraint category is key.
{fields} A sequence of XPath Expression property records.
```
Stream-constraint definitions are identified by their name and target namespace; Stream-constraint definition identities must be unique within an XML Schema.

Stream-constraint category defines what purpose the constraint was defined for:

- **(key)** The Stream-constraint Definition Component asserts the component instances resulting from evaluation of the fields XPath expression(s) to have identical content. This allows assigning the component instances in the stream to the appropriate sequence instance. Therefore, every particle of the surrounding stream model group has to have elements that match to these paths.

These constraints are specified along side the specification of types for the attributes and elements involved, i.e. something declared as of type integer can also serve as a key. Each constraint declaration has a name, which exists in a single symbol space for constraints. The identity conditions appealed to in checking these constraints apply to the values of the fields selected, not their lexical representation, as with the Identity-constraint definitions.  

{fields} specifies XPath expressions relative to the stream model group they are contained in. These must identify each a single node (element or attribute) whose content or value, which must be of a simple type, is used in the constraint. All fields defined here must appear in each particle of the surrounding stream model group, if category is **key**. It is possible to specify an ordered list of {fields}s, to cater to multi-field uniqueness and equality constraints.

In order to reduce the burden on implementers, in particular implementers of streaming processors, only restricted subsets of XPath expressions are allowed in fields, namely the same as defined for the fields of Identity-constraint Definitions.
5.2. THE XML SCHEMA EXTENSION

XML Representation of Stream-constraint Definition Schema Components

The XML representation for a stream-constraint definition schema component is a <streamConstraint> element information item. The correspondences between the properties of these information items and properties of the component they correspond to are as follows:

XML Representation Summary: streamConstraint Element Information Item

```
<streamConstraint
  id = ID
  name = NCName
  type = (key) Required.
  variety = (nonNegativeInteger | infinite) : infinite Optional.
  xpathDefaultNamespace = (anyURI | (#defaultNamespace
    | #targetNamespace | #local))
{any attributes with non-schema namespace . . .}>
  Content: (annotation?, field+)
</streamConstraint>

<field
  id = ID
  xpath = a subset of XPath expression
{any attributes with
  non-schema namespace . . .}>
  Content: (annotation?)
</field>
```

The corresponding schema component is as follows:

Stream-constraint Definition Schema Component

Property Representation

{name} The actual value of the name attribute
{target namespace} The actual value of the targetNamespace attribute of the ancestor schema element information item.
{stream-constraint category} The actual value of the type attribute.
{fields} A sequence of XPath Expression property records corresponding to the <field> element information item children, in order, following the rules given in XML Representation of Assertion Schema Components ([6, §3.13.2]), with <field> as the "host element" and xpath as the designated expression attribute.
{variety} If the stream-constraint category is key, the actual value of the variety attribute, if present, otherwise (variety attribute not present) "infinite".
{annotations} The annotation mapping of the set of elements containing the <streamConstraint> element, and the <field> children, if present, as defined in XML Representation of Annotation Schema Components ([6, §3.15.2]).
Constraints on XML Representations of Stream-constraint Definitions

Schema Representation Constraint: Stream-constraint Definition Representation OK

In addition to the conditions imposed on <streamConstraint> element information items by the schema for schema documents, the following must be true:

1. The corresponding stream-constraint definition must satisfy the conditions set out in Constraints on Stream-constraint Definition Schema Components.

Constraints on Stream-constraint Definition Schema Components

All stream-constraint definitions (see Stream-constraint Definitions) must satisfy the following constraints.

- Schema Component Constraint: Stream-constraint Definition Properties Correct
  
  The following must be true:
  
  The values of the properties of a stream-constraint definition are as described in the property tableau in "The Stream-constraint Definition Schema Component", modulo the impact of Missing Sub-components ([6, §5.3]).

- Schema Component Constraint: Fields Value OK
  
  Same as for Identity-constraint definitions.

SchemaForSchemas stream constraint

The new element streamConstraint can be seen in Figure 5.11, together with its internal type.

streamNext

The streamNext Schema Component

The streamNext schema component has the following properties:

```xml
<streamNext
   {annotations} A sequence of Annotation components.
   {selector} An XPath Expression property record.
   {next} A set of rules describing the next instance of the elements matched by the selector.
```

XML Representation of streamNext Schema Component

The XML representation for a streamNext schema component is a <streamNext> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

```xml
XML Representation Summary: streamNext Element Information Item
<streamNext
   id = ID
   selector = a subset of XPath expression. Starts with './'.
   Can contain node tests at its end.
   {any attributes with non-schema namespace . . .}>
```
<xs:element name="streamConstraint">
  <xs:annotation>
    <xs:documentation>
The stream-constraints to restrict stream model groups.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:element ref="xs:field" maxOccurs="unbounded"/>
        <xs:attribute name="name" type="xs:NCName" use="required"/>
        <xs:attribute name="type" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:enumeration value="key"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="variety" use="optional">
          <xs:simpleType>
            <xs:union memberTypes="xs:nonNegativeInteger">
              <xs:simpleType>
                <xs:restriction base="xs:NMTOKEN">
                  <xs:enumeration value="infinite"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:union>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

Figure 5.11: The new element "streamConstraint"

<nxt:comparator
  id = ID
  type = (greater | greaterequal | equal | smallerequal | smaller) Required.
  {any attributes with non-schema namespace . . .}>
Content: (annotation?, (nxt:first | nxt:next | nxt:current | nxt:value | nxt:operator){2})
</nxt:comparator>
<nxt:operator
  id = ID
CHAPTER 5. AN EXTENSION FOR XML SCHEMA

type = (addition | substraction | multiplication | division) Required.
{any attributes with non-schema namespace . . .}>
Content: (annotation?, (nxt:first | nxt:next | nxt:current | nxt:value | nxt:operator){2})
</nxt:operator>

<nxt:operator>
<nxt:first>
  id = ID
  xpath = a subset of XPath expression. Starts with "./".
  {any attributes with non-schema namespace . . .}>
  Content:
  </nxt:first>

<nxt:next>
  id = ID
  xpath = a subset of XPath expression. Starts with "./".
  {any attributes with non-schema namespace . . .}>
  Content:
  </nxt:next>

<nxt:current>
  id = ID
  xpath = a subset of XPath expression. Starts with "./".
  {any attributes with non-schema namespace . . .}>
  Content:
  </nxt:current>

<nxt:dependency>
<nxt:current>
  id = ID
  xpath = a subset of XPath expression. Starts with "./".
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, nxt:current, nxt:next)
  </nxt:dependency>

<nxt:current>
  id = ID
  xpath = a subset of XPath expression. Starts with "./".
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (nxt:value | nxt:comparator | nxt:set))
  </nxt:current>

<nxt:next>
  id = ID
  xpath = a subset of XPath expression. Starts with "./".
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (nxt:value | nxt:comparator | nxt:set))
  </nxt:next>

<nxt:set>
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: nxt:setElement+
  </nxt:set>

<nxt:setElement>
  id = ID
  {any attributes with non-schema namespace . . .}>
}
The corresponding schema component is as follows:

```
next value facet Schema Component
  Property Representation
  {selector}   An XPath Expression property record.
  {next}       The rules represented by the <nxt:comparator> and <nxt:dependency> element information items among the children.
  {annotations} The annotation corresponding to the <annotation> element information item in the children, if present, otherwise absent.
```

Constraints on XML Representation of streamNext Schema Component

Schema Representation Constraint: streamNext Representation OK

In addition to the conditions imposed on <streamNext> element information items by the schema for schema documents, the following must be true:

1. The corresponding streamNext definition must satisfy the conditions set out in Constraints on streamNext Schema Components

2. The actual value of the selector attribute of the <streamNext> element information item must be an XPath expression property record, starting by "./" and using element names and "/" or starting with "./" and giving a node test at its end to allow multiple element children of the streamSequence to be constrained by the next rules.

It is left to the writer of the Schema to guarantee that the next rule uses the correct datatypes for the fields figuring in the rules. The writer also has to know herself, that the next value constraints are only parsed and useful if used for direct children of a stream model group.

Constraints on streamNext Schema Components

All streamNext Schema Components must satisfy the following constraints.

- Schema Component Constraint: StreamNext Properties Correct

  The following must be true:

  - Selector must match only elements which are defined such to have all the fields used in the xpath attribute of the next, current and first elements used in the next rules.
Figure 5.12: Schema with import and the new element "streamNext" with its internal type.

**Schema For Schemas streamNext**

The new element next can be seen in Figure 5.12. The schema for the elements of the nxt namespace is given below.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:nxt="http://www.dbis.ethz.ch/streamingXQueryP-nxt"
    targetNamespace="http://www.dbis.ethz.ch/streamingXQueryP-nxt"
    elementFormDefault="qualified">
    <xs:element name="dependency">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="xs:annotation" minOccurs="0"/>
                <xs:choice maxOccurs="unbounded">
                    <xs:element ref="nxt:comparator"/>
                    <xs:element ref="nxt:dependency"/>
                </xs:choice>
            </xs:sequence>
            <xs:attribute name="selector" use="required">
                <xs:simpleType>
                    <xs:annotation>
                        <xs:documentation>A subset of XPath expressions for use
                            in streamNext selector. Relative from the streamSequence
                            and allowed to contain node tests at its end.
                        </xs:documentation>
                        <xs:documentation>A utility type, not for public use
                        </xs:documentation>
                    </xs:annotation>
                    <xs:restriction base="xs:token"/>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
    </xs:element>
</xs:schema>
```
<xs:complexType>
  <xs:attribute name="xpath">
    <xs:simpleType>
      <xs:annotation>
        <xs:documentation>A subset of XPath expressions for use in fields or streamNext {next,current,first}/@xpath</xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:token"/>
    </xs:simpleType>
  </xs:attribute>
  <xs:choice minOccurs="2" maxOccurs="2">
    <xs:element ref="nxt:first"/>
    <xs:element ref="nxt:next"/>
    <xs:element ref="nxt:current"/>
    <xs:element ref="nxt:value"/>
    <xs:element ref="nxt:operator"/>
  </xs:choice>
  <xs:attribute name="type">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration>greater</xs:enumeration>
        <xs:enumeration>greaterequal</xs:enumeration>
        <xs:enumeration>equal</xs:enumeration>
        <xs:enumeration>smallerequal</xs:enumeration>
        <xs:enumeration>smaller</xs:enumeration>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
<xs:choice minOccurs="2" maxOccurs="2">
  <xs:element ref="nxt:first"/>
  <xs:element ref="nxt:next"/>
  <xs:element ref="nxt:current"/>
  <xs:element ref="nxt:value"/>
  <xs:element ref="nxt:operator"/>
</xs:choice>

<xs:attribute name="type">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration>addition</xs:enumeration>
      <xs:enumeration>substraction</xs:enumeration>
      <xs:enumeration>multiplication</xs:enumeration>
      <xs:enumeration>division</xs:enumeration>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
</xs:complexType>

<xs:element name="next" type="streamNextxpath">
  <xs:element name="first" type="streamNextxpath">
    <xs:element name="current" type="streamNextxpath">
      <xs:element name="set">
        <xs:complexType>
          <xs:setElement maxOccurs="unbounded" type="PCDATA"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="value" type="PCDATA"/>
    </xs:complexType>
  </xs:element>
</xs:element>

<xs:element name="value" type="PCDATA"/>

<xs:complexType name="streamNextxpath">
  <xs:attribute name="xpath">
    <xs:simpleType>
      <xs:annotation>
        <xs:documentation>A subset of XPath expressions for use in fields or streamNext {next, current, first}/@xpath</xs:documentation>
        <xs:documentation>A utility type, not for public use</xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:token"/>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
</xs:element>
Chapter 6

Integration into processing models

In this section we analyse how the new schema extension integrates into existing processing models. We do this by looking at XQuery(P)[8, 9, 13, 14, 15] and Aurora[27]. The chapter is divided into three sections: In 6.1 we deliver a short introduction to the models, 6.2 shows the changes in semantics made possible by the extension and finally 6.3 lists the optimisations enabled by describing the stream.

We tried to analyse StreamSQL [25, 26] as well, but we were not able to find or receive any information about the processing model of StreamSQL and its assumptions on streams. Therefore, we decided to analyse only XQuery(P) and Aurora.

6.1 Short introduction

6.1.1 XQuery(P)

XQuery\(^1\) and its extensions make two main steps when evaluating a query [8]:

1. In the **static analysis phase** the query is compiled into the needed internal representation, an operator tree. Then the static context is initialised and filled with additional information from the prolog. If at the end of this phase the operation tree contains a type, function or variable name or namespace prefix that is not in the static context, a static error is thrown. After the initialisation, to each expression in the tree a static type is assigned. If static typing feature is in effect, a type error is raised. If an inferred type of an expression does not conform to that operand, a type error is raised. If the query passes the static analysis phase, its execution on valid input is guaranteed to produce a value of its result type or to raise a dynamic error.

2. During the dynamic evaluation phase the value of the expression is computed by evaluating it on the input data and the dynamic context. In this phase new data model values can be generated and the dynamic context can be extended. The dynamic context gets its information from the external environment and the static context. Each computed value is assigned a dynamic type, which might differ from the static type in being more specific. Even at this stage it might happen that an assigned dynamic type does not conform to an operand, so in this case a type error is raised.

The input to a query engine is therefore a data model instance and a query. The data model instance can also be generated from some other data. Mostly it will be generated from an XML file: The XML content is parsed into an instance of Infoset [4], the data model of XML. Infoset

\(^{1}\)http://www.w3.org/TR/2007/REC-xquery-20070123/#id-processing-model
can also be validated against an XML schema [5], resulting in a PSVI, a Post-schema-validation Infoset. From this Infoset or PSVI one finally generates the instance of its data model (XQuery Data Model, XDM [12]), XDM, as input to the XQuery processor. These steps will be integrated into most XQuery processing engines.

The full processing model overview can be seen in Figure 6.1.

![Figure 6.1: XQuery processing model overview. Image from [8, Section 2.2]](image)

### 6.1.1.1 XQuery(P) and streams

#### Finite streams

In XQuery every instance of XDM is an ordered sequence of zero or more nodes or atomic values. Atomic values are of a type defined in XML Schema [7] or derived from one of them. Nodes are either document, element, attribute, text, comment or processing instruction nodes. Therefore, every instance of the XDM is a collection of (sub)trees representing the according document (fragments), itself again structured by ordering.

Until this point there are no infinite instances of XDM. But since streams can also be finite, every finite stream is a valid instance of XDM and vice versa. So up to this point validation is defined over streams.

#### Infinite streams

Kraska wrote in [1, Section 4] an extension to XDM, allowing infinite instances of XDM on its top level. This extension seems not to change much, since the only visible difference is that a sequence itself can be infinite. The problem comes with validation and processing. The processing problems were already discussed in Section 3.1.2, existing and proposed solutions in Section 3.1.3 and Chapter 4. Kraska also states that with infinite streams non-blocking operators should output results before
all the input is consumed. Then he asks to change the semantics the way that in case of an error
the error is raised at the moment of its happening, even if there was already some output.

Validating infinite streams

What is open at this point is how to validate an infinite XDM instance. A full validation of the
stream in advance of query processing is no more possible due to the infinity of the stream. At
least elementwise validation is possible as soon as an element has arrived completely. After element
validation the element can be processed. The only possible way to validate a stream seems to prefix
validation, such that one can always say "up to the current position, the stream is valid". This leads
to the question of "what to do if at one moment, the stream is no more valid?" There are various
answers to this question:

- Throw an error at this position and abort the evaluation of the rest, since the stream is not
  valid. One might even declare the output so far (if any) as invalid.

- If only optimisations are realised using the schema, disable them from this moment on, if
  possible, switching back to the original query and its evaluation plan. This is possible, since
  the optimisations only make the evaluation faster but do not change semantics or results.

- More general, failure of validation could be used as triggers for certain actions, like above
to disable optimisations. Or as shown in Section 9.1 about the loss pattern, one could use
validation failures as triggers for certain actions, like inserting a loss event.

Infinite streams with schema extension

One more point is added to the validation questions: How to validate the new stream information
items from the extensions of this report:

- The validator needs state to remember previous items to check the next constraints and the
  streamConstraints.

- Validation of sequences must happen on level of interleaving instances. Since a streamSequence
describes a pattern that might appear on the stream in multiple instances at the same time
(interleaving), one has to decide on the keyConstraint value to which sequence it belongs and
with the last element of a sequence validate its state.

- Validation of sequences should happen elementwise and with respect to repetition. If a se-
  quence itself is infinite one can only validate it elementwise or at most by means of declaring
every iteration as valid or invalid.

- As soon as one repetition of an instance of a possible interleaving sequence is invalid the whole
  or only the following part of the sequence are to declare invalid.

- As soon as one sequence gets invalid the whole stream gets invalid, either from the moment
  of the invalidation or in total.

These are only proposals for possible stream validation strategies. It is left as topic for future work,
how streams are to be validated and what are the consequences of a failure of stream validation.

6.1.1.2 Refinement of streaming semantics

To be able to describe streams precisely, we require a small change in the stream semantics of
XQueryP:
End of stream

An "end of stream" signal must be an intended one. This means, if a stream aborts, breaks or gets silent for too long (causing a timeout), this should be reported or caught as an error/failure, not as "end of stream". This enables us to deal with some changes in semantics as well as some optimisations stated later on. Note that with this change only finite streams can cause an "end of stream".

Stopping a query execution

Wherever in the optimisations it is said that the execution is stopped, this means an instant stop, without signalling "end of stream" and therefore without producing open windows, since the stream would not have stopped at this point, but maybe even run forever. This makes sense if one looks at why we are stopping the processing: We only stop if we know, that no more results can be generated based on future input, e.g. no more window can be closed and evaluated. Therefore, we do not miss any result. The only change an early stop causes is a change in semantics, since now a query can terminate that would have run forever without the information about future elements.

Note, that only infinite streams should be stopped early, since only infinite streams would not produce open windows at the end (infinite streams have no end). If a query on a finite stream is stopped early, we may miss start events of windows that would be produced at the "end of stream" signal.

6.1.2 Aurora

With Aurora [27], one can create continuous and ad hoc queries as well as views, using mostly the same mechanisms by the popular boxes and arrows system. For this, operators are shown as boxes connected by arrows representing streams. Arrows actually represent a collection of streams with common schema. Every box can be connected to any number of following boxes, implicitly splitting the stream up into several. On arrows one can place connection points, which are access points for dynamic network modification. To be able to insert a new box at a connection point, this box needs also some information about tuples already passed by. Therefore, the connection points keep a history, the size definable by the user, which is persisted (this storage is often solved internally by using a BTree). Connection points also allow including of static data sets in to an Aurora network.

For doing this, one creates a connection point without upstream node and attaches its output to the nodes working on the static data. The three modes of queries are shown by examples in Figure 6.2. The topmost one is a continuous query, the one in the middle is a view and the third an ad hoc query.

As the name says, continuous queries run forever, if needed, being fed at the entry points with new tuples as they arrive and constantly outputting tuples at the endpoints of the network. Tuples that have proceeded through all paths are drained from the network.

Views are paths that are not connected to an application constantly. Applications can connect to them whenever they need to and pull next tuples out of it. These tuples may be pulled through all the path at the moment the application pulls or may be processed in advance and stored at any point along the path by the system to reduce latency.

Ad hoc queries can be attached to a connection point at any time and will produce results for the tuples stored in this point and all the following ones until the user disconnects the path again. Therefore, it is the same as a continuous query that simply starts at a later point in time than the rest of the network.

Although Aurora works push based, so every tuple gets pushed through the network as it arrives, ad hoc queries and views work at least partly pull based. As soon as storage is in use, like a connection point used to make stored data e.g. from a database available, the tuples can be
pulled as well as pushed. Here, pulling brings one advantage: If the first operator after a connection point accessing a DB is a filter, it can pull only those tuples that will pass it and avoid processing a lot of unneeded items. Another example are ad hoc queries: the moment they are connected to a connection point, the tuples in the storage of the point are pulled through the network. But as soon as the history is "used up", Aurora switches to the normal push based functionality, processing the ad hoc query path for each tuple as it arrives.

6.1.2.1 **Aurora and streams**

In Aurora a "stream is an append-only sequence with uniform type (schema)"[27, Section 5.1]. In addition to the payload fields of the application each tuple has some additional information fields as a sort of a header. This header contains the timestamp of its arrival in the system, a unique id, the type and some more. The type field is used by various operators to declare some special tuples. For example, a tuple can be marked as "revision tuple", which means that it should be understood as an update for some tuple already arrived. The type can also be used to denote some characteristics of the stream in some sort of punctuation notation (inspired by punctuations [23]). In Borealis[28], the distributed version of Aurora, there is a second timestamp field denoting the arrival time on the current node, which is updated whenever the item is sent to another node.

Against common practise Aurora does not assume its tuples to arrive in a sorted manner regarding some value like timestamp. Like that Aurora itself has not to guarantee ordered output, which might be impossible anyway (e.g. when merging streams) and can therefore schedule tuple processing more relaxed like processing first some high-priority tuples and then continuing on the normal arrivals. In addition, Aurora can use windowed queries on any attribute, not only the ordered one, which is another reason why it does not take the ordering assumption.

Since without these assumptions some items might not be ordered, the order-sensitive operators of Aurora need some arguments informing about ordering of arriving tuples. These arguments are specifications consisting of the name of the field on which the input will be ordered, an integer named \( \text{slack} \) and some field names the input is grouped by. The slack value tells the relaxation of the strict ordering. E.g. if it is 0, all tuples arrive in order, but if it is 1, there might appear one tuple before the one that should arrive when ordered correctly. So it tells the number of tuples a tuple may be misplaced in terms of order. If grouping is given, the slack variable assumes a prefiltering by the grouping fields, telling therefore only how many tuples of the same group may arrive before the one of the correct ordering. So the slack variable defines sort of a presorting window size. This resembles a lot to the adherence parameter \( k \) of the \( k \)-Constraints[31].
If an order-sensitive operator receives a tuple which is out of order with respect to the given specifications it simply discards the tuple. Like this there is no possibility of blocking the system by asking e.g. for a sort. Order-sensitive operators discard out-of-order items respecting the given ordering specifications. And for sort Aurora only supports the BSort operator, which is a bounded sort where you have to give the ordering specification that should be fulfilled in the output and which is impossible to use for strict ordering on infinite streams.

One part where ordering is needed are windowed aggregates. As long as the window is not defined over a tuple size but a value range (e.g. "span the window over all tuples having a value in field x smaller than the value of field x of the current tuple+ 1"), the field whose value is used to decide about window production is assumed to be ordered.

For its QoS and load shedding strategies, Aurora has also some additional fields in the header of the tuple, which denote if the tuple will be part of a window or starts a window. These fields can also contain some information about what values are not to appear anymore, seeming similar to punctuations\textsuperscript{23}. Like that the system can decide better which tuples it can drop causing less reduction of result accuracy.

6.1.2.2 Optimisation strategy

Aurora optimises queries dynamically by starting the query network in its original version. It breaks the network into subpaths between connection points. When optimising such a path it lets the input points hold back all arriving messages and processes all tuples still in the path. Then it optimises the path by various rules, constructs a new subpath and if its metrics state that it performs better, the paths are exchanged and the connection points instructed to let the data flow again. Like this, the network around this path, but not using it, is not affected by the blocking. Only the paths being fed by the one in progress will have a short growth in response time. Aurora allows only changes of paths without a connection point inside, which is exactly how this procedure works. The optimiser, a separate thread, cycles through all subnetworks between the connection points repeatedly and tries to optimise them.

For ad hoc queries in pull mode the procedure is slightly different. They are processed before starting the pulling as said above; if the following node is some filter and its attributes match the ones used by the BTree (used internally for storage in the starting connection point), this BTree is used to just pull the elements that would pass the filter. After switching to push mode it is optimised like the other subnetworks.

Some operators can hold a timeout after which they close a window if its closing element has not arrived until then. This helps to reduce the number of open windows.

6.1.2.3 QoS and load shedding

The core point in Aurora is that every output is associated with two-dimensional QoS graphs to specify the "utility of the output in terms of several performance-related and quality-related attributes" [27, Section 2.1]. These specifications are used to control how the system allocates resources to process the tuples on their way. The Aurora application administrator can give the system three QoS graphs. For every output stream there is at least one graph expected, giving the relation between QoS and delay in tuple processing using a threshold to tell after which delay the QoS degrades.

One can give two additional graphs per output. One shows the relation of QoS to percentage of delivered tuples and the other the relation of QoS to output value. With the first, Aurora is told how important it is not to drop too much or even no tuples while with the latter one can assign different importance to tuples depending on their value. An example for the latter is a power plant where sensors watch over the temperature in the reactor. Tuples near warning thresholds must
come through under all circumstances while tuples reporting standard working temperature are less important.

According to the current states in the graphs at each output, Aurora schedules operations on the individual paths, subpaths (called trains) or operators. In case of overload in the network Aurora drops tuples where needed and where possible. Again these information are gained from the importance assignments in the QoS graphs.

Another load shedding possibility has been introduced by Tatbul et al. [29], named window drop. This operator does not drop random tuples like the other ones used in Aurora, but analyses the window tuple and step size and drops whole windows by setting a special field in the tuple, which would open the window. Windowed operators do not start a window when this field is set to 0. The field is set to a positive value of the window condition field, indicating that all later tuples up to those having this value must be left in the stream as well. Therefore, the operators know on one side that they can start a window with this tuple and all other drop operators know that they can not drop this and following tuples to keep the integrity of the windows these tuples will figure in. Using window drop, instead of using tuple drop, the result by the operators are a subset of the results produced without load shedding, whereas when dropping single random tuples windows contain other tuples and therefore produce different window results.

6.2 Changes in semantics

This section will show how our extensions can change the semantics of the discussed processing models. They are divided into three categories:

- Warnings and aborts: By looking at different levels of information, the engine should be able to warn the user or even abort directly, if the given query may lead to blocking behaviour or is impossible to execute because of other causes.

- Operators: This section looks at adapted operators, which can benefit from the additional stream schema information. The possible changes may look familiar, since these are the ones that are also possible if punctuations [23] are used.

- Scheduling / planning / threading possibilities: By looking at the stream descriptions, the engine might deduce some information about thread usage, ordering and scheduling of executions. In this area, one has to watch out for pitfalls like windows getting produced when the stream stops independent from the end condition, which would not be possible to happen if the stream is infinite.

The semantic adaptations may depend on each other, applying one might trigger another. This is done to split up the individual adaptations into parts which will be used as base for other adaptations. To show these enabling parts of adaptations, there are some adaptations listed which are enabled without using our schema extensions (where "used information" is "none"). To use the transitivity built by this splitting strategy, the engine should loop testing for the various adaptations until no more applicable are found. If using semantic adaptations and optimisations, they should all be checked in the same loop.

Wherever possible, changes in semantics are described according to the following pattern:
- Context: describes the situation, what the query looks like, surroundings.
- Used Information: Which information from the schema is used.
- Action: What the optimiser should do.
6.2.1 XQuery(P)

6.2.1.1 Warnings and aborts

Blocking operator (on infinite sequences)

Blocking because all input will be read before outputting (blocking operator) is impossible to pre-
vent, if behaviour should stay the same (nearly the same behaviour possible, e.g. by using incre-
mental/running aggregates instead of the real ones).

Context: Blocking operator used, no alternative for this operator. Independent of operator
position in query.

Used information: Input sequence is infinite.

Action: Abort.

Infinite sequences

Blocking because of infinite sequences may be wanted, e.g. a FOR over a stream of events to alarm
in case of unusual reports. This is not really blocking, just an infinite query. If we could enable
pipelining and therefore return values before reading all input, everything can be fine).

Example 1

Context: FOR used on stream.

Used information: Stream is infinite.

Action: Warn.

6.2.1.2 Operators

Unblock

If some conditions are met, certain operators do not have to wait for all their input before producing
any output. For example look at the sort or order by operator. Normally it would have to read
all the input before being able to sort and output a result. But if it has the information that the
input is at least partly or even fully sorted, it can output some results early. If the schema says
that values arrive partly sorted, e.g. sorted increasing with respect to digits from third position and
higher, the operator can output its sorted values as soon as the next arriving item has a new value
at its third position.

Example 1

Context:

    for $item in $stream
    order by $item/myIntValue
    return $item/$name

Used information: $stream is infinite, consists only of one iterating element and arrives sorted with
respect to the digits of the hundreds column and higher.

Action: Set order operator to release generated sorted output as soon as third position of arriving
item changes with respect to its predecessor.
6.2.1.3 Scheduling / planning / threading possibilities

Set early stop points

If we can derive from all available information that after some point in time an end condition can never match again, we can stop listening to the stream.

- If the stream is infinite, we set an early stop, such that no "end of stream" appears. This is to prevent a production at an "end of stream" that without our intervention never would have happened.

- If the stream is not infinite, we terminate the input only early if force is used. If force is not used, we do not terminate early to not prevent the still open windows (or the ones being opened until the "end of stream") from producing at the real "end of stream".

Example 1

Context: The end or where clauses are chosen such that after some point they will never result to true again:

```xml
forseq $w in $seq landmark window
  start when fn:true()
  end position $e when $e = 100
return $w
```

Used information: stream is infinite
Action: stop evaluation after item position 100.

Example 2

```xml
forseq $w in $seq landmark window
  start curItem $s when $s/@start eq true
  end position $e when $seq[$e]/@end eq true
  where $e < 100
return $w
```

Used information: stream is infinite
Action: stop evaluation after item position 100.

Example 3

Context: Same, but other end condition possibility:

```xml
forseq $w in $seq landmark window
  start when fn:true()
  force end curItem $e when $e[self::midyear]
return $w
```

Used information: sequence consists of some elements, a <midyear> and some more elements, is not repeating and not interleaved. Therefore, <midyear> appears only once.
Action: stop listening after <midyear> (finite, but force is used).
6.2.2 Aurora

6.2.2.1 Warnings and aborts

The same operations are possible as for XQuery:

- The system can abort or even decline a query of which it knows that there will be no results because the stream does not contain any tuples responding to the query.
- Early stop points are triggable using stream descriptions about next values only as well as in combination with the ordering specifications and punctuation-like header fields.

6.2.2.2 Operators

If a sort operator would not only know that the stream it has to sort is infinite but that it is infinite and already (or at least partially) ordered, it can again be enabled to sort the input totally. Since our next rules and the order specification of Aurora do not have the same way to express ordering, the next rules can not replace order specifications used in Aurora [27, Sections 5.1 and 5.2.2]. The difference is that we tell in what value range the tuples are already sorted and order specification give the tuple range inside which the tuples may be unordered. Still they can be combined to enable more operations like unblocking operators by telling the operator not some slack variable to span the sort window over but by telling it that it has to span the sort window over all incoming tuples until some digit (e.g. the third from the right for the hundreds column) changes. Like this the infinite stream is virtually broken down into finite streams to be strictly ordered.

Another point where our stream descriptions could help, is the timeout Aurora places on windows. Aurora sets some timeout, such that if the tuple needed for closing a window does not appear inside the specified time, the window waits no more to close but is discarded. If our descriptions could tell the system that the expected item must appear or even will appear soon (in terms of tuples to wait), this timeout could be relaxed. Knowing that an item must appear soon is done e.g. by next rules, if we wait for an item with a value below a certain threshold and a next rule says that this value drops for every arriving item at minimum by some specified value and the current item value is already near the limit. Of course, Aurora has to keep the timeout in case of queries which need answers in a certain timeframe.

6.3 Optimisations made possible

So far we have figured out possible changes in semantics. Now we want to define what optimisations are enabled with which information about the streams, including concrete use cases. Again, the optimisations are split according to the discussed processing models.

The optimisations may depend on each other, applying one might trigger another. This is done to split up the individual optimisations into parts which will be used as base for other optimisations. To show these enabling parts of optimisations there are some optimisations listed which are enabled without using our stream descriptions (where "used information" is "none"). To use the transitivity built by this splitting strategy, the optimiser should loop testing for the various optimisations until no more applicable are found. If using semantic adaptations and optimisations, they should all be checked in the same loop.

The optimisations are written using the same pattern as the semantic changes (see Section 6.2) and are divided into the following sections:

- Query rewritings: Using information about iterations, intervals and value, the engine should be able to rewrite queries to optimise the required computations and amount of items to read.
6.3. OPTIMISATIONS MADE POSSIBLE

- Operator optimisations: This section looks at adapted operators that can benefit from the additional stream schema information. The possible optimisations may look familiar, since these are the ones that are also possible if punctuations [23] are used.

- Memory optimisations: Using size of elements or number of windows that will use an element, decide about pipelining, early tuple dropping, removing hopeless windows and early production in terms of required, impossible, useful or nice to have and enable them appropriate.

- Scheduling / planning / threading possibilities

6.3.1 XQuery(P)

6.3.1.1 Query rewritings

General FORSEQ → landmark window

Remember: General forseq would generate multiple variants of a subsequence with same start and end, the difference to windowed forseq is that not all elements between start/end must be contained, or in other words, elements do not have to be directly following each other. Therefore, the query having a where clause limiting the subsequence length to e.g. 2 (by using length() or just by the used "start/end conditions") does not point to sliding windows, since general forseq would return the start element paired up with each following element as a two-element sequence. Therefore, a rewrite from general forseq to windowed forseq can only be done when the user obviously checks for a consecutive sequence. The detection of such queries seems to be very hard, since it may need function analysis. In the following example we use the function continuousSeq() to detect if a sequence has no gaps in it. But since even this function is not completely the right way to detect continuous sequences, we think that it will not be possible to make the rewrite from general to windowed forseq.

Example 1

Context: forseq with where clause to assure that all elements in-between are contained (no jumps in sequence):

```xquery
forseq $s in $seq
  where continuousSeq($s)
  return $s
```

or

```xquery
forseq $s in $seq
  where $s[1]/@action eq "in" and $s[last()]/@action eq "out"
    and continuousSeq($s)
  return $s
```

Used information: none

Action: rewrite to landmark window. Start and end conditions are initialised to true. Later optimisations will copy corresponding parts from the where into the start and end clauses. Remove the continuity checker from the where:

\[\text{continuousSeq()} \text{ is declared in [1, Section 3.2]}\]
forseq $s$ in $seq$ landmark window
start when fn:true()
end when fn:true()
return $s$

or
forseq $s$ in $seq$ landmark window
start when fn:true()
end when fn:true()
where $s[1]/@action eq "in" and $s[last()]/@action eq "out"
return $s$

Landmark $\rightarrow$ sliding window

For this optimisation, we look if an end condition can only be fulfilled once per window. If that is the case and we know that the input is infinite, we do not have to keep it open and try to match the end condition again and again, like a landmark window would do. Instead, we can switch to a sliding window, that is removed after being produced.

Note that for this decision we do not only look at the evaluations of the end condition. As second indicator we need to differentiate between finite and infinite streams. The fact, that a window gets produced if it is open when the stream ends is important. If we do ignore it here and rewrite a landmark window without force or restricting where clause to a sliding window, we cause a change in semantics since we reduce the number of window evaluations. On the other side, if we know that the stream is infinite we do not have an end to produce at and therefore can apply this transformation. The same fits for force, since windows using force do not get produced at end of stream too if the end condition is not fulfilled.

As third possibility to enable sliding windows we use the where clause: If the where condition e.g. states that a certain element has to be contained only once in the window and we know from the schema that this element will occur every time before the end condition matches we know that only at the first end match the where clause is fulfilled. So we can again rewrite the landmark window to a sliding window. We can not remove the where clause after rewriting because if the stream is finite possibly open windows not fulfilling the where clause would be produced. Therefore, we have to keep that check. Another optimisation later on can remove it if the stream is infinite.

Example 1
Context: Landmark window whose start and end conditions imply that only one position fulfils the end condition. Since force is used, window will not produce at end of stream:

forseq $w$ in $seq$ landmark window
start position $s$ when fn:true()
force end position $e$ when $e-s=5$
return $w$

Used information: none
Action: rewrite landmark to sliding:

forseq $w$ in $seq$ sliding window
start position $s$ when fn:true()
force end position $e$ when $e-s=5$
return $w$
Example 2

Context: Landmark window whose where clause implies that only one position fulfils the end condition:

```xquery
forseq $w in $seq landmark window
  start position $s when $seq[$s]/@action eq "start"
  end position $e when $seq[$e]/@action eq "stop"
  where count($w[@action eq "stop"]) eq 1
return $w
```

Used information: stream is infinite

Action: rewrite landmark to sliding:

```xquery
forseq $w in $seq sliding window
  start position $s when $seq[$s]/@action eq "start"
  end position $e when $seq[$e]/@action eq "stop"
  where count($w[@action eq "stop"]) eq 1
return $w
```

Example 3

The examples before concern the standard detection methods, they are directly triggered from the query alone. Schema extensions help if they tell us that for each window only one element fulfilling the end condition will occur.

Context: Stream of deliveries to clients in one year. System should sum up prices of sold products in the first six months per client. First delivery to a client in a year is detected by having @seqid="1", after 6 months a `<midyear>` is injected:

```xquery
forseq $w in $seq landmark window
  start curItem $s when $s[self::delivery]/@seqid eq 1
  force end curItem $e when $e[self::midyear]
return sum($w/totalprice)
```

Used information: Since stream contains data from one single year, it is a non-repeating and non-interleaving sequence with four children: (sequence of deliveries, `<midyear/>`, sequence of deliveries, `<endyear/>`). So midyear only appears once.

Action: rewrite landmark to sliding.

Example 4

This could be extended to infinitely iterating sequences if the elements contain dates:

Context: same as before, but stream contains more than one year and each element has date information integrated. Query now checks for "in same year"

```xquery
forseq $w in $seq landmark window
  start curItem $s when $s[self::delivery]/@seqid eq 1
  force end curItem $e when $e[self::midyear]
  and $e/@year eq $s/@year
return sum($w/totalprice)
```
CHAPTER 6. INTEGRATION INTO PROCESSING MODELS

Used information: same top-level sequence as above, but now repeating. Next rule:
next((self::midyear, @year), "next > current"). Therefore, only one midyear will have the same year.

Action: rewrite landmark to sliding:

```
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 $s/@year
  return sum($w/totalprice)
```

Note: removing the additional and now unused clause may be hard, e.g. if the above end condition would combine both conditions (when $e/self::midyear/@year eq $s/@year).

Example 5

Yet another interesting use case is the following:

Context: where clause will only be fulfilled once, but not as before because <middle> appears only once. Stream will be a series of sequences, marked with start, middle and end.

```
forseq $w in $seq landmark 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 but not interleaved, all elements of the sequence are to appear once every iteration.

Action: Since between two <end> a <middle> is to appear, every window will contain too much <middle> after the first <end>. Before the first <end>, only one <middle> will appear. So we can stop watching for <end> in a window after the first one. This equals rewriting to sliding window:

```
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
```

Sliding → tumbling window

Now we look for cases where the conditions are even more restrictive, such that there can only be one window at a time. Then we can restrict the sliding window to a tumbling window, checking for the start condition only if no other window is open.

Example 1

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
```
6.3. Optimisations Made Possible

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.

```agile
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.

Example 2

Context: same as before, but now we’re handling entry gates.

```agile
forseq $w in $seq sliding window
start position $s when $seq[$s]/@action eq "in"
end curItem $cur when $cur/@action eq "out"
and $cur/person eq $seq[$s]/person
return $w[last()]/time - $w[1]/time
```

Used information: sequence consists of `<in>` and `<out>`, unique by combining with person (There exists a key on person). Interleaved, repeating.

Action: Rewrite to tumbling if prefiltering by person is enabled (only possible if query uses only items of one unique sequence, meaning prefiltering is set).

Copy (parts of) end into 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 now that this can be done in the start clause, we should 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.

Example 1

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:

```agile
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: sequence consists of `<in>` and `<out>`, unique by combining with person (There exists a key on person). Interleaved, repeating.

Action: Rewrite to tumbling if prefiltering by person is enabled (only possible if query uses only items of one unique sequence, meaning prefiltering is set).
CHAPTER 6. INTEGRATION INTO PROCESSING MODELS

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

Copy (parts of) WHERE into start/end

To reduce the number of open windows in the interest of resource optimisation, we look for start or
end conditions that are stated in the where clause instead of the start / end clauses.

These conditions can be detected by the fact, that they mostly use the positional selectors [1]
and [last()] on the bound sequence. Clauses using only [1] go into the start, those containing
[last()] are copied to the end clause. If shortcuts are used like position or the -Item keywords,
they are also good indicators for start and end clauses. Clauses using only position or item variables
from the start clause, should be copied there, clauses also or only using position or item variables
from the end clause go to the end clause.

We only copy the conditions into end conditions since moving them could change the semantics.
As example, if we move something from the where into the end condition, the window would (if
open at this moment) produce at the end of the stream. If the same condition is also or only in
the where clause, that would not happen. Therefore, we copy only here and check in a next step
(6.3.1.1) which parts of the where clause we can remove. Parts that go to the start condition can be
moved instead of copied because they are evaluated at window opening and therefore do not change
the semantics.

Example 1

Context: Query testing in the where on start and end elements

    forseq $w in $seq sliding window
    start when fn:true()
    end when fn:true()
    where $w[1]/@action eq 'in' and $w[1]/person eq 'Hans'
        and $w[last()]/@action eq 'out'
        and $w[1]/person eq $w[last()]/person
    return $w

Used information: none

Action: copy start and end conditions into their places. Remove clauses from where that went
into start clause.

    forseq $w in $seq sliding window
    start when $w[1]/@action eq 'in' and $w[1]/person eq 'Hans'
    end when $w[last()]/@action eq 'out' and $w[1]/person eq $w[last()]/person
    where $w[last()]/@action eq 'out' and $w[1]/person eq $w[last()]/person
    return $w
6.3. OPTIMISATIONS MADE POSSIBLE

Remove (parts of) WHERE

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.

Example 1

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

```xml
forseq $w in $seq sliding window
  start curItem $s when $s[@start]
  end curItem $e when $e[@end]
  where count($w[@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 6.3.1.1.

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

Add force

If the where clause only doubles the end condition, that is only to prevent a production of the window at the end of the stream. For this, the additional keyword `force` had been created, so use it and remove the where.

Example 1

Context: Where clause repeats end condition to make sure windows do not get produced on end of stream.

```xml
forseq $w in $seq sliding window
  start curItem $s when $s/@start eq 'true'
  end curItem $e when $e/@end eq 'true'
  where $e/@end eq 'true'
  return $w
```

Used information: none

Action: Since part of or the whole where only makes sure that window does not produce at end of stream, remove these parts and put force in front of end condition:
forseq $w$ in $seq$ sliding window

start curItem $s$ when $s/@start$ eq 'true'
force end curItem $e$ when $e/@end$ eq 'true'
return $w$

Remove force

If force is used but we have an infinite stream, we can remove it. This is because we defined the semantics of streams to only fire an "end of stream" if the stream really ends and not if an error occurs. Since an infinite stream has no end, there is no such event and the force is not needed.

Example 1

With an infinite sequence, streams will never end, so windows only produce when their end condition is satisfied. Therefore, one does not need to use force.

  Context: any where force is used in the query
  Used information: Sequence is infinite.
  Action: Remove force.

Example 2

Another situation where force can be removed even if the sequence is finite, is when sliding or tumbling windows are used and the schema tells us, that it is not possible for the start condition to be satisfied between the last end match and the end of stream and a window can only span one iteration of a sequence. Again, we make use of our adapted semantics, where end of stream can only occur after the (or all, if interleaved) sequence has (have) reached the end of an iteration cycle. The second condition is needed since conditions could span more than one iteration of a sequence, like start when $s[@count]$ eq '1' and end when $e[@count]$ eq '2'. If count stays the same during one iteration, these clauses would span two iterations. But between two iterations an "end of stream" could occur and we would produce a wrong window.

  Context:

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

  Used information: Sequence consists of <start>, some other elements and <end>, repeating, not interleaved. All elements of the sequence are to appear.

  Action: Remove force. No window can be opened between the last one is closed and the end of stream.

6.3.1.2 Operator optimisations

Reduce state

With the same conditions fulfilled as for unblocking (e.g. "if the schema says that values arrive partly sorted, e.g. sorted increasing with respect to digits from third position and higher"), some operators can reduce their state in addition or instead of unblocking. Again, the sort operator works as example, since it may drop all collected information about previous items, as soon as it outputs the generated sequence to unblock. The state so far is no longer needed, since the arriving items are guaranteed to be ordered in after the output items.
Example 1

Context:

```
for $item in $stream
    order by $item/myIntValue
    return $item/$name
```

Used information: $stream is infinite, consists only of one element, is repeating and arrives sorted with respect to its digits of the hundreds column and higher.

Action: Set order operator to reduce state by dropping all stored elements that get output as soon as third digit of myIntValue of arriving item changes with respect to its predecessor.

### 6.3.1.3 Memory optimisations

**Pipelining**

If tumbling windows are used, only one window can be open at a time. Therefore, it should be possible to feed the items inside the window directly to the return expression without materialising. If sliding windows are used it depends on the schema to tell how many windows may be open at a time. If early and parallel production of windows is set and there are only a small number of windows being open, it should be decided to feed the incoming elements directly.

**Specify tuples for early or out-of-the-middle garbage collecting**

When working on streams of large sequences, memorisation of all events could impose memory problems. One possibility to prevent this is to drop elements that will never be read.

Example 1

Context: Query using only the start and the end element of a window, e.g.

```
forseq $w in $seq landmark window
    start curItem $s when $s/@start eq "true"
    end curItem $e when $e/@end eq "true"
    return <res>{$s,$e}</res>
```

Used information: attribute names of all elements of the streamed sequence.

Action: specify to drop immediately all items not having @start or @end eq "true". Therefore, the materialised part of the stream gets very small.

**Set early production of window(s)**

It might save memory to use the information of the arrived element in all open windows and then discard the item, if the following conditions are met:

- parallel window execution is set
- All windows will be closed at some point in time (no end of stream (if any) possible before all open windows meet their end condition)
- Return values of the query can be updated while new elements arrive (aggregates, filters, etc)

This as well allows the classical one-pass stream-query evaluation.
Example 1

Context: Parallel window execution is set. Only updatable operators used in query.

```xml
forseq $w in $seq sliding window
  start curItem $s when $s[self::transaction-start]
  end curItem $e when $e[self::transaction-end] and $e/@id eq $s/@id
  return
  if $e/@type eq commit
    sum($w[self::transaction and ./@id eq $s/@id]/value)
  else
    ()
```

Used information: sequence only consists of `<transaction-start>`, a sequence of `<transaction>` and one `<transaction-end>`, repeated and interleaved, unique by @id, all elements are to appear.

Action: Since end of stream can only occur after a transaction-end, all windows will be closed. Now the trade-off between work for sum and memory has to be made: Keep for each window just a per-element updated sum or materialise the whole stream and sum up all committed windows? Again the schema will help us by telling the maximum number of possible @id values and therefore enabling a guess of the needed memory. If updating sums, evaluate windows early and in parallel and just keep rolling sum.

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.

Example 1

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>`.

```xml
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.
6.3. OPTIMISATIONS MADE POSSIBLE

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 straightforward 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.

6.3.1.4 Scheduling / planning / threading possibilities

Set stream-prefiltering (splitting windows)

This is a first step to parallel windows. We prefilter streams into several disjoint streams. This is only useful if we now process the streams in parallel. If not, we could just have used overlapping windows.

Example 1

Context: Query asks things about single groups of elements, not touching the others.
CHAPTER 6. INTEGRATION INTO PROCESSING MODELS

forseq $w$ in $\$seq$ landmark window
start curItem $s$ when $s/@type eq "start"
end curItem $e$ when $e/@type eq "end" and $e/@id eq s/@id
return $w[@id eq $w[1]/@id]/myelement

Used information: We have interleaving sequences.
Action: surround query with distinct-values:

for $id$ in distinct-values($seq)
return
  forseq $w$ in $\$seq[/@id eq $id]$ landmark window
  start curItem $s$ when $s/@type eq "start"
  end curItem $e$ when $e/@type eq "end"
  return $w/myelement

Set parallel window execution

From the schema we can read how many windows may be open approximately at the same time. If we know that we can handle them all in parallel, we should do so. So this is again a trade-off between available resources and fast results. Parallel windows are very interesting where we can use early-production and where we have prefiltering enabled.

Set number of threads

Again this number will be set according to available resources and assumed open windows. It is not the same as setting parallel window execution, since here we are only setting the number of threads to be available, like telling the system "you can use parallel windows, but not more than this amount". We define the maximal number of threads needed as the maximal number of windows that may be open/evaluated in parallel.

The number of possible parallel windows is defined as follows:

- If the selection of windows is based on a value which at the same time is the key of interleaving sequences, the variety of this key - if given - defines the number of parallel windows.

6.4 More on integration into processing models

As discussed in the section on "validating infinite streams", there are many open questions regarding the validation of infinite streams. Future work should also focus on this topic to clarify how infinite streams are validated, what are the consequences of a failed validation as well as what a successful infinite stream validation tells about the stream.

The lists in the above two sections about possible semantic changes and optimisations do not claim to be exhaustive. They are a collection of what is possible, but there will be more possibilities by combining them and/or analysing the processing models as well as the implemented stream description in more detail. The above listed possibilities are the ones being mainly useful, they are enabling optimisations, allowing to define more complex combinations on top of them. Future work on the integration of stream description into processing models could go deeper here and search for more optimisations as well as semantic adaptations.
Although this report is focusing on XQueryP, where no time is defined, information about dynamics might be interesting for other processing models, like e.g. Aurora [27]. Therefore, we peek shortly into the descriptions on dynamics, where they can be used and how they help. A deeper analysis of dynamics in stream processing could be target of future work. To allow itself to decide best about load shedding and QoS-calculations, Aurora bases completely on time information. Therefore, Aurora might gain some more optimisations or semantic adaptations if there would be some information about timing. The authors of Aurora state, that if they "know the expected rate of tuple production r(d) from each data source" [27, Section 4.4], they could use some static analysis "to ascertain if Aurora is sized correctly" [27, Section 4.4].

This is another information that could be given by stream descriptions, if it is known in advance. One could extend the schema description to give information about the following rates:

- repeating rate - After what time the sequence repeats itself?
- interleaving rate - How long does it take between two starting elements of interleaving instances of a sequence?
- element rate - After what time an element will arrive after its predecessor? This rate could be given in two flavours:
  - for the sequence, like a general measurement for all elements inside
  - in each element, denoting the time between the appearance of the predecessor and itself.
- In case of Aurora, one might try to inform about a priori tuple delay knowledge.

Note, that there are some different interests in rates: Some applications may need the minimal, some the maximal and others the average rates. Therefore, an extension should enable the indication of all three variants per rate.

Another issue are conflicts between given rates:

- If element rate is given for the sequence and for some or even all individual elements, there has to be a clear conflict resolution strategy. For example, one can decide to consider the individual element rate if given, else the general one, or nothing if none given.
- If multiple streams are aggregated into one, where should rates be stated? In the combination or in the individual stream? If allowed in both, which one has more weight? The interleave
rate of the choice will have some impact on the interleave rate of the contained sequences and others. There too, a clear conflict resolution strategy must be defined.

7.1 Use case examples

In the following we give some use cases to show where rate information could be used:

Entry Gates

If there are many people entering and leaving the building, the decision to prefilter or keep parallel windows may be assisted by some other information, like interval rates. If the system could read from the schema that people will enter in the morning and leave in the evening, therefore most or all windows are usually open for one whole day and the query only wants to calculate working times, the system may prefilter to keep single windows. If on the other hand the query is about telling every entering person who else has entered in the last 30 minutes, sliding windows are used and prefiltering impossible, since information about other persons is used. If it is a building where people enter and leave every hour repeatedly, like e.g. supporters going to their clients, sliding windows may be chosen, since they are not open for long time.

Another decision is also made possible by rate information: If windows are open for a short time only and very time varying, so that not many will be intersecting, materialising will not be a problem and can be done without further information. This is because the materialised events can be garbage collected as soon as the last window using it is closed. Since they are open only for short times, this will happen in short intervals. The same applies if people are inside the building the whole day, the stream of events depends only on the number of persons, since for each of them only two events are generated. Therefore, materialisation may again not be a problem if there is a limited number of persons.

RFID

Using rate information and thereof knowing how long a product is in the line, we know, that if the query is about handling individual products only, materialisation is not a problem. This is because we can let the events concerning each product be garbage collected after having handled its part of the query. On the other hand, if we know that we will have to materialise everything and a window from the query begins with the start and ends with the stop of the transport band, we may decide about memory problems.

Another decision we could make, if we know the number of bands or products on a band at the same time, is about threading: If the query only looks at items of one line but needs to be run over all lines at the same time, we can prefilter the stream into individual streams per line. If we then see that there are many products on the same line at a time and a query only looks at events of individual products, we could even handle the windows for each product in parallel. Since we know the number of lines and products in advance, we can estimate our needs regarding thread pool and memory and warn about or even abort a query that seems to be too heavy.

7.2 Optimisation example

If the information on streams also contains some rates, as how fast events arrive or how many windows may be started in parallel, we can calculate how long a window will be opened. From this information we may derive or plan e.g. how many threads should be available in the pool, how many elements will need to be memorised and for how long. That is how we can preallocate threads and/or space or even abort queries made impossible by their requirements:
7.2. OPTIMISATION EXAMPLE

- If not depending on the key constraint or the schema does not define a variety for it, the number of possible parallel windows can be computed with the rates and number of repetitions (not repeating -> *1, repeating=infinitive -> ∞):

\[
\text{interleaveRate/\text{max}} \sum_{\text{elementRate/\text{max}}} \ast \text{repeating} = \text{minimal number of parallel windows}
\]

\[
\text{interleaveRate/\text{min}} \sum_{\text{elementRate/\text{min}}} \ast \text{repeating} = \text{maximal number of parallel windows}
\]

Since our main target, XQueryP, has no timing information and no time semantics, we will not go further on this interesting topic but leave it open for future work on stream annotation and/or schema extension.
Chapter 8

Semantic analysis of FORSEQ

In this chapter we will analyse the static infinity semantics of FORSEQ in combination with the XML Schema extensions proposed in Chapter 5. We do this to prove the claimed advantages of the schema extension. The analysis is done the same way as it is made in [2]. For details, please refer to this document. As stated in Section 6.4, the possible optimisations and semantic adaptation lists are not exhaustive. Therefore, one will be able to find more ways to make certain FORSEQ constructs finite. If future work is made in searching further optimisations and semantic adaptations, there is the possibility to find more rules to describe the static infinity of FORSEQ too. Furthermore, future work could include doing a complete analysis of FORSEQ, including static side-effect analysis and analysis of dynamic evaluation.

8.1 Extended Basic Concepts

We have extended the lists which are used in [2] to describe elements used in the rules:

- *dynEnv*
  
  - *varValue* can now get not only a variable name to resolve to the value of the variable but instead the path to a field of the element behind the variable:
    
    \[\text{varValue} \left( \text{itemVar.field1} \Rightarrow x \right) \text{ tells that field1 of the element stored in itemVar has value } x.\]

  - *firstVal* is used to extract the value assigned to an element by a next rule using *first*:
    
    If there is a rule \( \text{next(fields, (first = 2)), firstVal(next(fields, (first = 2)))} \) returns '2'.

  - *varItemName* is used to extract the name of the element or attribute bound to the given variable. E.g. after the statement \( \text{let } \$\text{var} := \text{doc('this.xml')/author} \) we could ask the dynEnv about the name of the element bound to \$\text{var} by \( \text{varItemName} \left( \text{varVar} \right) \) if \( \text{statEnv} \vdash \text{VarName of } \text{var expands to } \text{varVar} \). As return value of our varItemName-call we would get the string "author".

8.2 Analysis type

We will do the same static infinity analysis as in [2]:

Rules are defined to detect possible errors for FORSEQ when infinite sequences are used as input. An exception is thrown if an error is detected during static analysis and a warning is shown if an error might happen during execution if this error is not decidable during static analysis. The
CHAPTER 8. SEMANTIC ANALYSIS OF FORSEQ

Problem of infinite input sequences is that if they are fed to a blocking operator, this operator will run forever. We will try to show, that we can reduce the number of such infinite runs by using our extension to decide better if a stopping evaluation is possible or not. All expressions get an annotation to show their infinity status:

- infinite: The result is an infinite stream, shown by $\text{statEnv} \vdash \text{Expr} \triangleright \infty$
- undecidable: The result may be an infinite or finite stream, static information is not enough to decide, shown by $\text{statEnv} \vdash \text{Expr} \triangleright \nu$
- finite: The result is a finite sequence, shown by $\text{statEnv} \vdash \text{Expr} \triangleright \wp$

If an error is detected, this is shown by $\text{statEnv} \vdash \text{Expr} \triangleright \text{StaticException}$.

To store the infinity status of an expression in a variable ($\text{var}_\infty$), we use $\text{statEnv} \vdash \text{Expr} \triangleright \text{var}_\infty$.

8.3 FORSEQ

8.3.1 Normalisation

Before any analysis, standard normalisation rules [9] are applied, without the one for where clauses; any where clause will stay untouched. This leads us to:

$$\begin{align*}
\text{forseq } & \$\text{seq in } \text{expr} \text{stream type } x \text{ window} \\
& \text{start position } s\text{pos cur Item } s\text{cur next Item } s\text{next prev Item } s\text{prev when expr}_{\text{start}} \\
& \text{end position } e\text{pos cur Item } e\text{cur next Item } e\text{next prev Item } e\text{prev when expr}_{\text{end}} \\
& \text{where expr}_{\text{where}} \\
& \text{return expr}_{\text{return}}
\end{align*}$$

Sometimes we will have an order by clause in the FORSEQ:

$$\begin{align*}
\text{forseq } & \$\text{seq in } \text{expr} \text{stream type } x \text{ window} \\
& \text{start position } s\text{pos cur Item } s\text{cur next Item } s\text{next prev Item } s\text{prev when expr}_{\text{start}} \\
& \text{end position } e\text{pos cur Item } e\text{cur next Item } e\text{next prev Item } e\text{prev when expr}_{\text{end}} \\
& \text{where expr}_{\text{where}} \\
& \text{order by field } y \\
& \text{return expr}_{\text{return}}
\end{align*}$$

8.3.2 Static infinity analysis

Since they are used as conditions, $\text{expr}_{\text{start}}$, $\text{expr}_{\text{end}}$ and $\text{expr}_{\text{where}}$ have to be finite in both cases, without and with the schema extension:

$$\begin{align*}
\text{statEnv} \vdash \text{Expr}_{\text{start}} \triangleright \lnot \wp \\
\text{statEnv} \vdash \text{forseq } \$\text{seq in } \ldots \text{return expr}_{\text{return}} \triangleright \text{StaticException}
\end{align*}$$

$$\begin{align*}
\text{statEnv} \vdash \text{Expr}_{\text{end}} \triangleright \lnot \wp \\
\text{statEnv} \vdash \text{forseq } \$\text{seq in } \ldots \text{return expr}_{\text{return}} \triangleright \text{StaticException}
\end{align*}$$

$$\begin{align*}
\text{statEnv} \vdash \text{Expr}_{\text{where}} \triangleright \lnot \wp \\
\text{statEnv} \vdash \text{forseq } \$\text{seq in } \ldots \text{return expr}_{\text{return}} \triangleright \text{StaticException}
\end{align*}$$
8.3. FORSEQ

8.3.2.1 Without schema extension

The main problem we have with FORSEQ is that without stream knowledge there is the same
infinity decision as with FOR.

- \( expr_{\text{return}} \triangleright \infty \): The query only produces results for the first window to be evaluated because
  the return expression runs forever. Therefore, the query does not terminate and the user only
  gets results for the window closed first.

\[
\begin{align*}
\text{statEnv} \vdash & \text{seq of var expands to } \text{VarSeq} \\
\text{statEnv} + \text{varInf}(\text{VarSeq} \triangleright \varphi) \vdash Expr_{\text{return}} \triangleright \infty; \\
\text{Warning (The Return Expression blocks the Iteration)}
\end{align*}
\]

- \( expr_{\text{stream}} \triangleright \infty \): The query never stops reading because the stream is infinite. Therefore, the
  query does not terminate but the user sees all results so far if \( expr_{\text{return}} \) is finite. If \( expr_{\text{return}} \)
  is undecidable, a warning is shown that the iteration might be blocked.

\[
\begin{align*}
\text{statEnv} \vdash & \text{Expr}_{\text{stream}} \triangleright \infty \\
\text{statEnv} \vdash & \text{seq of var expands to } \text{VarSeq} \\
\text{statEnv} + \text{varInf}(\text{VarSeq} \triangleright \varphi) \vdash Expr_{\text{return}} \triangleright \varphi; \\
\text{Warning (The Return Expression might block the Iteration)}
\end{align*}
\]

- If both expressions are undecidable, FORSEQ is undecidable and a warning is shown about
  the execution being possibly blocked.

\[
\begin{align*}
\text{statEnv} \vdash & \text{Expr}_{\text{stream}} \triangleright \varphi \\
\text{statEnv} \vdash & \text{seq of var expands to } \text{VarSeq} \\
\text{statEnv} + \text{varInf}(\text{VarSeq} \triangleright \varphi) \vdash Expr_{\text{return}} \triangleright \varphi; \\
\text{Warning (The Return Expression might block the Iteration)}
\end{align*}
\]

- If \( expr_{\text{stream}} \) is finite but \( expr_{\text{return}} \) is undecidable, FORSEQ is undecidable and the user will
  get a warning about the execution possibly being blocked.

\[
\begin{align*}
\text{statEnv} \vdash & \text{Expr}_{\text{stream}} \triangleright \varphi \\
\text{statEnv} \vdash & \text{seq of var expands to } \text{VarSeq} \\
\text{statEnv} + \text{varInf}(\text{VarSeq} \triangleright \varphi) \vdash Expr_{\text{return}} \triangleright \varphi; \\
\text{Warning (The Return Expression might block the Iteration)}
\end{align*}
\]

- If \( expr_{\text{return}} \) is finite the infinity status of FORSEQ is determined by the status of \( expr_{\text{stream}} \).
In all other cases FORSEQ will terminate. Note that dependent on the window types (or worst, in case of general FORSEQ) there might be a huge number of windows and therefore the execution might take quite a while.

8.3.2.2 With schema extension

By adding stream descriptions and applying the options and semantic adaptations shown in Sections 6.3.1 and 6.2.1 one is able to filter out some cases where FORSEQ terminates even if \( \text{expr}_{\text{stream}} \) is infinite. To facilitate the following formulae we introduce some predicates on the various expressions as well as some functions to extract information from XML schema.

- \( \text{predPosLimit}(\text{expr}, \text{posVar}) \) checks if the given expression \( \text{expr} \) is limited by the given position variable \( \text{posVar} \). With limited we mean from a certain value of \( \text{posVar} \) on \( \text{expr} \) will always return false. Note that this predicate uses \( \text{dynEnv} \) to evaluate since it needs variable values. An example for such a limited expression is \( \text{end position} \: \text{sep} \: \text{pos} \: \text{when} \: \text{sep} \: \text{pos} \: \leq 23 \), which will return false for every value for \( \text{sep} \: \text{pos} \) greater than 23. The existence quantifier in the second line is needed to tell that there may be additional constraints in the expressions but it can evaluate to true, whereas the all quantifier tells that when the position limit is reached the other parts of the expression cannot reduce the limiting function of the position (no way to get \( \text{expr} \) to true after the limit). An example for a non-limited expression is \( \text{end curItem} \: \text{sec} \: \text{ur} \: \text{pos} \: \text{when} \: \text{sec} \: \text{ur} \: \text{pos} \: < 23 \lor \text{sec} \: \text{ur} / \text{name} \: \text{eq} \: \text{Miller}^1 \).

\[
\text{predPosLimit}(\text{expr}, \text{posVar}) = \begin{cases} 
\exists_1 : ((\forall_2 \text{dynEnv} : g > i, \text{dynEnv} \: \text{varValue} (\text{posVar} \Rightarrow i, \text{posVar} \Rightarrow g) \vdash \text{expr} \Rightarrow \text{false}) \land \\
(\exists_3 \text{dynEnv} : \text{dynEnv} \: \text{varValue} (\text{posVar} \Rightarrow i) \vdash \text{expr} \Rightarrow \text{true}) \Rightarrow \text{true} \\
\text{else} : \Rightarrow \text{false}
\end{cases}
\]

- \( \text{predRelPosDep}(\text{expr}, \text{posVar}1, \text{posVar}2) \) checks if the given expression \( \text{expr} \) is limited by a dependency between the given two position variables \( \text{posVar}1 \) and \( \text{posVar}2 \). With limited we mean for every value of \( \text{posVar}1 \) there is a value for \( \text{posVar}2 \) from which on \( \text{expr} \) will always evaluate to false. An example for such a limited expression is the end expression in the following cutout from a FORSEQ: \( \text{end position} \: \text{sep} \: \text{pos} \: \text{when} \: \text{sep} \: \text{pos} + 23 \geq \text{sep} \: \text{pos} \). It will return false for every value of \( \text{sep} \: \text{pos} \) being greater than \( \text{sep} \: \text{pos} + 23 \). The quantifiers are used the same way as in \( \text{predPosLimit} \).

\[
\text{predRelPosDep}(\text{expr}, \text{posVar}1, \text{posVar}2) = \begin{cases} 
\forall_1 : \exists_2 : ((\forall_3 \text{dynEnv} : g > p_1, \text{dynEnv} \: \text{varValue} (\text{posVar}1 \Rightarrow p_1, \text{posVar}2 \Rightarrow g) \vdash \text{expr} \Rightarrow \text{false}) \land \\
(\exists_3 \text{dynEnv} : \text{dynEnv} \: \text{varValue} (\text{posVar}1 \Rightarrow p_1, \text{posVar}2 \Rightarrow p_2) \vdash \text{expr} \Rightarrow \text{true}) \Rightarrow \text{true} \\
\text{else} : \Rightarrow \text{false}
\end{cases}
\]

- \( \text{getFields}(\text{expr}, \text{itemVar}) \) returns a list of every xpath occurring in \( \text{expr} \) starting with the unexpanded name of \( \text{itemVar} \). Before returning this list, it cuts away the start element (the \( \text{itemVar} \)) including the following '/'. If after this removal the path is empty, it is replaced by '/'.
getEleDefs(schema,fields) returns all element definition children of all streamSequence or streamChoice definitions from schema which have the given fields. fields is a list of relative XPaths to fields (elements or attributes). getEleDefs tries for every xpath if itself or a subpath of it is a legal xpath inside each element definition children of all streamSequence or streamChoice definitions of schema and returns all element definitions which fulfill this test.

matches(selector,eleDef) checks if selector would select any instance of the element definition eleDef. In the case of next rules this means eleDef has all fields required by the next element’s selector attribute. fitsDef(eleDef,element) returns true if and only if element is an instance of eleDef.

getSchemaFieldNextRules(schema,eleDef,field) returns all next rules defined on the given field field of the given element definition eleDef in the given XML Schema schema. It does so by checking if there is a next rule in schema whose fields part is fulfilled by the eleDef and if field figures in the rules. If schema has rules using field it returns them as a list of rules. A rule uses field if one of its xpath attributes ends by the fieldname, therefore the substring after the fieldname of this xpath attribute value is the empty string.

getSchemaFieldKeys(schema,field) returns all streamConstraints of type key of the schema schema defined only on the given field field.

predValLimit(expr,itemVar,op,field) checks if the given expression expr is limited by the value of field of the item stored in variable itemVar compared against some defined value with the comparator op. An example for such a limited expression where the op is ‘<’, the itemVar is ’ecur’, and the field is ’step’, is end curItem $ecur when $ecur/step < 23. The field might also be the item itself by just using the variable name without any path addition, given as parameter by the value ‘.’. If such a constraint exists, expr will evaluate to false forever after some point in the stream since the next rule does not allow the field value to become valid again.

predValLimit(expr,itemVar,op,field) =

\[
\exists_i : \left( \forall_{\text{dynEnv},g \text{op} i} : \text{dynEnv} + \text{varValue}(\text{itemVar}.\text{field} \Rightarrow g) \vdash \text{expr} \Rightarrow \text{false} \right) \land \\
(\exists_{\text{dynEnv}} : \text{dynEnv} + \text{varValue}(\text{itemVar}.\text{field} \Rightarrow i) \vdash \text{expr} \Rightarrow \text{true}) \Rightarrow \text{true}
\]

else \Rightarrow \text{false}

predNextBreaksValLimit(eleDef,op,field,schema) checks if there is a next rule defined in the schema schema using the field field of the element definition eleDef that guarantees that the next instance will satisfy the comparison operation op against the current instance. usesOnly(rule,field) asserts that rule only compares field. nextOpCurrent(rule,op) asserts that the rule constrains next to fulfill op against current when written like the function name (“next op current”). E.g. nextOpCurrent(’next > current’,’>’) as well as nextOpCurrent(’current < next’,’<’) return true.

predNextBreaksValLimit(eleDef,op,field,schema) =

\[
\exists_{\text{rule} \in \text{rules}} : \left( \text{usesOnly(rule, field)} \land \text{nextOpCurrent(rule, op)} \Rightarrow \text{true} \right) \\
\text{else} \Rightarrow \text{false}
\]
• predValLimitBreakable(expr, schema, itemVar) checks if the given expression \( expr \) is looking for a defined value of a field of the item stored in variable \( itemVar \) and if this field is constrained by a next rule in the given XML schema \( schema \). An example for such a limited expression is \( end\ cur\ Item\ $Secur\ when\ $Secur/step < 23 \) and there exists the rule \( next((step), (next.step > current.step)) \). The value of the field step will increase only and therefore at some point in the stream it will be greater than 23. If there exists such a limit and it is attacked by a next rule, the result is true, otherwise false.

\[
pred\ Val\ Limit\ Breakable(expr, schema, itemVar) = \\
\{ \\
\begin{align*}
& fields = getFields(expr, itemVar); \\
& eleDefs = getEleDefs(schema, fields); \\
& \exists eleDef \in eleDefs, field \in fields, op \in \{<, \leq, =, \geq, >\} : (pred\ Val\ Limit(expr.itemVar, op, field) \land \\
& \quad pred\ Next\ Breaks\ Val\ Limit(eleDef, op, field, schema)) \Rightarrow true \\
& \text{else} \Rightarrow false
\end{align*}
\]

• predFirstOverLimit(eleDef, op, field, schema) checks if the given expression \( expr \) is limited by the value of \( field \) of the item stored in variable \( itemVar \) compared against some defined value with the comparator \( op \) and if this limit is not reached by the first element of a stream because of a next rule constraining the first element. An example for such a limited expression is \( end\ cur\ Item\ $Secur\ when\ $Secur/step < 23 \) and there exists the rule \( next((step), (first.step = 50)) \). If such a constraint exists \( expr \) will evaluate to false when using the value of the rule.

\[
pred\ First\ Over\ Limit(eleDef, op, field, schema) = \\
\{ \\
\begin{align*}
& rules = getSchemaFieldNextRules(schema, eleDef, field); \\
& \exists rule \in rules : (isFirst(rule) \land \\
& \quad \forall dynEnv : dynEnv + varValue(itemVar.field \Rightarrow firstVal(rule)) \setminus expr \Rightarrow false \land \\
& \quad pred\ Val\ Limit(expr, itemVar, op, field) \Rightarrow true \\
& \text{else} \Rightarrow false
\end{align*}
\]

• predValLimitUnreachable(expr, schema, itemVar) checks if the given expression \( expr \) is looking for a defined value of a field of the item stored in variable \( itemVar \) and if this field is constrained by a next rule in the given XML schema \( schema \) in a way it can never be fulfilled. An example for such a limited expression is \( end\ cur\ Item\ $Secur\ when\ $Secur/step < 23 \) and there exists the rule \( next((step), (first.step = 50, next.step > current.step)) \). The value of the field step of the first arriving item will be 50 and the later arriving items’ step values will increase only. Therefore, the step value will never be smaller than 23 and no single window will be evaluated. If there exists such a limit and it is made unreachable by a next rule on next values and on the first of the limited field, the result is true, otherwise false.

\[
pred\ Val\ Limit\ Unreachable(expr, schema, itemVar) = \\
\{ \\
\begin{align*}
& fields = getFields(expr, itemVar); \\
& eleDefs = getEleDefs(schema, fields); \\
& \exists eleDef \in eleDefs, field \in fields, op \in \{<, \leq, =, \geq, >\} : (pred\ First\ Over\ Limit(eleDef, op, field, schema) \land \\
& \quad pred\ Next\ Breaks\ Val\ Limit(eleDef, op, field, schema)) \Rightarrow true \\
& \text{else} \Rightarrow false
\end{align*}
\]

• predValEquality(expr, itemVar1, itemVar2, field) checks if the expression \( expr \) is only evaluating to true if both \( itemVars \) have the same value in field \( field \).

\[
pred\ Val\ Equality(expr, itemVar1, itemVar2, field) = \\
\]
8.3. FORSEQ

∀val : ((∀dynEnv,val2≠val : dynEnv + varValue(itemVar1.field ⇒ val, itemVar2.field ⇒ val2) ⊢ expr ⇒ false)∧
(∃dynEnv : dynEnv + varValue(itemVar1.field ⇒ val, itemVar2.field ⇒ val) ⊢ expr ⇒ true)) ⇒ true
else := false

• predFieldEqualityIsKey(expr, itemVar1, itemVar2, schema) checks if the expression expr is only evaluating to true if both itemVars have the same value in some field and if this field is defined as a key in schema schema.

predFieldEqualityIsKey(expr, itemVar1, itemVar2, schema) =
{∃field∈getFields(expr,itemVar1) : (predValEquality(expr, itemVar1, itemVar2, field)∧
getSchemaFieldKeys(schema, field)) ⇒ true
else ⇒ false

• schemaItemMustOccur(name, schema) checks if the schema guarantees that an element with name name will occur. This is done by looking if from any element with this name itself and all its parents up to the streamSequence element have minOccurs not 0. The following XQuery counts for every element with the given name the occurrence of a minOccurs='0' in its parents. If this number is 0 it returns '1' for that element, else nothing. Therefore, if we count some '1' for all elements with the given name, we have some path which must occur.

schemaItemMustOccur(name, schema) =
count(for $ele in $schema//streamSequence//$name
   return if(count($schema//streamSequence//*[.//$ele and @minOccurs='0'])>0)
   then()
   else '1'
) > 0

• predItemOccurrence(expr, itemVar, name) checks if the expression expr is only evaluating to true if the item in value itemVar has certain name and if the schema guarantees that an element with this name will occur.

predItemOccurrence(expr, itemVar, name) =
{∀dynEnv,name2≠name : (dynEnv + varItemName(itemVar ⇒ name2) ⊢ expr ⇒ false)∧
∃dynEnv : (dynEnv + varItemName(itemVar ⇒ name) ⊢ expr ⇒ true)) ⇒ true
else := false

• predItemMustOccur(expr, itemVar, schema) checks if the expression expr is only evaluating to true if the item stored in itemVar has certain name and if the schema guarantees that an element with this name will occur.

predItemMustOccur(expr, itemVar, schema) =
{∃name : (predItemOccurrence(expr, itemVar, name)∧
schemaItemMustOccur(name, schema)) ⇒ true
else := false

With the predicates from above the following cases can be filtered out to be analysed in more detail. In each rule where we want to check if an expression is looking for a certain value in expr_end, we check only for curItem by e.g. VarEcurs. It is important that all these rules are also applicable for VarEprev and VarEnext, standing for the previous or the next item respectively (prevItem and nextItem in the end expression). We omitted these rules since they facilitate the same cases only with other expressions. The same fits for expr_start where one could also check for VarSprev and VarSnext.
• If \( \text{expr}_\text{end} \) or \( \text{expr}_\text{where} \) look for an absolute position of the last element of a window, the engine can declare an early stop point after this position since there will be no more matches of the conditions. Therefore, the query infinity depends only on \( \text{expr}_\text{return} \):

\[
\text{statEnv} \vdash \text{Expr}_\text{stream} \triangleright \infty \\
\text{statEnv} \vdash \text{seq of var expands to VarSeq} \\
\text{statEnv} \vdash \text{epos of var expands to VarEpos} \\
\text{predPosLimit(} \text{expr}_\text{end}, \text{VarEpos)} \lor \text{predPosLimit(} \text{expr}_\text{where}, \text{VarEpos)}
\]

\[
\text{statEnv} + \text{varInf(VarSeq \triangleright \varphi, VarEpos \triangleright \varphi)} \vdash \text{Expr}_\text{return} \triangleright \text{var}_\infty \_\text{return}
\]

\[
\text{statEnv} \vdash \text{forseq}$\text{seq in ... return Expr}_\text{return} \triangleright \text{var}_\infty \_\text{return}
\]

• If \( \text{expr}_\text{end}, \text{expr}_\text{where} \) or \( \text{expr}_\text{start} \) look for an absolute value of a field of the first or last element of a window and the schema tells that this field is ordered, the engine stops if the schema tells us the first value to expect is constrained in a way that the value of one of the expressions is not reachable. Therefore, the query is finite:

\[
\text{statEnv} \vdash \text{Expr}_\text{stream} \triangleright \infty \\
\text{statEnv} \vdash \text{seq of var expands to VarSeq} \\
\text{statEnv} \vdash \text{scur of var expands to VarScur} \\
\text{statEnv} \vdash \text{ecur of var expands to VarEcur} \\
(\text{predV alLimitUnreachable(} \text{expr}_\text{start}, \text{schema, VarScur)} \lor \\
\text{predV alLimitUnreachable(} \text{expr}_\text{end}, \text{schema, VarScur)} \lor \\
\text{predV alLimitUnreachable(} \text{expr}_\text{where}, \text{schema, VarScur)} \lor \\
\text{predV alLimitUnreachable(} \text{expr}_\text{where}, \text{schema, VarEcur)}
\]

\[
\text{statEnv} + \text{varInf(VarSeq \triangleright \varphi, VarScur \triangleright \varphi, VarEcur \triangleright \varphi)} \vdash \text{Expr}_\text{return} \triangleright \text{var}_\infty \_\text{return}
\]

\[
\text{statEnv} \vdash \text{forseq}$\text{seq in ... return Expr}_\text{return} \triangleright \varphi
\]

• If \( \text{expr}_\text{end} \) or \( \text{expr}_\text{where} \) look for an absolute value of a field of the last element of a window and the schema tells that this field is ordered, the engine can declare an early stop point as soon as the value \( \text{expr}_\text{end} \) or \( \text{expr}_\text{where} \) is looking for is not reachable anymore. Therefore, the query infinity depends only on \( \text{expr}_\text{return} \). It might even be that the engine stops the query before the first window is evaluated because the value the one of the expressions is looking for is unreachable. In this case the query would terminate. But since that is not known before runtime we ignore this fact:

\[
\text{statEnv} \vdash \text{Expr}_\text{stream} \triangleright \infty \\
\text{statEnv} \vdash \text{seq of var expands to VarSeq} \\
\text{statEnv} \vdash \text{ecur of var expands to VarEcur} \\
(\text{predV alLimitUnbreakable(} \text{expr}_\text{end}, \text{schema, VarEcur)} \lor \\
\text{predV alLimitUnbreakable(} \text{expr}_\text{end}, \text{schema, VarEcur)} \lor \\
\text{predV alLimitUnbreakable(} \text{expr}_\text{where}, \text{schema, VarEcur)} \lor \\
\text{predV alLimitUnbreakable(} \text{expr}_\text{where}, \text{schema, VarEcur})
\]

\[
\text{statEnv} + \text{varInf(VarSeq \triangleright \varphi, VarEcur \triangleright \varphi)} \vdash \text{Expr}_\text{return} \triangleright \text{var}_\infty \_\text{return}
\]

\[
\text{statEnv} \vdash \text{forseq}$\text{seq in ... return Expr}_\text{return} \triangleright \text{var}_\infty \_\text{return}
\]

• If \( \text{expr}_\text{start} \) looks for an absolute position of the first element of a window and \( \text{expr}_\text{end} \) looks for a position relative to the start position, the engine can declare an early stop point after the last possible end match, since the number of start matches is limited and therefore the possible end matches are limited too. Therefore, the query infinity depends only on \( \text{expr}_\text{return} \):
\[
\begin{align*}
\text{statEnv} \vdash Expr_{\text{stream}} \triangleright \infty \\
\text{statEnv} \vdash \text{seq of var expands to VarSeq} \\
\text{statEnv} \vdash \text{spos of var expands to VarSpos} \\
\text{statEnv} \vdash \text{epos of var expands to VarEpos} \\
predPosLimit(\text{expr}_{\text{start}}, \text{VarSpos}) \\
predRelPosDep(\text{expr}_{\text{end}}, \text{VarSpos}, \text{VarEpos}) \\
\text{statEnv} + \text{varInf}(\text{VarSeq} \triangleright \varnothing, \text{VarSpos} \triangleright \varnothing, \text{VarEpos} \triangleright \varnothing) \vdash \text{expr}_{\text{return}} \triangleright \text{var}_{\infty_{\text{return}}} \\
\text{statEnv} \vdash \text{forseq $\$\text{seq in ... return} \text{Expr}_{\text{return}} \triangleright \text{var}_{\infty_{\text{return}}}$}
\end{align*}
\]

- If \text{expr}_{\text{start}} looks for an absolute value of a field of the first element of a window and the schema tells that this field is ordered and \text{expr}_{\text{end}} looks for a position relative to the start position, the engine can declare an early stop point after the last possible end match as soon as the value \text{expr}_{\text{start}} is looking for is not reachable anymore since the number of start matches and therefore also the possible end matches are limited. Therefore, the query infinity depends only on \text{expr}_{\text{return}}. Because of the size of this formula it is shown in sideways in Figure 8.1.

- If the window type \text{type}_x is not landmark: If the number of matches of \text{expr}_{\text{start}} is limited by one of the above two cases but \text{expr}_{\text{end}} only looks for some field to have the same content as the corresponding field in the start element and the schema specifies a key on this field and shows that the end element must appear at least once per sequence iteration, we know that \text{expr}_{\text{end}} will match for every start match. Since we have no landmark window, an end match implies the removal of the window and therefore the query infinity depends only on \text{expr}_{\text{return}}. Because of the size of this formula it is shown in sideways in Figure 8.2.

- If \text{expr}_{\text{end}} only checks for the occurrence of a certain item and this item is guaranteed to occur by the schema, windows will be closed for sure. Again, if we have some limited number of starts of windows the query infinity depends only on \text{expr}_{\text{return}}. Because of the size of this formula it is shown in sideways in Figure 8.3.
\[
\text{statEnv} \vdash \text{Expr}_{\text{stream}} \triangleright \infty \\
\text{statEnv} \vdash \text{seq} \text{ of } \text{var} \text{ expands to } \text{VarSeq} \\
\text{statEnv} \vdash \text{scur of } \text{var} \text{ expands to } \text{VarScur} \\
\text{statEnv} \vdash \text{spos of } \text{var} \text{ expands to } \text{VarSpos} \\
\text{statEnv} \vdash \text{epos of } \text{var} \text{ expands to } \text{VarEpos} \\
predValLimitBreakable(\text{expr}_{\text{start}}, \text{schema}, \text{VarScur}) \\
predRelPosDep(\text{expr}_{\text{end}}, \text{VarSpos}, \text{VarEpos}) \\
\text{statEnv} + \text{varInf} \left( \text{VarSeq} \triangleright \varphi, \text{VarScur} \triangleright \varphi, \text{VarSpos} \triangleright \varphi, \text{VarEpos} \triangleright \varphi \right) \vdash \text{Expr}_{\text{return}} \triangleright \varphi_{\infty, \text{return}}
\]

Figure 8.1: Number of start matches is limited, end condition looks for position relative to start position

\[
\text{statEnv} \vdash \text{type} \neq \text{landmark} \\
\text{statEnv} \vdash \text{Expr}_{\text{stream}} \triangleright \infty \\
\text{statEnv} \vdash \text{seq} \text{ of } \text{var} \text{ expands to } \text{VarSeq} \\
\text{statEnv} \vdash \text{spos of } \text{var} \text{ expands to } \text{VarSpos} \\
\text{statEnv} \vdash \text{scur of } \text{var} \text{ expands to } \text{VarScur} \\
\text{statEnv} \vdash \text{ecur of } \text{var} \text{ expands to } \text{VarEcur} \\
predPosLimit(\text{expr}_{\text{start}}, \text{varSpos}) \lor \text{predValLimitBreakable}(\text{expr}_{\text{start}}, \text{schema}, \text{VarScur}) \\
predFieldEqualityIsKey(\text{expr}_{\text{end}}, \text{VarScur}, \text{VarEcur}, \text{schema}) \\
predItemMustOccur(\text{expr}_{\text{end}}, \text{itemVar}, \text{schema}) \\
\text{statEnv} + \text{varInf} \left( \text{VarSeq} \triangleright \varphi, \text{VarSpos} \triangleright \varphi, \text{VarScur} \triangleright \varphi, \text{VarEcur} \triangleright \varphi \right) \vdash \text{Expr}_{\text{return}} \triangleright \varphi_{\infty, \text{return}}
\]

Figure 8.2: Number of start matches is limited, end condition looks for value constrained by key
\begin{align*}
\text{statEnv} \vdash Expr_{\text{stream}} &\succ \infty \\
\text{statEnv} \vdash seq of var &\text{ expands to VarSeq} \\
\text{statEnv} \vdash spos of var &\text{ expands to VarSpos} \\
\text{statEnv} \vdash scur of var &\text{ expands to VarScur} \\
\text{statEnv} \vdash ecur of var &\text{ expands to VarEcur} \\
\text{predPosLimit}(\text{expr}_{\text{start}}, \text{varSpos}) \lor \text{predValLimitBreakable}(\text{expr}_{\text{start}}, \text{schema}, \text{VarScur}) \\
\text{predItemMustOccur}(\text{expr}_{\text{end}}, \text{itemVar}, \text{schema}) \\
\text{statEnv} + \text{varInf}(\text{VarSeq} \succ \varphi, \text{varSpos} \succ \varphi, \text{varScur} \succ \varphi, \text{varEcur} \succ \varphi) &\vdash Expr_{\text{return}} \succ \text{var}_\infty_{\text{return}} \\
\text{statEnv} \vdash \text{forseq}\$seq in \ldots \text{return} \ &\text{Expr}_{\text{return}} \succ \text{var}_\infty_{\text{return}}
\end{align*}

Figure 8.3: End condition looks for element which must occur per schema definition
Some more cases do not concern termination but production of results before end of stream:

- If a query uses `order by`, the first result normally will be produced at end of stream since only then the order operator can be sure that the according elements are really sorted in the correct manner. If the schema tells the engine that the fields the `order by` is looking at are partially ordered, the engine can use non-blocking versions of the order operator and tell them at which states they can produce early results. If the fields are ordered completely, the `order by` clause and therefore the order operator can be dropped. If the `expr_{end}` of the unblocked query is infinite, one will only get results for the first window. If it is finite, results are produced each time a window is bound.

- If the schema defines a key and the query uses in each window only items with the same key-value, the engine can enable splitting parallel windows, handling windows of different key-values in parallel. Therefore, even in the case of an infinite `expr_{end}` one will get results for multiple windows.
Chapter 9

Related work

9.1 Loss pattern

Silberbauer [22] proposes the loss pattern to detect "cancellations of an online order". The online platform fires events after each significant user activity. If these events are not fired after some timespan, a loss event is logged. With the loss event, the system may react to losses in real-time. By analysing and aggregating these loss events, the operator of the online platform can detect the weak points where users cancel the orders, detect hardware problems and maybe find out why the cancellations occur at these points. With our stream descriptions, the engine could detect when an online operation is aborted, since the next arriving item is not the expected one but a new start of an operation. As soon as it detects this abort it could generate a loss event. Note that for this either the schema must tell that at any point there is the possibility of a new start of the operation (sequences of operation workflows for each possible number of items in one choice) or the schema lists only the successful operation and as soon as the stream can not be validated because an abort happened, a loss event is fired and the validation or the overall stream handling process restarts with the new start of the operation. That seems to be the only connection between the loss pattern and our stream descriptions.

9.2 Punctuated streams

Tucker et al [23] describe in their work how to use punctuation semantics to optimise 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.

This is a part of what our optimisations can do as well. But in this area punctuations are much more powerful, since we provide only partial matching and only against schema-listed fields, whereas punctuations can have wildcards and limitations for every field an XML element has.

On the other hand, punctuations have to be inserted by some logic at any place in the stream generation or pass-by stations. This logic however will need some information about when to inject a punctuation, like e.g. "every hour". And this is the place where our solution can fit in or wrap around: Instead of delegating the punctuation insertion statically to the individual places in the stream, the query engine could inject them according to the schema information. E.g. if the schema tells that some attribute either stays the same or increases, the engine could inject at each increase a punctuation for the previous value.

The only cases where we have similar behaviour like punctuations, is when all the attributes needed by the operator have descriptions by <nxt:next> and therefore are sorted. If <nxt:dependency> is used instead of <nxt:next> we can enlarge the imitation of punctuations,
if there is no circle in the dependency graph. This case reflects the case where punctuations for individual elements or element ranges are delivered.

For all the other optimisations we do, punctuations are not enough, since they only provide information about no more occurring elements and not about orderings or other information.

9.3 k-Constraints

Babu et al describe in [31] their work on so called k-Constraints. k-Constraints are the work closest related to ours. We just describe stream descriptions of which a part are very similar constraints. The difference is again a similar one as to the approaches taken in Aurora (of which the slack parameter resembles partly the adherence parameter of k-Constraints): We do not give an absolute range in terms of tuple count where our constraints are valid. Our constraints describe different connections, like dependencies between items by next rules or keys. What our solution does not do compared to k-Constraints, is that we describe the individual streams in an aggregated stream, whereas k-Constraints can describe dependencies between different streams. In addition, with k-Mon, Babu et al. already have a working program which is able to detect possible useful k-Constraints on streams and to exploit them automatically. Our work only describes how to annotate streams and how these stream descriptions can be used to optimise the operations on the stream. On the other side we can describe streams more fine grained by next rules, telling exact dependencies between items, instead of just telling that the item being sought will appear in the range of the next k elements, if it appears.
Chapter 10

Conclusion and future work

In this report we proposed to annotate streams such that processing models can profit from additional knowledge by means of changed semantics and optimising possibilities.

We showed why it is necessary to define a stream description language at all, showing by use cases where processing engines could profit and defined the requirements of minimal additional information.

Following up to these requirements we formed a solution proposal by defining the information needed to fulfill the requirements in a minimal way.

To concretise the general solution proposal, we defined a stream description language based on XML Schema by extending XML Schema with the stream description extensions defined in the step before.

Having designed a stream description language, we analysed two processing models on the view of how they handle streams, what they use streams for and what assumptions they take about streams. We then showed for each of them the semantic changes and the optimisations the additional stream knowledge enables.

Since not all processing models are aware of time, we discussed potential extensions regarding dynamic information in a separate chapter.

To show semantic changes, we also analysed the FORSEQ semantics, since there, many decisions about infinity are enabled the first time by our extensions.

Future work

Future work will consider more detailed integration into processing models to find more complex optimisations as well as semantic adaptations and stream validation (see Section 6.4). As we have stated in Section 6.3.1.3, there are still some descriptions missing to state all inter-dependencies between elements. One possible extension could be to have different selectors for current and next, to tell e.g. that only the next <midyear/> will have a greater year-attribute value than the preceding <endyear/>. At the moment we can only select all <midyear/> and <endyear/> elements and tell that the next will have a greater or equal year-attribute value. And that each <midyear/> will have a greater year-attribute value as in the previous iteration.

To facilitate the process of finding more optimisations and semantic adaptations, future work should also include implementing the stream descriptions into an XQuery environment like e.g. MXQuery [16].

In addition, future work could involve a more complete semantic analysis of FORSEQ, doing the static side-effect analysis as well as the analysis of the dynamic evaluation. As written in the introduction to Chapter 8, there will also be the possibility to find more cases where FORSEQ will
terminate for sure or where it will depend only on a limited number of facts, like e.g. the infinity status of the return expression.

In yet another approach one could extend this work to also contain information about stream dynamics. A short outlook into this topic has been given in Chapter 7.
Bibliography


