Master Thesis

Data lineage/provenance in XQuery

Author(s):
Steiger, Beat

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006117075

Rights / License:
In Copyright - Non-Commercial Use Permitted
Data lineage/provenance in XQuery

Master thesis

Mentor
Prof. Donald Kossmann
donald.kossmann@inf.ethz.ch

Supervisors
Dr. Peter M. Fischer
petfisch@inf.ethz.ch

Kyumars Sheykh Esmaili
kyumarss@inf.ethz.ch

Dr. Boris Glavic
glavic@ifi.unizh.ch

Author
Beat Steiger
steigebe@student.ethz.ch

Systems Group, ETH
Zurich
www.systems.ethz.ch
Zurich, Switzerland

4 Jul 2010
ABSTRACT

Provenance data is required by various domains like scientific data management or workflow systems. For relational databases, generation and processing of provenance data is well documented. There exist various systems which process provenance data in relational databases, e.g. [2],[3] and [5]. But data is often stored in an XML format. Instead of mapping XML data into a relational format, it would be much easier to process XML data using XQuery. Therefore the goal of this thesis is to enable an XQuery engine to generate and store provenance information and to provide query functionality for the provenance data.
# Table of Contents

1 Introduction.................................................................................................................. 5  
1.1 Motivation.................................................................................................................... 5  
1.2 Problem statement......................................................................................................... 5  
   1.2.1 Provenance data processing and representation......................................................... 5  
   1.2.2 Provenance data storage and querying....................................................................... 5  
   1.2.3 Provenance data generation....................................................................................... 6  
1.3 Overview........................................................................................................................ 6  
2 Background...................................................................................................................... 7  
   2.1 Data Provenance.......................................................................................................... 7  
      2.1.1 Contribution semantics......................................................................................... 7  
      2.1.2 Provenance computation....................................................................................... 7  
      2.1.3 Provenance representation.................................................................................... 8  
      2.1.4 Use Cases.............................................................................................................. 8  
         2.1.4.1 Curated databases............................................................................................... 8  
         2.1.4.2 Data warehouses............................................................................................... 9  
   2.2 Core concepts of XQuery............................................................................................ 9  
      2.2.1 The XQuery and XPath Data Model (XDM)......................................................... 9  
      2.2.2 XQuery concepts................................................................................................. 10  
         2.2.2.1 Atomization........................................................................................................ 10  
         2.2.2.2 Effective boolean value.................................................................................... 10  
         2.2.2.3 Document order............................................................................................... 10  
      2.2.3 XQuery language................................................................................................... 11  
         2.2.3.1 Prolog................................................................................................................ 15  
3 Challenges of provenance in XQuery.............................................................................. 16  
   3.1 Challenges of the XQuery data model......................................................................... 16  
   3.2 Challenges of the XQuery language........................................................................... 16  
4 Concepts of XQuery provenance..................................................................................... 17  
   4.1 Contribution semantic classification of provenance in XQuery................................. 19  
      4.1.1 Where-provenance................................................................................................. 19  
         4.1.1.1 Identical copy contribution............................................................................... 20  
         4.1.1.2 Value copy contribution................................................................................... 21  
      4.1.2 Why-provenance.................................................................................................... 21  
         4.1.2.1 Value-influence contribution.......................................................................... 22  
         4.1.2.2 Order-influence contribution............................................................................ 22  
         4.1.2.3 Condition-influence contribution.................................................................... 23  
      4.1.3 How-provenance..................................................................................................... 23  
         4.1.3.1 Transformation provenance.............................................................................. 24  
   4.2 Classification of provenance by the scope of its contribution.................................... 25  
      4.2.1 Sub scope contribution.......................................................................................... 25  
      4.2.2 Super scope contribution....................................................................................... 26  
      4.2.3 Invariant scope contribution.................................................................................. 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.4</td>
<td>Unrelated scope contribution</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Provenance information on the ordering of data</td>
<td>29</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Document order</td>
<td>29</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Query order</td>
<td>30</td>
</tr>
<tr>
<td>4.3.3</td>
<td>User defined order</td>
<td>30</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Unordered</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>Representation of XQuery provenance</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Provenance graph</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Provenance data serialization</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Querying the provenance</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Coupling query data and provenance data</td>
<td>42</td>
</tr>
<tr>
<td>6.2</td>
<td>Provenance data retrieval</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and future work</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Appendix A Schema definition of serialized Provenance</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Appendix B List of operators</td>
<td>52</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 Motivation

Information about the origins and derivation process of data, so called provenance data, is required by various domains like scientific data management or workflow systems to reason about quality of derived data. For relational databases there exist various systems which generate provenance data and provide query facilities for provenance. However there is no such system present for XQuery processing.

1.2 Problem statement

The goal of this thesis is to define a way to represent, store, generate and query provenance information for data transformations expressed in XQuery 1.0.

1.2.1 Provenance data processing and representation

In this thesis, the following questions should be addressed and solved:

- How should provenance data be represented?
  There are two main approaches to represent provenance data. If transformations applied on data can be inverted provenance information on the applied transformations could be used to compute the original data. Information about the origins of data could be annotated to the data and these annotations are then propagated through transformations.

- What's the granularity of data for which provenance information should be available?
  In relational systems there is a basic hierarchical structure – views, relations, tables, tuples, attributes - which could be used to define the granularity of data for which provenance information is processed. For XML data we first need to determine what structure could be used to define granularity and then decide on what level of granularity provenance data should be processed.

- What kind of provenance data should be processed?
  The provenance data could include information about the origin of data, about the presence or absence of data or about the transformations applied to data.

1.2.2 Provenance data storage and querying

Considering the storage and querying of provenance data the following questions should be addressed:

- In what format should provenance data be stored?
  A suitable data format to represent provenance data should be selected depending on the requirements and characteristics of XQuery provenance data. Is XML for example a useful format to store provenance data?

- How should provenance information be linked to the data?
There are various ways to link provenance data with the actual data. Provenance data could be stored along with the actual data and just be marked as provenance data to be distinguished. Provenance data could use some kind of reference (e.g. identifiers, location description, etc.) to refer to the actual data.

- What kind of queries will be asked on provenance information and how can these be translated to the query language of the storage?

Depending on the data format used to represent provenance data there exists a query language that is suited to query provenance data.

### 1.2.3 Provenance data generation

This thesis should present a way to automatically generate provenance information for a given XQuery expression. Therefore an existing XQuery engine should be extended to generate provenance data.

### 1.3 Overview

Section 2 provides some background information on provenance and XQuery. The background information on provenance explains the concept of provenance and the main aspects of provenance information. The background information on XQuery takes a closer look at the data model of XQuery and the XQuery language itself.

Section 3 points out the challenges of defining a provenance management system for XQuery. The subsections focus on the challenges posed by the XQuery data model and the challenges posed by the XQuery language.

The sections 4.1, 4.2 and 4.3 describe in detail our approach to represent and store XQuery provenance data. In section 4.1 we define contribution semantic categories of provenance data. These categories capture different aspects of provenance data and are required to interpret retrieved provenance data. In section 4.2 we also categorize provenance data. But unlike in section 4.1 the focus of categorization is on the scope of contribution. This categorization allows us to deal with one of the challenges described in section 3, the arbitrary nesting of XML data. Another challenge mentioned in section 3, is the role of ordered sequences in XQuery. In section 4.3, we describe how our provenance model is able to describe the provenance of the order of a given sequence of data items.

The sections 5 and 6 describe the representation of XQuery provenance we defined and how provenance information is retrieved from this representation.
2 BACKGROUND

This section contains two subsections providing background information on provenance and XQuery in general. If you are familiar with these concepts, you may skip this section.

2.1 Data Provenance

Data provenance, also referred to as lineage, pedigree, parentage, genealogy or filiation, is metdata, which describes the steps by which data was derived and the sources, from which this data was derived. Provenance information focused on the derivation steps or transformations applied to data, [7] refers to as transformation provenance and provenance information about the source data from which data is derived [7] calls source provenance.

Since data provenance has been studied for various data models (such as relational databases, file systems etc.) the term data item will be used to identify the smallest data unit for which provenance information is available. Most data models have a hierarchical structure and the term level of detail refers to the granularity of a data item in that model. For example in a relational data model a data item could be a tuple, a table or a relation.

2.1.1 Contribution semantics

The contribution semantic describes what kind of information about the provenance of data is provided by a specific provenance information item. For example, provenance information could be about transformations applied on data or the origins of data. The following is a list of terms used to describe categories of contribution semantics.

Where-provenance: The term where-provenance was introduced by [6]. Where-provenance captures the origin of a data item, i.e. the original data where a data item was copied from.

Why-provenance: The term why-provenance was introduced by [6]. Why-provenance captures all data that was used in the computation or transformation of a data item. The provenance information captured by where-provenance is a subset of the provenance information captured by why-provenance.

How-provenance: The commonly used notion of the term how-provenance refers to the semiring-model as defined by Todd Green et al. in [4]. In difference to that notion, in this thesis we use the term how-provenance to describe provenance information that focuses on the transformations applied to data. The provenance information captured by why-provenance is a subset of the provenance information captured by how-provenance.

2.1.2 Provenance computation

There are two main approaches on how to compute provenance data. The eager approach generates provenance data when data is created or transformed. The lazy approach generates provenance data when it is requested. Most of the eager approaches are based on annotating data with information about its sources and transformations. Lazy approaches often need the transformations applied to data to be invertible meaning the source data items can be computed from the result data items using the inverse transformation.
2.1.3 Provenance representation

The representation of provenance information strongly depends on the way provenance information is computed. Lazy approaches using inverse transformations just need to store the applied transformations so these can be inverted to determine the source data. But lazy approaches are rather limited. For example, formal techniques to determine the inverse of a relational query are limited to a certain class of relational queries. Also the provenance information that could be provided by lazy approaches, using inversion of transformations, is limited to the transformations used to derive data and the source data derived by inversion of these transformations. Eager approaches on the other hand could store any data and metadata, besides the derivation history that is available during the derivation process. This information is usually stored in data annotations. But the representation of annotations varies based on the domain, application or purpose for which provenance information is used. Have a look at the use cases in section 2.1.4 for examples. The data processing architecture, such as service oriented architectures or databases, also have an influence on how data is annotated. Representing provenance information in the same data model as the one used by the data processing system sure is an advantage, since provenance information could be stored directly in the data repositories and be queried with the same query language used for the normal data. But the data model of the data processing system might not be suited to represent provenance data, which would reduce query and storage performance.

2.1.4 Use Cases

This section lists use cases which use provenance or lineage data. These use cases should give you an idea where and how provenance or lineage data can be generated and/or be used.

2.1.4.1 Curated databases

Curated databases, as discussed in [1], are databases whose content is collected and maintained mostly by human effort such as consultation, verification and aggregation of existing sources and interpretation of new raw data. Examples for curated databases are UniProt, a database of protein sequence data in molecular biology or the CIA World Factbook, probably the most widely used source of demographic data. Wikipedia or wikis in general are also curated in the sense that their content is entered and verified by humans. But in difference to curated databases, entries in wikis are not structured enough to be machine readable.

Most data in a curated database is copied and edited from existing sources like other curated databases or printed Articles. To manually keep track of the origins of curated data is tedious and error-prone. Therefore automated provenance recording is very helpful.

Data in curated databases is often annotated with meta data such as provenance, the function of a gene in the case of protein databases, etc. If a user of the database annotates data that is the result of some transformation provenance information is needed to propagate this annotation back to the original data. For example, a user of a curated database annotates a data item in a materialized view that is derived from multiple sources. Provenance information can be used, to find the sources from which the view was derived and to propagate the annotation to these sources.
2.1.4.2 Data warehouses

Data warehouses integrate data from multiple sources into an aggregated view while keeping the original depth of data. This view is updated frequently to reflect changes in the source databases. The sources integrated into a data warehouse are usually snapshots of operational databases and external sources. On these sources, a series of cleansing operations and transformations is applied to extract and identify relevant data (Data Mining). The warehouse view is a result of layered data views. Data provenance is needed to trace the data in the warehouse view back to the source from which it was generated (drill through). This is useful to find the sources of faulty data as well as finding additional characteristics of data which were removed by some of the layered views.

2.2 Core concepts of XQuery

XQuery is a query language designed to retrieve and interpret information from XML data sources. This section provides an overview on XQuery 1.0 which is an extension on XPath 2.0.

2.2.1 The XQuery and XPath Data Model (XDM)

XDM is the data model used to represent information contained in the input of any XSLT XQuery processor and all permissible values of expressions in the XSLT, XQuery and XPath languages. XDM is based on the infoset [9], hence any XML data can be represented in the XDM and every XDM instance can be serialized to XML. But in addition XDM provides support for XML Schema types, which provides precise type information on any XML data.

The predefined types available in XDM are the same as the primitive datatypes defined in the XML Schema definition [12], plus five additional types:

- `xs:untyped`
- `xs:untypedAtomic`
- `xs:anyAtomicType`
- `xs:dayTimeDuration`
- `xs:yearMonthDuration`

These additional types are defined in the same namespace as the primitive datatypes of XML Schema. For a definition of the five additional types see section 2.6.2 of [8]. Types are represented in XDM using their unique type name.

Any value represented in XDM is a sequence of nodes and/or atomic values. In the following the term *item* is used to identify a sequence of length one containing either a *node* or an *atomic value*.

XDM distinguishes 7 different node kinds:

- document nodes
- element nodes
- attribute nodes
- text nodes
• namespace nodes
• processing-instruction nodes
• comment nodes

Each node has a unique node identity, a typed value and a string value. Element nodes, attribute nodes, namespace nodes and processing-instruction nodes also have a name. Each node has 17 different so called accessors which represent all the properties of a node, such as child nodes, the parent node, attributes, the uri of the document this node comes from, the type of the node, the nodes value etc. Any node can be the root of a tree which is the representation of an XML fragment in XDM.

An atomic value has only two properties:

• A string-value, which defines the representation of the atomic value as a character sequence.
• An atomic type, which defines how to interpret the character sequence representation.

For example the string "42" in combination with the schema type xs:integer is the XDM representation of the atomic integer value 42. Upon creation of an atomic value, the string is checked to be a valid representation of the associated atomic type.

2.2.2 XQuery concepts

2.2.2.1 Atomization

Any value can be converted to a sequence of typed or untyped atomic values if required by the context. Many operators, for example arithmetic operators, work on atomic values. Atomization provides the means to implicitly convert any sequence of items to a sequence of atomic values. Atomization can be explicitly applied to a value using the built-in fn:data() function. For detailed information on atomization see section 2.4.2 in [10].

2.2.2.2 Effective boolean value

In XQuery almost every XDM value (i.e. every sequence of items, including single items and the empty sequence) has a defined boolean representation, the so called effective boolean value. In certain situations the effective boolean value is required to determine the result of a boolean expression. For example the following expression using the boolean and operator is evaluated to true: <a/> and <b/>. Although the element nodes <a/> and <b/> are not boolean values the boolean expression can be evaluated, because the and operator implicitly determines the effective boolean value of its operands. To explicitly determine the effective boolean value of a given value the build-in fn:boolean() function can be used. For specifications of the fn:boolean() function see section 15.1.1 in [11].

2.2.2.3 Document order

Document order is an ordering defined among all the nodes accessible during the processing of a given query. All accessible nodes belong to one or more trees (i.e. documents or fragments). Document order is defined as the order in which nodes occur in the tree, i.e. the order in which a
depth-first traversal from left to right visits the nodes of a tree. If all accessible nodes in a query belong to multiple trees, the ordering of the trees is depending on the implementation of the query engine.

2.2.3 XQuery language

The basic building block of XQuery is the expression. The language provides several kinds of expressions which may be constructed from keywords, symbols, and operands. In general, the operands of an expression are other expressions. XQuery allows expressions to be nested with full generality.

The following is a list of categories of expressions, each providing a textual description of the expressions in this category and an example of such an expression. For completeness or formal definitions see [10].

Primary expressions

Primary expressions are the basic primitives of the language. They include literals, variable references, context item expressions and function calls. A primary expression may also be created by enclosing any expression in parentheses, which is sometimes helpful in controlling the precedence of operators.

A literal is a direct syntactic representation of an atomic value. For example, a quoted string - "string" - or any number - 42.

A variable reference is a variable name preceded by a $-sign. Example: $varname

A context item expression is a single '.' character which evaluates to the context item. The context item is the current node or atomic value being processed.

A function call consists of the function name followed by the functions arguments enclosed in parentheses. The following list shows all possible origins functions in XQuery:

- A function could be a built-in function of XQuery. See [11] for a complete list of all built-in functions.
- A function could be defined in the prolog of the query.
- A function could be defined in an imported library module.
- A function could be provided by the external environment as part of the static context.

The static context of an expression is the information that is available during static analysis of the expression which includes the function signatures of all available functions.

Path expressions

Path expressions are used to locate nodes within trees. A path expression consists of one or more steps separated by "/" or "/". The result of a path expression is a sequence of nodes, which are at the location - within the tree of the path expression's context - specified by the steps. Each step specifies an axis, a node test and an optional predicate. The axis and the node test are separated by a double colon "::". The predicate is at the end of a step expression and enclosed in square brackets ' [ ' '] '.

The axis of a step expression specifies the range and direction where to look for the target
nodes of a step. For example the child axis defines all child nodes of all nodes in the context as possible target nodes of a step expression. For a complete list of axes see [10].

The node test of a step expression allows to filter the target nodes selected by the axis by their name and type. Let's assume for example the axis of a step expression selected a sequence of element, comment and text nodes. With the node test we can now filter out all the nodes with the name "nodename", which would yield a sequence of all element nodes having the name "nodename". The comment and text nodes would be filtered out since these kinds of nodes do not have a name property.

The predicates of a step expression are optional. A predicate allows to specify a so called predicate expression which is evaluated on every node returned by the node test. If the predicate expression evaluates to a numeric atomic value (i.e. xs:float, xs:double, xs:decimal or any type derived from these three), this value is truncated if possible and then compared to the position of the current node. If this position is equal to the truncated value, the node is included in the result of the step expression, otherwise not. If the predicate expression does not evaluate to a numeric atomic value, the effective boolean value of the result of the predicate expression is determined. If the effective boolean value is true, the current node is included in the step expressions result. For example we could filter out all nodes which have an attribute with the name "id" and the value "1" using the following predicate [@id eq "1"] or select the second node using the following predicate [2].

For ease of use, step expressions have an abbreviated syntax. Most abbreviations are simple substitutions of the axis. The at-sign "@" is the short version of the attribute axis, a single dot "." is the short version of the self axis, two dots "." is the short version of the parent axis and omitting the axis is the short version for the child axis. Beginning the step expression with two slashes "//" instead of one extends the context for the step expression to include all descendants of the original context.

**Examples**

```
(<a><b id="1"/></a>, <a><b id="2"/></a>)
/child::*:b
[attribute::id eq "1"]
```

The context of this step expression contains two element nodes which both have an id attribute. The step expression selects all child nodes of the nodes in the context, which have the name b and an attribute with the name id which has a value equal to "1".

```
<b id="1"/>
```

The result of this step expression is the first child of the first element in the context. It fulfills the predicate condition - the id attribute is "1" - and the node test - its name is b.

```
(<a><b id="1"/></a>, <a><b id="2"/></a>)
/b[@id eq "1"]
```

This path expression is equal to the first example but uses abbreviated syntax.

**Sequence expressions**

The so called comma operator "," is used to explicitly create a sequence of items. The result of each expression in a sequence of expressions separated by the comma operator and enclosed into brackets is concatenated into one sequence containing all results. The sequence is flattened, that means if any of the expressions concatenated with the comma operator evaluates to a sequence, this sequence is dissolved and each of its items added separately to the newly
constructed sequence.

Examples

- \((1, 2, 3)\) This sequence expression consisting of three literal expressions separated by the comma operator evaluates to a sequence of literals.

- \((1, (2, 3))\) This sequence expression consisting of a literal expression and a sequence expression separated by the comma operator evaluates to a sequence of three literals. The result of the inner sequence expression is flattened.

Arithmetic expressions

XQuery provides arithmetic operators for addition +, subtraction -, division \(\text{div}\), multiplication \(*\), integer division \(\text{idiv}\) and modulus \(\text{mod}\). Operands are implicitly atomized and then cast to xs:double.

Comparison expressions

XQuery provides three kinds of comparison expressions called value comparison, general comparison and node comparison.

Value comparison operators are used to compare two singleton values. The following lists the value comparison operators: equality eq, greater than gt, less than lt, greater equal than ge, less or equal than le. Operands are implicitly atomized. A value comparison expression evaluates either to true, false, the empty sequence or an error.

General comparison operators are used to compare sequences of items. General comparisons are existentially quantified comparisons, which means if there exists a pair of items, one item from the first operand the other from the second operand, that fulfils the comparison criterion the general comparison evaluates to true. The following lists the general comparison operators: equality =, inequality !=, less than <, greater than >, less or equal than <=, greater or equal than >=. Operators are implicitly atomized. A general comparison expression evaluates to true, false or an error.

Node comparisons are used to compare two nodes by their identity or by their document order. The following lists the node comparison operators: equality of node identity is, precedence in the document order \(<\), succession in document order \(>>\).

Constructors

Any instance of an XDM node kind can be constructed in XQuery using the corresponding node constructor. For most node kinds there exist two kinds of constructors: direct constructors and computed constructors.

Direct constructors allow to construct nodes by just writing the nodes' representation in XML. The content of element, attribute and namespace nodes could also contain expressions in curly brackets. These expressions are evaluated and their result added to the constructed node's content.
Examples

This are two direct constructors creating an element node with the name shoe, no child nodes and an attribute node with name size and value 7

This expression contains two direct node constructors. The element constructor is the same as in the previous example. But the attribute constructor contains an expression to compute its value.

Computed constructors not only allow to use expressions to compute the content of a node but also to compute its name. A computed constructor begins with a keyword that identifies the type of node to be created: element, attribute, document, text, processing-instruction or comment. For named nodes follows the name or an expression enclosed in curly brackets computing the name. At the end of a computed constructor follows a list of expressions enclosed in curly brackets which compute the node’s content.

FLWOR expressions

FLWOR expressions allow to iterate over sequences of items and bind each item individually to a variable. The name FLWOR comes from the keywords of this expression: for, let, where, order by and return.

For and let clauses generate an ordered sequence of tuples of bound variables, called tuple stream or binding tuple sequence. The optional where clause allows to filter this tuple stream. The optional order by clause allows to reorder the the tuple stream. And finally the return clause constructs the result of the FLWOR expression. The return clause is evaluated on each tuple, which remains in the tuple stream after being filtered by the where clause, using the bindings defined by the tuple. The result of a FLWOR expression is an ordered sequence containing the results of the return clause evaluations on each tuple.

Conditional expressions

Conditional expressions are based on the keywords if, then and else. The result of a conditional expression is chosen based on the effective boolean value of the expression specified after the if keyword, the so called test expression. The values a conditional expression may choose for the result are specified by the expressions after the then and else keyword. These expressions are called, then-expression and else-expression.

Example

Assume we are a logistics company and need to categorize parcels by their size. The $size variable holds the maximum dimension (i.e. the maximum of breadth, width and height) of a parcel, in meters. The conditional expression returns "small", if the effective boolean value of the test-expression $size lt 2 is true, i.e. if the atomization of the value bound to the $size variable returns a numeric value smaller than 2. Otherwise the conditional expression returns "large".
2.2.3.1 Prolog

The query body of a query is defined as the one expression which defines the result of the query. Any query text before the query body belongs to the prolog. In the prolog modules are declared and/or imported, external variables declared, schema definitions imported, locally defined functions declared etc.
3 CHALLENGES OF PROVENANCE IN XQUERY

This section lists some of the challenges we encountered when defining a provenance management system for XQuery. In subsection 3.1 we discuss the challenges of provenance posed by the data model of XQuery – XDM. In subsection 3.2 we discuss the challenges posed by the expressiveness of the XQuery language. As SQL might be the most prominent rival as a query language providing provenance management, some of the following discussions point out the differences between provenance management in a relational database and XQuery.

3.1 Challenges of the XQuery data model

One of the greater challenges is the dual notion of data represented in XDM. XDM uses trees of nodes to represent the hierarchically structured XML data and atomic values to represent any literal value, such as strings and numbers. Every XDM node has a unique node identity which could be used to distinguish two nodes with different origins, even if they have same name and content. On the other hand atomic values have no identity. So if we were to use node identity to derive provenance we would not be able to distinguish provenance information of an atomic value which was derived multiple times. For example the why-provenance of the atomic value 42 derived by a simple query which computes the number 42 twice, once by adding 40 and 2 and once by multiplication of 6 and 7 would include all operands of all derivations of 42: 40, 2, 6 and 7.

Relational data is hierarchically structured into relations, tuples and attributes. Attributes could themselves contain further structured data, but in general the structural hierarchy ends there. Hence the attribute of a tuple in a relation is the natural structural unit of data for provenance management. Provenance information on an attribute relates to the entire value of this attribute and any transformation involves the entire value of this attribute. XML data has also a hierarchical structure but unlike in a relational setting its hierarchy is not limited. Every XML element could have child elements. So if we choose XML elements or rather XDM nodes - since XDM is the data model used in XQuery - to be the data unit for provenance data management, then provenance information associated to an XDM node consists of the sum of all provenance information of all child nodes plus the provenance information of the XDM node itself. Propagating provenance information through transformations becomes infinitely more complex since provenance of all descendant children needs to be propagated as well.

A relation in a relational database is a set or bag of tuples. The order of the tuples is of no importance unless the order is defined by an ‘ORDER BY’ clause. So the order of the tuples in a result either depends on the implementation of the relational database or was defined by an ‘ORDER BY’ clause. In XQuery any data is treated as an ordered sequence of XDM nodes. Changing the order of the nodes in a sequence may not change the where-provenance of the individual nodes but there are use cases where the provenance of the order itself is of interest. For example one might be interested in the order of the authors listing of a bibliographic database.

3.2 Challenges of the XQuery language

The dual notion of XDM nodes and atomic values leads to implicit operations, such as atomization and effective boolean value, which map nodes to atomic values where it is needed. Although these operations are performed implicitly by most transformations requiring atomic
values, making them explicit for provenance management would generate much more comprehensible provenance information.

Atomization itself is a difficult transformation to categorize its input nodes into contribution semantic categories. The way how nodes are atomized depends on the node kind, type and value. For example the atomization of a typed element node basically returns a sequence containing all typed values of the node, whereas the atomization of an untyped element node basically returns the string concatenation of the string values of its text node descendants.

Data transformations in relational databases can be expressed in relational algebra. The number of operators in relational algebra is fairly small: access a base relation, projection, selection, union, intersection, complement, crossproduct, joins and aggregation. Once it is defined how these operators handle provenance information the problem of propagating provenance through data transformations is solved. For example, [2] defines eight rewrite rules which define how to rewrite SQL queries such that provenance information is included and properly propagated. The number of functions and operators defined in XQuery 1.0 is much larger. See [11] for a complete list of functions and operators defined in XQuery 1.0. Defining for each function or operator how to handle provenance information would be tedious, however functions and operators can be categorized into groups of similar provenance behaviour.

In relational databases, data is mainly copied or aggregated. One exception are constants in a query, for example when adding something like ‘42 AS attname’ in a ‘SELECT’ clause. In XQuery, every XDM node type can be created within a query. The content and name (if it has one) of a created node, can be computed or defined by a constant. So what's the where-provenance of a constructed XDM instance? Any node copied into the content of a constructed node gets a new identity to express that it is not just the same node in a different place but a new node with identical content. Nevertheless, provenance information should somehow express that this new node originated from another node.

In XQuery, arbitrary recursion is possible and the depth of recursion and memory usage is limited by the implementation of the query engine. There are XQuery implementations, for example Zorba xquery, which can evaluate infinite recursive expressions using lazy evaluation. So an XQuery provenance management system should be able to represent possibly infinite data structures and data derivations.
4 CONCEPTS OF XQUERY PROVENANCE

Some of the most standard provenance queries are of the form: “What input items to this transformation, contributed to that specific item in the result?” Or in the other direction: “To what items in the result, did this input item contribute?” So we looked at some standard XQuery use cases and tried to decide, what the input items of a given query are. We soon realized that most queries loaded some documents and just filtered out and joined nodes of these documents. The input items of the queries, came of course from the loaded documents. But knowing the documents loaded by a query is not enough information for most provenance applications. We needed a more fine grained provenance model, which provides information about the individual expressions of a query and the input and output of these expressions. In this section we present that provenance model in detail.

The provenance model uses contribution semantic categories to distinguish the contributions of input items to the output of an expression. Section 4.1 details the contribution semantic categories we defined.

We distinguish two categories of where-provenance: identical copy and value copy. This distinction is needed because XDM nodes are not just typed values, but have additional properties, such as node identity or accessors like node-name or document-uri. Some transformations preserve all the node properties and some just preserve the content value of a node.

We categorized why-provenance in three different categories: value-influence, condition-influence and order-influence. With this categorization, we can distinguish data items by the aspect of the result they influenced. Value-influence contributions are data items that were used to compute the value of the result. Order-influence contributions are data items that were used to determine the order among data items in the result. Condition-influence contributions are data items that were used in a condition of a branching or filtering expression.

In section 4.2 we define another way of categorizing provenance data. The focus of this categorization is on the scope of contribution. This categorization allows us to deal with the arbitrary nesting of XML data. For each transformation step we detect the transitions in the hierarchical structure. The scope categorization of input data items enables us to determine whether the contribution of a given input data item was to a result node itself or to one of its descendants or ancestors.

In section 4.3 we describe how our provenance model is able to describe the provenance of the order of a given sequence of data items. We define three kinds of order on which the ordering of a sequence could be based: document order, user defined order and query order. Document order is a total ordering of nodes defined by the XQuery specification. The ordering of a sequence is user defined if it was order with an order by clause. If the order of a sequence is based on the order in which expressions in a query are nested, we call this query order.
4.1 Contribution semantic classification of provenance in XQuery

Provenance can be structured into a hierarchical set of categories, each category capturing one important aspect of provenance contribution. Figure 1 shows an overview of the contribution semantic categories we used in our XQuery provenance model. This categorization allows to query provenance data for specific provenance information. The more detailed the contribution semantic categorization is, the more specific provenance could be queried. Due to the higher complexity of XQuery compared to standard SQL, we need a detailed contribution semantic categorization of provenance data. Once we have established a contribution semantic categorization we can classify the inputs to functions and operators of XQuery according to their contribution to the output. Using this we can trace the aspects of provenance captured by the contribution semantic categories through the nested expressions of a query.

![Figure 1: categorization of data provenance](image)

4.1.1 Where-provenance

The where-provenance of a given result data item includes all data items that were copied to the result data item. In terms of data represented in XDM this means the where-provenance of a node or atomic value, which is a result of a data transformation, includes all input nodes of the transformation whose string value or typed value remains unchanged by the transformation and all input atomic values of the transformation whose string value is part of the result either as the string value of an atomic value or string-value or name of a node. Usually nodes included in the where-provenance of a result keep their identity if they were not atomized by the transformation, but there is one exception: Any node copied into the content of a constructed node as one of its descendant nodes gets a new identity. We further distinguish where-provenance into identical copy contributions and value copy contributions.
4.1.1.1 Identical copy contribution

The contribution semantic category of identical copy contributions includes all nodes and atomic values in the where-provenance of a transformation’s result node or result atomic value which have an identical node identity and an identical typed-value in the result. With other words, the identical copy contributions to a result of a transformation are all nodes, whose identity and typed-value has not been modified by the transformation and all atomic values, whose string-value and atomic type has not been modified by the transformation.

In the following examples provenance data is represented like a set of serialized XML data. Each serialized item in the provenance data is separated by a comma ",". The sequence of the comma separated serialized items is enclosed in curly brackets "{" "}". The order among the individual items in the provenance is arbitrary. Although this representation is similar to the representation of sets, provenance data may contain duplicate entries.

Examples

The provenance information of every item in the result of a sequence expression which concatenates several items into a sequence using the comma operator contains the items itself as its identical copy contribution. The construction of a sequence using the comma operator does not modify any property of an item.

Example of identical copy contributions to the result of a sequence expression:

*(41, 42, 43)* This query consists of one sequence expression that concatenates the atomic values 41, 42 and 43 into a sequence

*(41, 42, 43)* The result of this query is of course a sequence containing the three atomic values in the order defined by the query.

*(42)* The identical copy contribution of the second node in the result, 42, is the atomic value 42 itself.

The identical copy contributions of a node in a sequence resulting from a path expression contains each node from which a step expression of the path expression led to the derivation of the given node.

Example of identical copy contributions to the result of a path expression:

```xml
<book>
  <author>
    <name>Joe</name>
  </author>
  <author>
    <name>John</name>
  </author>
</book>/author/name
```

*(<name>Joe</name>, <name>John</name>)* The path expression in this example consists of two step expressions. The first step expression queries all child nodes with the name 'author' of the given XML fragment rooted at the <book> element. The second step queries all child nodes with the name 'name' of the result of the first step expression.

The result of this example path expression is a sequence containing two element nodes.

The identical copy contributions of the first node in the result of the path expressions
contain the <book> element, because this was the source from which the first step expression started, and the <author> element with the child element <name>Joe</name>, because this was the source from which the second step expression started. May this feels unintuitive. Why should the <book> element and <author> element be an identical copy contribution when they are not in the result? The reason for this lies in the scope of contribution. Have a look at section 4.2 to learn about contribution scope.

4.1.1.2 Value copy contribution

The contribution semantic category of value copy contributions is equal to the category of where-provenance in XQuery. Hence all provenance data classified as where-provenance is in the category of value copy contributions. Every input node or input atomic value to a transformation whose string-value has not been modified by the transformation and still is the string-value of one node or one atomic value in the result of the transformation or the name property of a node in the result of the transformation belongs to the contribution semantic category of value copy contributions.

Example

When constructing a node the atomic value defining the name and the items in the content are all value copy contributions. The name of a constructed node has obviously to be valid according to XML specification. Therefore if the name is specified by an expression the result of this expression is atomized and cast to the xs:QName type. The xs:QName type describes all qualified node names.

```
<element title="#some title"/>
<title>some title</title>
{title, "some title"}
```

This query creates an element node with the name 'title' and a single text node descendant with the value "some title".

The result of this query is of course the constructed element node with its text node child.

The value copy contributions to the constructed element node are two atomic values: title and "some title".

4.1.2 Why-provenance

Why-provenance includes all data that was used to compute the result of a transformation. This includes all where-provenance, but also for example, data that was used to determine the order of a sequence or the condition of a branching or filtering expression. Any input node or input atomic value of a transformation belongs to the category of why-provenance. Why-
provenance is split into three disjunct categories specified by the target of the contribution: 
*Value-influence contributions*, which captures all data items that had an influence on the value of a data item in the result of the transformation, *Condition-influence contributions*, which contains all data items that were used in the evaluation of a conditional expression or filtering expression and *Order-influence contributions*, which captures all data items that were used to determine the ordering of the data items in a sequence of data items in the result.

### 4.1.2.1 Value-influence contribution

The contribution semantic category of value-influence contributions includes all input atomic values to a transformation for which changing the string-value and then reevaluating the transformation would return a different value for one of the node properties listed below for at least one node in the result or a different string-value for at least one atomic value in the result.

The contribution semantic category of value-influence contributions also includes all input nodes to a transformation for which changing the value of one of the node properties listed below and then reevaluating the transformation would return a different value for one of the node properties listed below for at least one node in the result or a different string-value for at least one atomic value in the result.

Node properties that could be affected by value-influence contributions:

- base-uri
- document-uri
- attributes
- children
- node-name
- type-name
- string-value
- typed-value
- nilled

**Examples**

All operands in an arithmetic expression are value influence contributions to the result of the arithmetic expression.

\[1 + 41\] The result of this simple addition is of course 42. The operands 1 and 41 are value-influence contributions to the result 42.

\[42*1\] The result of this multiplication is also 42, but according to definition, the first operand is even an identical copy contribution to the result since its type and string value are the same. But as the category of value-influence contributions is a superset of the category of identical copy contributions, the first operand is also a value-influence contribution.

The original node is a value-influence contribution to any atomic value in the result of atomization of an element, document or attribute node.

### 4.1.2.2 Order-influence contribution

The contribution semantic category of order-influence contributions includes all nodes and atomic values which are input to expressions in any order specification of an order by clause and all atomic values computed by evaluating the order specifications on every binding tuple in the tuple stream of the corresponding FLWOR expression.
Example

```xquery
for $x$ in (<a>2</a>, <a>1</a>)
order by $x$ ascending
return $x
```

This is the result of the FLWOR expression.

```xquery
(<a>1</a>, <a>2</a>)
```

The order-by clause of this FLWOR expression implicitly atomizes the values bound to the variable $x$ of each tuple in the tuple stream, created by the `for` clause. The atomic values determined by the atomization are used to reorder the tuple stream in ascending order.

{<a>1</a>, <a>2</a>, "1", "2"}

The order-influence contributions to both nodes in the result are the two nodes themselves, because they were input to the implicit atomization of the `order by` clause, and the atomic string values "1" and "2" which are the result of the atomization.

4.1.2.3 Condition-influence contribution

The contribution semantic category of condition-influence contributions includes all nodes and atomic values which are input to one of the following expressions:

- any test-expression of a conditional expression.
- any expression in the condition of a `where` clause
- any expression in the predicate of a path expression
- any expression in the filter condition of a filter expression

The boolean atomic value which results of evaluating any of these conditions is also included in the condition-influence contributions category.

Example

```xquery
if (1 gt 2) then "1" else "2"
```

This conditional expression compares the atomic integer values 1 and 2 using the greater than operator. The result of this expression is of course "2".

{1, 2, false}

The condition-influence contributions to the result of this expression are the the inputs to the condition: 1 and 2, and the result of the evaluation of the condition: false.

4.1.3 How-provenance

The commonly used notion of the term how-provenance refers to the semiring-model as defined by Todd Green et al. in [4]. In difference to that notion, in this thesis we use the term how-provenance to describe provenance information that focuses on the transformations applied to data. How-provenance does not only focus on the data contributing to the result of a transformation but also includes information about the transformation itself.

In XQuery, transformations consist of expressions. Operands of an expression are themselves expressions. The query plan of an query represents this nesting of expressions as a tree. Each node in the query plan represents an expression and its children represent its operand.
expressions. From now on the term *operator* will be used to refer to nodes in the query plan.

4.1.3.1 *Transformation provenance*

Transformation provenance in general includes all transformation steps that were applied to derive a data item. In XQuery, the query plan could be seen as the transformation provenance of the query result. The model, we defined to represent transformation provenance, is a graph that includes all operators and all input and output data of those operators. Section 5 gives describes this so called provenance graph in detail.
4.2 Classification of provenance by the scope of its contribution

Another way to categorize contributions is to distinguish the scope of contribution. Due to the tree structure of XML data, an input data item to a transformation may only contribute to a descendant of a result node. Or the other way round, only a child or descendant of an input node contributes to the result. The contribution scope is only defined for provenance information of nodes. Atomic values don’t have ancestors or descendants and hence cannot be set into a scope relation with their provenance information. As provenance information of a node includes all provenance information of its descendants, the contribution scope information could be used to filter contributions to a certain node of interest. For example, when using provenance information to determine the origins of erroneous data, the scope information could be used to narrow the search to those nodes, which directly contributed to the node containing the error.

We use four categories to distinguish contribution scope. Each of the following subsections describes one of the contribution scope categories.

In the following, a root node is a node, which is the root of an XML fragment in a sequence. A root node in the result of a transformation is a result root node.

For example, in the following sequence the element nodes represented in bold font are root nodes: \(<a/>, \langle b\rangle \langle c/\rangle \langle /b\rangle, 42\)

![Figure 2: scope categorization of data provenance](image)

4.2.1 Sub scope contribution

Sub scope contributions are all nodes and atomic values in the provenance of a result root node, whose contributions are to one of the result root node’s descendants or attributes. Although attributes are not defined as children of element nodes (and hence are never a descendant of any node) we consider them to be in the sub scope of their parent node to express the inverse notion of an element node being in the super scope of an attribute.

**Example**

Any input to a upward step in a path expression is in the sub scope its corresponding node in the result of the step expression.

```
doc("bib.xml")//chapter/..
```

The doc function loads the content of the “bib.xml” document (see Example 1). The first step expression selects all chapter elements in that document. The second step expression selects all parent nodes of those chapter elements.
The result of the query is a sequence containing the the two book elements from the document.

The sub scope contributions to the second book element in the result are the two child elements named 'chapter' of that book element. They are the input to the upward navigating step expression which derives the second book element.

Example 1: Sample document "bib.xml" containing bibliographic data

4.2.2 Super scope contribution

Super scope contributions are all nodes and atomic values in the provenance of a result root node, whose contributions are to one of the ancestors of the result root node.

Example

Any input to a downward step in a path expression is in the super scope of its corresponding
node in the result of the step expression. The nodes selected by a downward step expression from a node in the input of the step expression are identical copy contributions to the corresponding nodes in the result. The identical copy contribution of the node itself is to a parent or ancestor of the nodes selected by the step expression.

\[
\text{doc("bib.xml")//book/title}
\]

The doc function loads the content of the “bib.xml” document (see Example 1). The first step expression selects all book elements in that document. The second step expression selects all title elements among the child nodes of these book elements.

\[
\langle\text{title}\rangle\text{XQuery: the XML query language}\langle/\text{title}\rangle, \\
\langle\text{title}\rangle\text{XQuery}\langle/\text{title}\rangle)
\]

The result of the query is a sequence containing the two title elements that are children of the book elements.

\[
\{\langle\text{book isbn="0596006349"} \\
... \\
\langle/\text{book}\rangle, \\
\langle\text{bib} \\
... \\
\langle/\text{bib}\rangle}\}
\]

The super scope contributions to the second title element in the result are the parent book element (from the second step expression) and the document itself (from the first step expression).

### 4.2.3 Invariant scope contribution

The category of invariant scope contributions includes all items in the provenance of a result root node, whose contributions are to at least one of the result root node’s properties listed below.

Node properties that could be affected by invariant scope contributions:

- base-uri
- document-uri
- node-name
- type-name
- nilled

For result root nodes of the node kinds listed below the category of invariant scope contributions includes, in addition to the items included above, all items in their provenance whose contributions are to their string-value property or typed-value property.

Node kinds, for which contributions to the string-value or typed-value are included in the category of invariant scope contributions:

- attribute nodes
- comment nodes
- namespace nodes
- text nodes
- processing instruction nodes

Value-influence contributions to a descendant of a node which affect the string-value or typed-value property of this descendant also affect the string-value or typed-value property of
the node itself. The typed-value of a node which has descendants is defined as the sequence of typed-values of its descendants and the string-value as the string concatenation of the string-values of its descendants. Therefore we need to exclude contributions to the string-value or typed-value property of nodes, which can have child nodes, from the category of invariant scope contributions. The node kinds which can have child nodes are element nodes and document nodes.

**Example**

When constructing a named node, for example an element node, the name definition of this node is an invariant scope contribution to the constructed node.

```xml
<element title="XQuery"/>
```

This query creates an element node named title which has a single child node. The child node is a text node with the value “XQuery”.

```xml
<title>XQuery</title>
```

The result of the query

```xml
{title}
```

The invariant scope contribution to the element node created by the query is the atomic value 'title' of type xs:QName used to define the name of the element. xs:QName is the atomic type of valid node names.

### 4.2.4 Unrelated scope contribution

Unrelated scope contributions are all nodes and atomic values in the why-provenance of a root node in the result of a transformation, whose contributions are either to an atomic value or a node with no common ancestor with the root node.

Unrelated scope contributions are also all nodes and atomic values in the why-provenance of an atomic value in the result of a transformation.

For example order-influence contributions or condition-influence contributions are usually unrelated scope contributions to every root node in the result of a transformation.
4.3 Provenance information on the ordering of data

Any value represented in XDM is an ordered sequence. In this section we define where a certain ordering of a sequence can come from. For this reason we defined three basic kinds of ordering: query order, document order and user defined order. The order of a sequence may rely on at least one of these order kinds. The order kinds on which the order of a sequence is based on we call order provenance.

Expressions can be classified according to the order kind of their result. Most expressions do not order their result explicitly, they just preserve the ordering of their input. To find out the order provenance of a given sequence, we just have to find those expressions which explicitly order their result and whose result contributes to the given sequence. The for and let clauses of a FLWOR expression transform their input data into a tuple stream. We call this tuple stream a binding tuple sequence, since it is a sequence of binding tuples. Each binding tuple is a tuple of variable bindings, defined by the for and let clauses of the FLWOR expression. The final ordering of the result of a FLWOR expression, is based on the ordering of the binding tuple sequence at the end of the evaluation of a FLWOR expression.

Table 1 shows which expressions establish what kind of order for sequences and which clauses establish what kind of order for binding tuple sequences. We use the term sequence order to refer to the order kind which defines the order of a sequence and the term binding order to refer to the order kind of a binding tuple sequence.

<table>
<thead>
<tr>
<th>sequences</th>
<th>query order</th>
<th>document order</th>
<th>user defined order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sequence expression</td>
<td>path expression</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set expression</td>
<td></td>
</tr>
<tr>
<td>binding tuple</td>
<td>for and let clauses</td>
<td>-</td>
<td>order by clause</td>
</tr>
<tr>
<td>sequences</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Order provenance of expressions

Given the query order of the for and let clauses, the FLWOR expression maps the sequence order of the input sequences of the for and let clauses, to a binding order. At the end of a FLWOR expression the data items created by the return clause are ordered into a sequence based on the present binding order. Hence FLWOR expressions allow transforming sequence order into binding order and vice versa. Figure 3 shows again which kinds of expressions use what kind of ordering for their result. But also displays the role of FLWOR expressions, mapping between the two kinds of sequences.

4.3.1 Document order

Document order is a total order defined among all nodes which are accessible during the evaluation of a given query. The relative order of two nodes from the same document or XML fragment is defined by the visiting order of a depth-first traversal of the corresponding tree of XDM nodes. The relative order of two nodes from different documents or XML fragments is depending on the implementation of the query processor but is stable, which means during the evaluation of a query the relative order among the two nodes does not change. Document order also does not interleave nodes from different documents or XML Fragments. The relative order among attribute nodes which have the same parent node is depending on the implementation of
the query processor.

Path expressions, set expressions and the built-in XQuery functions `fn:id` and `fn:idref` are the only expressions which explicitly order their result in document order. Any other expression which returns its result ordered in document order does so by preserving the document order of one of its input sequences. Or the ordering of the result is coincidentally equal to document order.

### 4.3.2 Query order

Any ordering of a sequence or binding tuple sequence defined implicitly by the order of expressions in the query code we call query order. Query order is created by sequence expressions and FLWOR expressions.

Sequence expressions order their result according to the order of their operand expressions given by the query text. The operand expression standing left of a comma operator precedes all operand expressions on the right. For example, the following query `(1,2,3)` creates a sequence, in which the atomic items 1, 2, 3 are ordered as they were given in the query.

The order of a binding tuple sequence is based on query order unless it is specified by an `order by` clause. The order of a binding tuple sequence is based on the order of the input sequences to the `for` clauses and the order of nesting of the `for` and `let` clauses. The nesting of `for` and `let` clauses is defined in the query code and therefore defined in query order. Hence the order of the result of a FLWOR expression containing at least two nested `for` or `let` clauses and no `order by` clause is based on the order of the input sequences to the `for` clauses and the query order which specifies the nesting of the `for` and `let` clauses.

### 4.3.3 User defined order

Any ordering of a sequence or binding tuple sequence defined according to an ordering criterion specified by the user is called user defined order. In XQuery the only way to order a sequence according to a defined order specification is to specify an `order by` clause in a FLWOR expression. The `order by` clause allows to define a set of expressions, so called order specifications, which are evaluated on each tuple in the tuple stream and their results atomized.

Figure 3: order provenance

![Diagram of order provenance]

- **query order**
- **user defined order**
- **sequence order**
- **binding order**
- **FLWOR**
- **for, let**
- **path expression, set expression**
- **document order**
- **order by**
and then compared, if possible, to determine precedence. The binding tuple sequence returned by an \texttt{order by} clause is ordered in a user defined order.

4.3.4 \textit{Unordered}

XQuery allows to specify the so called \textit{unordered mode} for the evaluation of expressions. Path expressions, set expressions, \texttt{fn:id} and \texttt{fn:idref} function calls evaluated in the unordered mode are not required to establish document order for their results. The order of the result of these expressions is depending on the implementation of the query processor and therefore considered to be unordered.
5 REPRESENTATION OF XQUERY PROVENANCE

This section describes how XQuery provenance data is represented. Subsection 5.1 describes the model used to represent XQuery provenance data and subsection 5.2 describes how we serialize this model into an XML representation.

The provenance model for XQuery provenance should be able to represent why-provenance. Many XQuery expressions do not simply copy data, but compute more complex transformations. If our model would only represent the where-provenance of data, much provenance information would be lost. The model should also provide information about the transformations used to derive a result from its why-provenance and where-provenance. Without this transformation provenance information, it would be difficult to relate provenance data with the result. Especially why-provenance information, such as condition-influence contributions and order-influence contributions would be difficult to relate to a given result. For example, the condition-influence contribution data of a simple filter expression on the sequence \((1, 2)\) of integer values comparing the context item with the constant \(2\) using the equality comparison would be \(2, 2\) and \(\text{true}\) for the single atomic value \(2\) in the result. Without knowing that the first \(2\) came from the context item and the second \(2\) was a constant and \(\text{true}\), the result of an equality comparison, the condition-influence contributions would be difficult to interpret.

5.1 Provenance graph

Our goal was to find a provenance model which captures where-provenance, why-provenance and transformation provenance and provides provenance information on every individual node and atomic value that exists during the evaluation of a query. Once we have defined such a model we could simplify it, if not all provenance information is of interest. The solution is a graph of the complete data flow for the evaluation of a given query. This graph is a directed acyclic graph with two different node types: \emph{operator nodes} and \emph{data nodes} and two different kinds of edges: \emph{input edges} and \emph{output edges}.

\emph{Operator nodes} represent the individual expressions or clauses of expressions in a query. Every operator could define one or more parameters which specify the operators behaviour. For example, an operator node representing an \emph{order by} clause of a FLWOR expression specifies the order modifier (e.g. ascending, descending) with a corresponding parameter. Every operator that orders its output is annotated with the corresponding order kind (i.e. document order, query order, user defined order or unordered). Operators which preserve the ordering of the input, or return a single item are not annotated. The Appendix B is a list of all operator nodes that are available.

\emph{Data nodes} represent the data processed by the expressions in a query. Each data node represents a single XDM value (i.e. a sequence of items or a single item) or a binding tuple sequence of a FLWOR expression.

An edge leading from a data node to an operator node is an \emph{input edge}. Input edges determine the data items that are processed by an operator. Each input edge is annotated with the contribution semantic category and the contribution scope category of the input data to the output of the operator.

An edge leading from an operator node to a data node is an \emph{output edge}. Output edges determine the data items returned by an operator.
Each operator in the provenance graph corresponds to an operator in the query plan of a given query. The query plan of a query is a tree of operator nodes. Each operator node of the query plan represents an expression of the query and its children are the operand expressions of that expression. The root of the query plan tree represents the expression that defines the query body of a query. The output of the operator node corresponding to this root of the query plan is the result of the query. In the following, the term root operator is used to refer to the operator node which corresponds to the root of the query plan.

Example

Items in data nodes in Figure 4 and Figure 5 are represented by their serialized XML representation. Sequences are represented as a comma separated sequence of their items enclosed in brackets "(" ")". Bindings are represented as a variable name and an item separated by a comma and enclosed in square brackets "[" "]". Binding tuples are represented as a comma separated sequence of bindings enclosed in curly brackets "{" "}". Binding tuple sequences are represented as a comma separated sequence of binding tuples enclosed in brackets "(" ")". For example, the following represents a binding tuple sequence containing two binding tuples which define two bindings: (\{[x, 1], [y, 2]\}, \{[x, 42], [y, 21]\})

Figure 4 shows the provenance graph of the following query. The query is a FLWOR expression over a sequence of atomic integer values. The sequence creation operator creating the input sequence to the for clause operator is not included in the graph because it otherwise would not have fit onto the page.

\[
\begin{align*}
&\text{for } \$x \text{ in } (1,2,3) \\
&\text{where } \$x \text{ gt } 1 \\
&\text{order by } \$x \text{ descending} \\
&\text{return } \$x
\end{align*}
\]

Figure 5 shows a simplified version of the provenance graph which only includes value-influence contributions. This graph still is a directed acyclic graph. Each edge represents a transformation step in the evaluation of the query that had an influence on the value of the result.

To derive this simplified graph, first all operator nodes are removed. For every operator node, every pair of one of its input edges and one of its output edges is replaced by an edge leading from the source node of the input edge to the target node of the output edge. Each of the edges created this way is labelled with the contribution semantic category of the corresponding input edge. Finally all edges that are not labelled with a contribution semantic category included in the value-influence category are removed.

5.2 Provenance data serialization

To store XQuery provenance information we decided to use XML. Using XML for provenance storage has the advantage that the actual data and provenance data can be queried by the same language. But the tree structure of XML is not suited to represent directed acyclic graphs like the provenance graph. To solve this problem we had two options: either we transformed the provenance graph into a tree by duplicating nodes with multiple parent nodes, or we used references to refer to multiple parent nodes. We settled for the second option. Duplicating nodes would destroy the node identity property of nodes such that comparing individual nodes in the provenance information could not be done with the node comparison
Figure 4: Provenance graph of a FLWOR expression
operators of XQuery.

When serializing XDM values, any atomic value becomes a text node in the serialized document. For example the following sequence of atomic integer values \((4, 2)\) would be serialized to a text node with the string-value "4 2". Serialized atomic values cannot be deserialized into their original XDM representation. A text node with the value "4 2" could have been a real text node before serialization. To preserve the atomic values in the serialized provenance data we wrap each atomic value with an atomicValue element. atomicValue elements have a \(\text{typeName}\) attribute that stores the type of the atomic value. For example the atomic integer value 42 would be serialized to the following atomicValue element:

\[
<\text{atomicValue \(\text{typeName}="\text{xs:integer}\)}">42</\text{atomicValue}>
\]

The serialization format of XQuery provenance is specified by a schema definition (Appendix A). All provenance data, represented in XML is in its own namespace, such that it can be distinguished from data of the query evaluation.

A serialized provenance graph basically consists of two lists of nodes: A list of the operator nodes and a list of data nodes.

The list of operator nodes is represented by an \text{operators} element containing one \text{rootOperator} element and a sequence of \text{operator} elements. The \text{rootOperator} element represents the root operator of the provenance graph. The \text{operator} elements represent the other operator nodes of the provenance graph. The \text{rootOperator} element and all \text{operator} elements have a unique \text{id} attribute which is used to identify a specific operator.

Each \text{operator} element contains a sequence of \text{parameter} elements and a sequence of \text{argument} elements. The \text{parameter} elements describe the parameters of a given operator. The \text{argument} elements describe the input data of an operator. An argument element has exactly one of the following child elements: \text{dataRef} element, \text{constantRef} element, \text{sequenceRef} element or \text{bindingTupleSequenceRef} element.

\text{dataRef} elements define a reference to a node or an atomic value, that was derived by an expression in the query. To identify the operator corresponding to the expression that derived the

\[
\begin{aligned}
\{1, 2, 3\} & \quad \{\{x, 1\}, \{x, 2\}, \{x, 3\}\} \\
\{\{x, 2\}, \{x, 3\}\} & \quad 1 \quad 1 \quad 2 \quad 1 \quad 3 \quad 1 \\
\{\{x, 3\}, \{x, 2\}\} & \quad 2 \quad 3 \quad (f, t, t) \\
\{2, 3\} & \quad 3 \quad 2 \\
\{3, 2\} & \\
\end{aligned}
\]

\textit{Figure 5: Simplified graph, which only contains value-influence contributions}
item referenced by the dataRef element, it has a transformationProvenance attribute. The transformationProvenance attribute basically represents an output edge of the provenance graph.

constantRef elements define a reference to an atomic value which was not derived by the query but is given as a literal expression.

sequenceRef elements define a reference to a sequence that was derived by an expression in the query. Similar to dataRef elements sequenceRef elements have a transformationProvenance attribute to identify the operator that produced the referenced sequence.

bindingTupleSequenceRef elements define a reference to a binding tuple sequence that was derived by a for, let, order by or where clause of a FLWOR expression in the query. Similar to dataRef elements, bindingTupleSequenceRef elements have a transformationProvenance attribute to identify the operator that produced the referenced binding tuple sequence.

The list of data nodes is represented by a queryData element containing a sequence of XMLData elements, sequence elements and bindingSequence elements. The XMLData elements contain all the XML documents, XML fragments and atomic values that exist during the evaluation of a given query. Each sequence element represents a sequence of nodes and atomic values. The content of a sequence element is a sequence of dataRef elements. dataRef elements are references to nodes or atomic values represented in XMLData elements. The content of bindingSequence elements is a sequence of bindingTuple elements which represents the tuple stream inside a FLWOR expression. Each bindingTuple element contains a sequence of binding elements which associate a variable name with the data bound to that variable using dataRef elements.

Each XMLData element has a unique id attribute. This id attribute is used by dataRef elements to identify the XMLData element that contains the node or atomic value they refer to. To identify the node inside an XMLData element, dataRef elements have a xpath attribute that contains a path expression describing the path from the XMLData element to the node. The path expressions used by dataRef elements are limited to child steps and positional predicates. More complex path expressions are not required to identify a single node in a document or fragment.

Similar to XMLData elements, the sequence elements and bindingSequence elements have a unique id attribute which is used by the sequenceRef elements and bindingTupleSequenceRef elements to identify the referenced sequence and binding tuple sequence.

**Example**

The following document is the serialized graph from Figure 4. The root element of the document is the xps:provenance element which defines the provenance namespace xps.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsps:provenance xmlns:xsps="http://www.systems.ethz.ch/xquery/provenance/serialization"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.systems.ethz.ch/xquery/provenance/serialization file:./serialization.xsd"
><xsps:operators>
<xsps:rootOperator xps:type="FLWOR_expr" xps:id="FLWOR1"
 xps:order="orderPreserving">
<xsps:argument xps:position="1" xps:contributionScope="unrelated"
 xps:contributionSemantic="identical_copy">
```
</xps:argument>
<xps:argument xmlns:xps="http://www.example.com/xps" xps:position="2" xps:contributionScope="unrelated" xps:contributionSemantic="identical_copy">
</xps:argument>
</xps:rootOperator>
<xps:operator xmlns:xps="http://www.example.com/xps" xps:type="return_clause" xps:id="return1" xps:order="orderPreserving">
  <xps:parameter xmlns:xps="http://www.example.com/xps" xps:name="tuple">
    <xps:value>1</xps:value>
  </xps:parameter>
  <xps:argument xmlns:xps="http://www.example.com/xps" xps:position="1" xps:contributionScope="unrelated" xps:contributionSemantic="order-influence">
    <xps:bindingTupleSequenceRef xmlns:xps="http://www.example.com/xps" xps:bindingTupleSequence="bs1" xps:transformationProvenance="order_by1"/>
  </xps:argument>
  <xps:argument xmlns:xps="http://www.example.com/xps" xps:position="2" xps:contributionScope="unrelated" xps:contributionSemantic="identical_copy">
  </xps:argument>
</xps:operator>
<xps:operator xmlns:xps="http://www.example.com/xps" xps:type="return_clause" xps:id="return2" xps:order="orderPreserving">
  <xps:parameter xmlns:xps="http://www.example.com/xps" xps:name="tuple">
    <xps:value>2</xps:value>
  </xps:parameter>
  <xps:argument xmlns:xps="http://www.example.com/xps" xps:position="1" xps:contributionScope="unrelated" xps:contributionSemantic="order-influence">
    <xps:bindingTupleSequenceRef xmlns:xps="http://www.example.com/xps" xps:bindingTupleSequence="bs1" xps:transformationProvenance="order_by1"/>
  </xps:argument>
  <xps:argument xmlns:xps="http://www.example.com/xps" xps:position="2" xps:contributionScope="unrelated" xps:contributionSemantic="identical_copy">
  </xps:argument>
</xps:operator>
<xps:operator xmlns:xps="http://www.example.com/xps" xps:type="variable" xps:id="variable1" xps:order="orderPreserving">
  <xps:parameter xmlns:xps="http://www.example.com/xps" xps:name="tuple">
    <xps:value>1</xps:value>
  </xps:parameter>
  <xps:parameter xmlns:xps="http://www.example.com/xps" xps:name="var">
    <xps:value>x</xps:value>
  </xps:parameter>
  <xps:argument xmlns:xps="http://www.example.com/xps" xps:position="1" xps:contributionScope="unrelated" xps:contributionSemantic="identical_copy">
    <xps:bindingTupleSequenceRef xmlns:xps="http://www.example.com/xps" xps:bindingTupleSequence="bs1" xps:transformationProvenance="order_by1"/>
  </xps:argument>
</xps:operator>
<xps:operator xmlns:xps="http://www.example.com/xps" xps:type="variable" xps:id="variable2" xps:order="orderPreserving">
  <xps:parameter xmlns:xps="http://www.example.com/xps" xps:name="tuple">
    <xps:value>2</xps:value>
  </xps:parameter>
  <xps:parameter xmlns:xps="http://www.example.com/xps" xps:name="var">
    <xps:value>x</xps:value>
  </xps:parameter>
  <xps:argument xmlns:xps="http://www.example.com/xps" xps:position="1" xps:contributionScope="unrelated" xps:contributionSemantic="identical_copy">
    <xps:bindingTupleSequenceRef xmlns:xps="http://www.example.com/xps" xps:bindingTupleSequence="bs1" xps:transformationProvenance="order_by1"/>
  </xps:argument>
</xps:operator>
<xps:operator xmlns:xps="http://www.example.com/xps" xps:type="order_by_clause" xps:id="order_by1" xps:order="userDefinedOrder">
  <xps:parameter xmlns:xps="http://www.example.com/xps" xps:name="order_modifier">
    <xps:value>descending</xps:value>
  </xps:parameter>
</xps:operator>
6 QUERYING THE PROVENANCE

In this section we discuss how to associate provenance information with the actual query data and how we could query the provenance graph.

6.1 Coupling query data and provenance data

To query provenance data, we need a way to find the provenance information of a given node. For example, we would like to find all value-influence contributions to a given node in the result. How do we access the provenance data of that node? There are various solutions to associate provenance data with the actual query data.

The provenance data of a node that can have child nodes, such as document nodes and element nodes, could be stored as a child node in a specific provenance namespace. This would make it very easy to access provenance information. All that's required to get the provenance data, is to select the descendants in the provenance namespace. The drawback of this method is, that nodes that can not have child nodes, (i.e. attribute nodes, text nodes, comment nodes, processing instruction nodes and namespace nodes) would not have any directly associated provenance information. The only way to access provenance information of these nodes would be to derive it from the provenance information of their parent nodes. But even that is not always possible since not all nodes in a query result have parent nodes.

Another way to associate provenance information with query data would be to extend the data model of XQuery (XDM) such that every node has an additional provenance accessor which returns provenance information of a given node. Similar to some other accessors, like the base-uri accessor or the document-uri accessor, the value of the provenance accessor could be exposed to XQuery by extending the query language with an additional function fn:provenance() which returns the provenance information of the context item. The only drawback of this approach is that atomic values do not have accessors and therefore cannot associate provenance information. But since atomic values do not have an identity like nodes, the provenance information associated with atomic values may refer to multiple independent transformations which derived the same atomic value. Provenance information of atomic values should be accessed by traversing the provenance graph starting from the root operator.

In what format is provenance data returned when accessing it for a given node? It seems there are two options: precompute all provenance data of each contribution semantic category for each item in the query data or compute the provenance information that is required when it is queried.

If we choose to precompute all provenance data for all nodes, accessing provenance information of a given node would return a sequence of items for each contribution semantic category containing the items that fall into the given category. Accessing the transformation provenance of a given node would return the query plan of the query used to derive that node.

If we choose to compute provenance information when it is required, accessing provenance information of a given node would return the provenance graph and a reference to the operator, that created the given node, or the operator, that last accessed the given node. With that information, we then can retrieve the provenance information needed by traversing the provenance graph and filtering out the required provenance information.
6.2 Provenance data retrieval

In this subsection we describe how to retrieve specific provenance information from the provenance graph.

The serialized provenance graph uses ids and references to represent every edge of the provenance graph. Following a path in the provenance graph means to perform a join operation for every edge we want to follow. Querying provenance this way would be tedious, therefore we provide an XQuery module that contains functions for querying the provenance graph. The provenance query module contains functions to retrieve all provenance data of a specific contribution semantic category for a given node. These functions recursively follow each path in the provenance graph that begins at a given starting node. The starting node is the operator node that last accessed the given node. While following a path in the provenance graph, the functions collect all data nodes on the path that fall into the required contribution category. Since the provenance graph is a directed acyclic graph the recursion eventually arrives at a leaf node which usually is a data node of a constant value.

As in a directed acyclic graph some nodes have multiple parent nodes, these nodes can be reached by different paths. This leads to duplicated items in the provenance information. For where-provenance duplicated items do not make sense, either the result or part of it was copied from a given item or not. For why-provenance however duplicate items make sense. Each item that can be reached by various paths, influences the result in various ways. For that reason the functions in the provenance query module that retrieve where-provenance eliminate duplicate entries in the computed provenance.

The provenance query module also contains functions to query the provenance of the relative order among two nodes or atomic values in a given sequence. The result of such an order provenance query would be a sequence of base order kinds, such as query order, document order, user defined order or unordered. To find out, what base order kinds had an influence on the relative order of two items in a given sequence, we first find the operator node that defined the given sequence. If this operator specifies the order kind of its output, we return this order kind and are done. Otherwise we trace back all paths in the provenance graph that lead to the operator creating the sequence of interest and determine for each operator on these path the order kind it specifies. If an operator on these paths is not order preserving we add its order kind to the result and stop following this path any further. All of the order kinds recorded in this way had an influence to the order of the given sequence. For example, we would like to know what kind of ordering defined the relative order among the two atomic values in the result of the example of Figure 4. We first determine that the FLWOR operator created the result sequence. The FLWOR operator does not specify any order kind, so we trace back all paths leading to this operator. All paths converge at the order by operator which orders its result in a user defined order. We are done, the relative order among the two atomic values in the result is based on user defined order.
7 CONCLUSIONS AND FUTURE WORK

In this thesis we showed a provenance model capable of representing detailed provenance information for XML data processed with XQuery. Furthermore we showed how the model can be serialized into an XML representation and be queried with XQuery. This allows to derive data quality information within the same process that derives the data itself.

For now, there is not much research done in the area of XQuery provenance. This thesis might be the first approach to describe provenance for XQuery.

Our model is capable to represent provenance information for all data transformations in standard XQuery. But there are several recommendations to the XQuery standard which extend XQuery with additional functionality. Examples of such extensions are XQuery and XPath Full Text 1.0, XQuery Update Facility 1.0 or XQuery Scripting Extension 1.0. Future research is required to determine how provenance management can be done in these extensions.

In this thesis we didn't discuss how to handle imported XML data that has already provenance information. The data loaded by a query, using the fn:doc() function, or data from external variables maybe has already provenance information. This provenance information should be integrated into the new provenance information generated by the XQuery provenance system. Further research on how to integrate existing provenance data is required.

Another interesting use of provenance data in XQuery would be to determine the cause of errors. XQuery distinguishes three kinds of errors: static errors, dynamic errors and type errors. Static errors show up in during the static analysis of a query. Syntax errors, for example, are static errors. Dynamic errors show up during the evaluation of a query. For example numeric overflow is a dynamic error. Type errors can show up during the static analysis as well as during the evaluation of a query. For errors that show up during the evaluation of a query, provenance information could help to determine the cause of the error. How and for what kinds of dynamic errors the cause can be resolved using provenance information is a field for further research.

We initially intended to describe a way to automatically generate provenance information for a given query and provide an implementation of XQuery provenance for an existing XQuery processor. Automatic provenance generation, according to the provenance model we specified, is essential to apply the work presented in this thesis. But establishing and defining the provenance model and query facilities required more time than estimated. So the automatic provenance generation is a topic for further research. However, we may shortly present the approach we intended to take. Our intention was to use the trace function of XQuery to automatically generate provenance information. The idea is to 'wrap' every operator of a given query with a trace function that writes information required for provenance to a file. This file could then be processed to derive the provenance model.
REFERENCES


APPENDIX A  SCHEMA DEFINITION OF SERIALIZED PROVENANCE

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xps="http://www.systems.ethz.ch/xquery/provenance/serialization"
    targetNamespace="http://www.systems.ethz.ch/xquery/provenance/serialization"
    elementFormDefault="qualified"
    attributeFormDefault="qualified">

    <xs:annotation>
        <xs:documentation>
            This schema models the serialized Provenance data of a given query.
        </xs:documentation>
    </xs:annotation>

    <xs:element
        name="provenance">
        <xs:complexType>
            <xs:sequence>
                <xs:element
                    name="operators">
                    <xs:annotation>
                        <xs:documentation>
                            This element contains all operator nodes of the transformation provenance graph
                        </xs:documentation>
                    </xs:annotation>
                </xs:element>
                <xs:element
                    name="queryData">
                    <xs:annotation>
                        <xs:documentation>
                            This element contains all data nodes of the transformation provenance graph
                        </xs:documentation>
                    </xs:annotation>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element
        name="operators">
        <xs:complexType>
            <xs:sequence>
                <xs:element
                    name="rootOperator" type="xps:operatorType"/>
                <xs:element
                    name="operator" minOccurs="0" maxOccurs="unbounded" type="xps:operatorType"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element
        name="queryData">
        <xs:complexType>
            <xs:sequence>
                <xs:element
                    name="XMLData" maxOccurs="unbounded" type="xps:XMLDataType"/>
                <xs:element
                    name="sequence" minOccurs="0" maxOccurs="unbounded" type="xps:sequenceType"/>
                <xs:element
                    name="bindingSequence" minOccurs="0" maxOccurs="unbounded" type="xps:bindingSequenceType"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
xpath="xps:queryData/xps:bindingSequence/xps:bindingTuple/xps:binding/xps:dataRef"/>
<xs:field xpath="/xps:data"/>

<xs:complexType name="operatorType">
  <xs:sequence>
    <xs:element name="parameter" minOccurs="0" maxOccurs="unbounded">
      <xs:annotation>
        <xs:documentation>
          parameters are input of an operator which is not XDM. E.g. the ordering direction of the order by clause (ascending or descending)
        </xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:choice>
          <xs:element name="value" type="xs:anySimpleType"/>
          <xs:element ref="xps:atomicValue"/>
          <xs:element name="sequenceType" type="xps:sequenceTypeExpressionType"/>
        </xs:choice>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="contributionSemantic" type="xs:simpleType">
          <xs:restriction base="xs:string">
            <xs:enumeration value="condition-influence"/>
            <xs:enumeration value="order-influence"/>
            <xs:enumeration value="value-influence"/>
          </xs:restriction>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
    <xs:element name="argument" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:choice>
          <xs:element name="dataRef" type="xps:dataRefTransformationProvenanceType"/>
          <xs:element name="constantRef" type="xps:dataRefType"/>
          <xs:element name="sequenceRef" type="xps:sequenceRefType"/>
          <xs:element name="bindingTupleSequenceRef" type="xps:sequenceRefType"/>
        </xs:choice>
        <xs:attribute name="position" type="xs:int" use="required"/>
        <xs:attribute name="contributionScope" use="required" type="xs:simpleType">
          <xs:restriction base="xs:string">
            <xs:enumeration value="unbounded"/>
          </xs:restriction>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
    <xs:element name="bindingTupleSequenceRef" type="xps:sequenceRefType"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="dataRefTransformationProvenanceType">
  <xs:complexContent>
    <xs:extension base="xps:dataRefType">
      <xs:attribute name="transformationProvenance" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="sequenceRefType">
  <xs:sequence/>
  <xs:attribute name="sequence" type="xs:string" use="required"/>
  <xs:attribute name="transformationProvenance" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="bindingRefType">
  <xs:sequence/>
  <xs:attribute name="bindingSequence" type="xs:string" use="required"/>
  <xs:attribute name="position" type="xs:integer" use="required"/>
  <xs:attribute name="variable" type="xs:string" use="required"/>
  <xs:attribute name="transformationProvenance" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="XMLDataType" mixed="true">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip" namespace="#\" any"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="sequenceType">
  <xs:sequence>
    <xs:element name="dataRef" type="xps:dataRefType" minOccurs="0" maxOccurs="unbounded"/>
    <xs:attribute name="id" type="xs:string" use="required"/>
    <xs:attribute name="provenance" type="xs:string" use="required"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="bindingSequenceType">
  <xs:sequence>
    <xs:element name="bindingTuple" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="binding" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="dataRef" type="xps:dataRefType"/>
              </xs:sequence>
              <xs:attribute name="varName" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="position" use="required" type="xs:int">
          <xs:annotation>
            <xs:documentation>
              position within the binding tuple sequence
            </xs:documentation>
          </xs:annotation>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="level" type="xs:integer" use="required">
    <xs:annotation>
      <xs:documentation>
        nesting depth of the binding tuple sequence. E.g. 2 for binding tuple sequence generated by 2. for clause. each nesting of a for or let clause adds one
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:complexType>

</xs:element>
</xs:sequence>
</xs:complexType>
</xs:complexType>
</xs:complexType>
</xs:complexType>
<xs:schema>

  <xs:element name="level" type="xs:string"/>

  <xs:complexType name="atomicValueType">
    <xs:simpleContent>
      <xs:extension base="xs:anySimpleType">
        <xs:attribute name="typeName" type="xs:QName" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:simpleType name="sequenceTypeExpressionType">
    <xs:restriction base="xs:string">
      <xs:pattern value="empty-sequence\(\)"/>
      <xs:pattern value="document-node\(\(\{\{i\c*\}\}(,\{i\c*\}?)*\)\)?\)
      | schema-element\(\(\{i\c*\}\)\)?\)[^\+\?]*\)"
      | element\(\(\{i\c*\}\)(,\{i\c*\}?)*\)[^\+\?]*\)"
      | attribute\(\(\{i\c*\}\)(,\{i\c*\}?)*\)[^\+\?]*\)"
      | schema-element\(\{i\c*\]\)[^\+\?]*\)"
      | schema-attribute\(\{i\c*\]\)[^\+\?]*\)"
      | processing-instruction\(\{i\c*\]\)[^\+\?]*\)"
      | comment\(\)\)[^\+\?]*\)"
      | text\(\)\)[^\+\?]*\)"
      | node\(\)\)[^\+\?]*\)"
      | item\(\)\)[^\+\?]*\)"
      | \(\)\)[^\+\?]*\)"
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
# Appendix B List of Operators

The following is a list of all possible operator nodes in a provenance graph. For each operator we define its type, its ordering behaviour, its parameters, its arguments and its output. Arguments are either required or optional indicated by the word 'required' or 'optional' at the beginning of an argument description. After that follows the type description of the argument. The type description of an argument is a comma separated list of possible kinds of data nodes allowed for that argument. After the type description follows the contribution semantic category of this argument.

<table>
<thead>
<tr>
<th>type</th>
<th>for_clause</th>
<th>order</th>
<th>orderPreserving</th>
<th>parameters</th>
<th>arguments</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• var: variable name</td>
<td>• optional: binding sequence (identical copy) • required: sequence of nodes or atomic values (identical copy)</td>
<td>binding sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>let_clause</th>
<th>order</th>
<th>orderPreserving</th>
<th>parameters</th>
<th>arguments</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• var: variable name</td>
<td>• optional: binding sequence (identical copy) • required: node, atomic value, sequence (identical copy)</td>
<td>binding sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>where_clause</th>
<th>order</th>
<th>orderPreserving</th>
<th>parameters</th>
<th>arguments</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• required: binding sequence (identical copy) • required: sequence (condition-influence)</td>
<td>binding sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>where_expr</th>
<th>order</th>
<th>orderPreserving</th>
<th>parameters</th>
<th>arguments</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• required (n times): boolean atomic value (identical copy)</td>
<td>sequence of boolean atomic values</td>
</tr>
</tbody>
</table>

52
<table>
<thead>
<tr>
<th>type</th>
<th>order_by_clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>userDefinedOrder</td>
</tr>
<tr>
<td>parameters</td>
<td>• order_modifier: 'ascending', 'descending'</td>
</tr>
</tbody>
</table>
| arguments    | • required: binding sequence (identical copy)  
|              | • required: sequence (order-influence)       |
| output       | binding sequence             |

<table>
<thead>
<tr>
<th>type</th>
<th>order_by_expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required (n times): numeric atomic value</td>
</tr>
<tr>
<td>output</td>
<td>sequence of numeric atomic values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>return_clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
</tbody>
</table>
| arguments    | • required: binding sequence (order-influence)  
|              | • required: node, atomic value, sequence (identical copy) |
| output       | node, atomic value, sequence  |

<table>
<thead>
<tr>
<th>type</th>
<th>FLWOR_expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required (n times): node, atomic value, sequence</td>
</tr>
<tr>
<td>output</td>
<td>sequence of nodes and/or atomic values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>quantified_expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• var: variable name</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: sequence</td>
</tr>
<tr>
<td>output</td>
<td>binding sequence</td>
</tr>
<tr>
<td>type</td>
<td>satisfies_clause</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• quantor: 'some', 'every'</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: binding sequence (value-influence)</td>
</tr>
<tr>
<td></td>
<td>• required: sequence of boolean atomic values (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>boolean atomic value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>case_clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• var: variable name</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: node, atomic value, sequence (identical copy)</td>
</tr>
<tr>
<td></td>
<td>• required: boolean atomic value (condition-influence)</td>
</tr>
<tr>
<td>output</td>
<td>binding sequence (possibly empty binding sequence if no variable is specified)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>typeswitch_expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: node, atomic value, sequence (identical copy)</td>
</tr>
<tr>
<td>output</td>
<td>node, atomic value, sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>instance_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• sequence_type: a sequence type description</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: node, atomic value, sequence (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>if_then_else_expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (condition-influence)</td>
</tr>
<tr>
<td></td>
<td>• required: node, atomic value, sequence (identical copy)</td>
</tr>
<tr>
<td></td>
<td>• required: node, atomic value, sequence (identical copy)</td>
</tr>
<tr>
<td>output</td>
<td>node, atomic value, sequence</td>
</tr>
<tr>
<td>type</td>
<td>step_expr</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>order</td>
<td>documentOrder</td>
</tr>
</tbody>
</table>
| parameters              | •  axis: 'child', 'descendant', 'attribute', 'self', 'descendant-or-self', 'following-sibling', 'following', 'parent', 'ancestor', 'preceding-sibling', 'preceding', 'ancestor-or-self'  
|                        | •  type: 'document', 'element', 'attribute', 'text', 'comment', 'processing-instruction', 'namespace'  
|                        | •  name: a valid node name              
|                        | •  uri: a valid uri                     |
| arguments               | •  required: sequence (identical copy)  |
| output                  | sequence                                |

<table>
<thead>
<tr>
<th>type</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
</tbody>
</table>
| parameters              | •  tuple: integer value                
|                        | •  var: variable name                  |
| arguments               | •  required: binding sequence (identical copy)  |
| output                  | node, atomic value, sequence           |

<table>
<thead>
<tr>
<th>type</th>
<th>user_defined_function_call</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>•  name: function name</td>
</tr>
<tr>
<td>arguments</td>
<td>•  required (m times): atomic value, node, sequence</td>
</tr>
<tr>
<td>output</td>
<td>binding sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>user_defined_function_return</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>•  name: function name</td>
</tr>
</tbody>
</table>
| arguments               | •  required: binding sequence          
|                        | •  required: atomic value, node, sequence (identical copy)  |
| output                  | atomic value, node, sequence           |

<table>
<thead>
<tr>
<th>type</th>
<th>function_call</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>•  name: function name</td>
</tr>
<tr>
<td>arguments</td>
<td>•  required (m times): atomic value, node, sequence (value influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value, node, sequence</td>
</tr>
<tr>
<td>type</td>
<td>fn:id</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
</tr>
<tr>
<td>order</td>
<td>documentOrder</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
</tbody>
</table>
| arguments  | • required: atomic value (value-influence)  
          | • required: node (value-influence)         |
| output     | sequence |

<table>
<thead>
<tr>
<th>type</th>
<th>fn:idref</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>documentOrder</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
</tbody>
</table>
| arguments  | • required: atomic value (value-influence)  
          | • required: node (value-influence)         |
| output     | sequence |

<table>
<thead>
<tr>
<th>type</th>
<th>sequence_constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>queryOrder</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required (n times): atomic value, node, sequence (identical copy)</td>
</tr>
<tr>
<td>output</td>
<td>sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>set_operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>documentOrder</td>
</tr>
<tr>
<td>parameters</td>
<td>• opName: 'union', 'intersect', 'except'</td>
</tr>
</tbody>
</table>
| arguments  | • required: sequence (identical copy)  
          | • required: sequence (identical copy) |
| output     | sequence |

<table>
<thead>
<tr>
<th>type</th>
<th>value_comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• opName: 'eq', 'gt', 'lt', 'ge', 'le'</td>
</tr>
</tbody>
</table>
| arguments  | • required: atomic value (value-influence)  
          | • required: atomic value (value-influence) |
| output     | atomic value |

56
<table>
<thead>
<tr>
<th>type</th>
<th>general_comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• <code>opName</code>: <code>=', '&gt;', '&lt;', '&gt;=', '&lt;=', '!='</code></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: sequence (value-influence)</td>
</tr>
<tr>
<td></td>
<td>• required: sequence (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>node_comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• <code>opName</code>: <code>is</code>, <code>&lt;&lt;</code>, <code>&gt;&gt;</code></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: node (value-influence)</td>
</tr>
<tr>
<td></td>
<td>• required: node (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>atomization</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: sequence (identical copy)</td>
</tr>
<tr>
<td>output</td>
<td>node, atomic value (n times)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>value_influence_atomization</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: element node, document node, attribute node (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>sequence of atomic values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>value_copy_atomization</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: processing instruction node, comment node, text node (value copy)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
<tr>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>type</td>
<td>identical_copy_atomization</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (identical copy)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
<tr>
<td>type</td>
<td>arithmetic_expr</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• operator: ‘+’, ‘-’, ‘*’, ‘/’, ‘idiv’, ‘mod’</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (value-influence)</td>
</tr>
<tr>
<td></td>
<td>• required: atomic value (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
<tr>
<td>type</td>
<td>logical_expr</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• operator: ‘and’, ‘or’</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: boolean atomic value (value-influence)</td>
</tr>
<tr>
<td></td>
<td>• required: boolean atomic value (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
<tr>
<td>type</td>
<td>document_constr</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: sequence (value copy)</td>
</tr>
<tr>
<td>output</td>
<td>node</td>
</tr>
<tr>
<td>type</td>
<td>element_constr</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (value copy)</td>
</tr>
<tr>
<td></td>
<td>• required: sequence (value copy)</td>
</tr>
<tr>
<td>output</td>
<td>node</td>
</tr>
<tr>
<td>type</td>
<td>attribute_constr</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
</tbody>
</table>
| arguments          | • required: atomic value (value copy)  
|                    | • required: atomic value (value copy)  |
| output             | node             |

<table>
<thead>
<tr>
<th>type</th>
<th>text_constr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (value copy)</td>
</tr>
<tr>
<td>output</td>
<td>node</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>comment_constr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (value copy)</td>
</tr>
<tr>
<td>output</td>
<td>node</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>processing-instruction_constr</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
</tbody>
</table>
| arguments          | • required: atomic value (value copy)  
|                    | • required: atomic value (value copy)  |
| output             | node                          |

<table>
<thead>
<tr>
<th>type</th>
<th>cast</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• targetType: atomic type</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
<tr>
<td>type</td>
<td>castable</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>order</td>
<td>orderPreserving</td>
</tr>
<tr>
<td>parameters</td>
<td>• targetType: atomic type</td>
</tr>
<tr>
<td>arguments</td>
<td>• required: atomic value (value-influence)</td>
</tr>
<tr>
<td>output</td>
<td>atomic value</td>
</tr>
</tbody>
</table>