Projecting XML Documents

Amélie Marian
Columbia University

Jérôme Siméon
Bell Laboratories

Abstract

XQuery is not only useful to query XML in databases, but also to applications that must process XML documents as files or streams. These applications suffer from the limitations of current main-memory XQuery processors which break for rather small documents. In this paper we propose techniques, based on a notion of projection for XML, which can be used to drastically reduce memory requirements in XQuery processors. The main contribution of the paper is a static analysis technique that can identify at compile time which parts of the input document are needed to answer an arbitrary XQuery. We present a loading algorithm that takes the resulting information to build a *projected document*, which is smaller than the original document, and on which the query yields the same result. We implemented projection in the Galax XQuery processor. Our experiments show that projection reduces memory requirements by a factor of 20 on average, and is effective for a wide variety of queries. In addition, projection results in some speedup during query evaluation.

1 Introduction

After several years of development by the World Wide Web Consortium, XQuery [36] is becoming more stable and starts being implemented and used. Although originally designed to query XML databases [27, 38, 6], XQuery is now being considered as a viable alternative in the context of many other XML applications, such as streaming [35], information integration [14, 22], services [12], full text querying [3, 16], the Semantic Web [2, 26, 24], or simply to process the growing number of XML files generated from various data sources. Main-memory XQuery processors [15, 17, 25, 28] are often the primary choice for those applications that do not wish or cannot afford to build secondary storage indexes or load a database before starting query processing.

However, existing main-memory XQuery implementations break for rather small documents. Table 1 shows the largest document that we were able to process with four popular XQuery implementations and two XSLT implementations, on an IBM T3 laptop with 256Mb of RAM[1]. Memory limitations are problematic as larger XML documents are becoming more common. Only half of the systems we tried, including the more mature XSLT implementations, were able to process the XML version of the EDICT English-Japanese

[1]Those tests were run using the first query of the XMark [31] benchmark, which is a simple lookup query. The XSLT implementations were tested using a simple XSLT translation of the original XQuery.

XQuery Processors	Maximum Document Size
QuiP [25]	7 Mb
Kweelt [28]	17 Mb
IPSI-XQ [15]	27 Mb
Galax [17]	33 Mb
XSLT Processors	**Maximum Document Size**
Saxon [30]	50 Mb
Xalan [33]	75 Mb

Table 1: XML processors maximum document size

dictionary[2] (about 28Mb), and none of them were able to process the XML version of DBLP[3] (about 145Mb). This is due in part to the significant overhead imposed by XML data models [4, 13], which have been reported in [20, 32], but more importantly to the fact that implementations load the complete document in memory before processing it. In this paper, we propose techniques based on a notion of projection for XML documents to address current memory limitations in main-memory XQuery processors.

We define the projection of an XML document by the set of paths, within the document tree, which specify the nodes to keep in a *projected document*. Our approach relies on the following simple idea: for a given query, a projected document, smaller than the original document, and on which the query yields the same result, can be created. The main technical challenge is to be able to identify at compile-time the paths which are required to evaluate a given query. This requires a static analysis of the paths used within a given XQuery expression, and is difficult because of both syntactic and semantics aspects of the XQuery language.

Syntactic sugar. XQuery often offers several ways to write the same operation. Navigation steps can be written with either the XPath abbreviated or unabbreviated syntax [34], composition of steps can be written with the XPath notation (`/` and `//`) or by composing `for` loops with navigation steps, predicates can be written with the XPath notation (`[..]`) or using a `where` clause, etc.

Variables. The analysis must be able to remember which paths were used to compute the content of each variable, in order to apply navigation steps on variables correctly.

Composability. XQuery expressions can be composed arbitrarily, which means navigation can occur within any sub-expression or be applied to a previously computed result. The analysis has to identify on which part of the document a particular step applies. For instance, if a new element is constructed, further navigation does not apply to the input document and must not be taken into account. Moreover, only certain sub-expressions contribute to the result, while other sub-expressions (such as predicates) do not. Hence navigation steps must be applied selectively to a subset of the paths previously computed.

[2]`http://www.csse.monash.edu.au/~jwb/j_jmdict.html`
[3]`http://dblp.uni-trier.de/xml`

The main contribution of the paper is a static inference algorithm which addresses these problems and computes the paths needed by an arbitrary XQuery expression at compile time. More specifically, the paper makes the following contributions:

- We define a notion of projection suitable for XML documents, based on paths within the document tree.
- We develop a static analysis algorithm for XQuery, which computes the set of paths used during the evaluation of a given expression. This algorithm is shown to be correct, i.e., a query yields the same result when evaluated on the projected document for the inferred paths, as on the original document.
- We present a loading algorithm which, given a set of projection paths, builds the projected document suitable for query processing. This algorithm works for both XML files and XML streams.
- We show how projection techniques can be integrated with minimal effort in a standard XQuery processing architecture.
- We present detailed experiments that demonstrate the effectiveness of the static analysis, and study the impact of projection on execution time.

We implemented the projection technique as part of the Galax [17] XQuery engine. Using projection, we were able to run more than half of the XMark [31] queries over a 1-Gigabyte document using an IBM T23 laptop with 256 Megabytes of memory, and all queries on a 100Mb document, increasing the maximal document size for every query by at least a factor of 5. To the best of our knowledge, this is the first XQuery implementation to support querying over such large XML files without the need for secondary storage indices. Our projection implementation can be downloaded from the Web or tried on-line at:

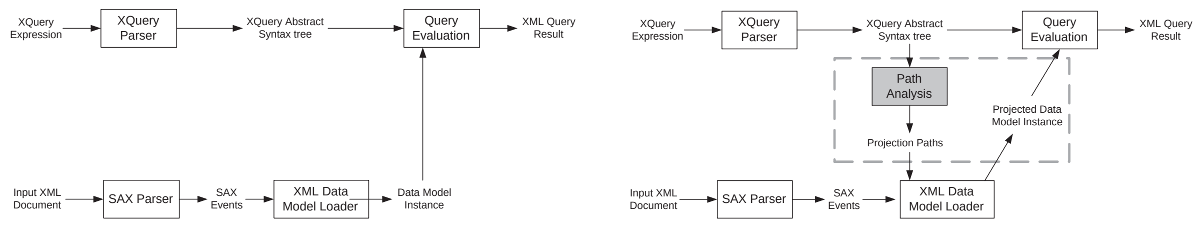
`http://db.bell-labs.com/galax/optimization/`

The rest of the paper is organized as follows. Section 2 describes the architecture of a main-memory XQuery processor and how projection impacts on that architecture. The notion of XML projection is introduced in Section 3. Section 4 gives the static path analysis algorithm for XQuery, and Section 5 describes the loading algorithm. Section 6 contains the experimental evaluation for our projection techniques. We review the related work in Section 7, and conclude the paper with some future work in Section 8.

2 Processing XQuery in Main-Memory

Before describing the projection technique, we first show how it fits in a typical main-memory XQuery processor. We use the Galax system [17] as an illustration. We believe our projection technique can be applied to any other main-memory XQuery implementation in a similar way.

Architecture. Figure 1(a) shows the Galax processing architecture in the absence of projection. On the one hand, the XQuery expression is parsed to an abstract syntax tree. On the other hand, the input document is



(a) XQuery processing architecture

(b) XQuery processing architecture with projection

Figure 1: XQuery processing architecture

parsed in a streamed fashion using SAX [29], then loaded in memory as an XML data model instance. In the case of a streaming processor, the document is parsed directly from the network instead of a local file. Finally, the query is applied on the data model instance to yield a result.

Document Size (text)	**Memory Usage**	**Memory as percentage of Document Size**
500Kb	2.2Mb	392 %
10Mb	38.9Mb	341 %
20Mb	77.9Mb	339 %
50Mb	192.4Mb	339 %

Table 2: Document size in Galax: file vs. memory

The role of the data model. The need for building a data model in memory before query processing is due, to a large extent, to the complexity of evaluating languages such as XSLT and XQuery. Processing XML only as a stream without building a data model instance is an active area of research [1, 9], but such approaches only consider fragments of XPath, and cannot deal with most XQuery expressions. Indeed, many XQuery expressions (joins, type operations such as `typeswitch`, operations on document order, backward XPath axis, function calls, let expressions, namespaces, sorting, etc.) require to materialize part(s) of the document. This is typically done using one of the existing XML data models [4, 13], which provide information necessary for query processing such as node identity, type annotations resulting from validation, namespace nodes, pointers to parent nodes, etc.

The complexity of XML data models accentuate the problems related to memory management in XQuery implementations. Benchmarks [20, 32] show that the size of a DOM representation in memory is typically 4-5 times larger than the original file. Some techniques can be used to build a more compact representation. For instance, Galax uses a simple hash-table to compress the tag names used in the document instead of duplicating them. Still, Table 2 shows that the data model representation in Galax is still 3-4 times larger than the original file. Rather than trying to improve further the data model representation, we focus on avoiding to build a complete data model instance in the first place. Our projection approach is independent from the data model implementation and is expected to result in memory gains regardless of the underlying data model.

Architecture with projection. An advantage of the projection technique is that it can be integrated in a main-memory XQuery processor with minimal effort. Figure 1(b) shows how the modified architecture with projection works: after the query is parsed, it is analyzed to produce a set of projection paths. The result of this analysis is sent to the data model loader which uses it to build a "projected" data model instance which contains only the nodes specified by the projection paths.

3 XML Projection

We now define projection over an XML document, and introduce some notations for the projection paths.

3.1 Example

We illustrate projection using a simple example. Consider the first query of the XMark benchmark [31], which returns the name of the person with `id` attribute `"person0"`.

XMark Query 1

```
  for $b in /site/people/person[@id="person0"]
  return $b/name
```

XMark queries are expressed against a document containing information about auctions, including bidders, bids, items with their descriptions organized by categories, and their location organized by region, etc. A fraction of the XMark document is shown on Figure 2. This is likely that some of the information in the document is not required to answer any particular query. In the case of XMark Query 1 only the `person` elements with their `id` attribute and `name` children are actually needed. The corresponding subset of the original document is indicated in bold on Figure 2.

We use simple path expressions, that we call the *projection paths*, to describe the corresponding subset of the original document. For XMark Query 1, we only need two projection paths:

```
/site/people/person/@id
/site/people/person/name #
```

The '#' notation is used to indicate that the name elements' subtrees, which are part of the query result, should be kept. The result of applying the projection paths to a given document is called the *projected document*. Our projection approach is based on the following observations:

- The projected document tends to be much smaller than the original document. For XMark Query 1, it is less than 2% of the original document.
- The query on the projected document yields the same result as if run on the original document.

```
<site>
  <regions>...</regions>
  <categories>...</categories>
  <catgraph>...</catgraph>
  <people>
    ...
    <person id="person120">
      <name>Wagar Bougaut</name>
      <emailaddress>mailto:Bougaut@wgt.edu</emailaddress>
    </person>
    <person id="person121">
      <name>Waheed Rando</name>
      <emailaddress>mailto:Rando@pitt.edu</emailaddress>
      <address>
        <street>32 Mallela St</street>
        <city>Tucson</city>
        <country>United States</country>
        <zipcode>37</zipcode>
      </address>
      <creditcard>7486 5185 1962 7735</creditcard>
      <profile income="59224.09">
        <education>Other</education>
        <business>Yes</business>
        <age>35</age>
      </profile>
    </person>
    ...
```

Figure 2: The XMark auction document.

It is clear that different sets of projection paths will result in different projected documents. A query will only give a correct result on the projected document if it preserves the information needed to evaluate the query. The algorithm in Section 4 is such that it preserves the information needed to evaluate the given query.

3.2 XML Data Model

We use a simple XML tree data model. We assume the existence of two infinite sets, **String** of string values, and **QName** of qualified names [7]. In addition, $Sequence(S)$ stands for the set of all ordered sequences composed with elements of the set S.

Definition 3.1: [XML Document] A *XML document* is a 5-tuple $(N, tag, child, attr, root)$, where:

- N is a set of nodes;

- tag is a function mapping nodes to their labels or text content, i.e., from N to **QName** $\cup$ **String**;
- $child$ is a function mapping nodes to their children, i.e., from N to $Sequence(N)$;
- $attr$ is a partial function from nodes and attribute name to a sequence of atomic values, i.e., from $N \times \mathbf{QName}$ to $Sequence($**String**$)$;
- $root$ is a special element of N called the root of the tree.

■

3.3 Projection Paths

We define projection paths using a simple fragment of XPath [34], which contains forward[4] navigation but not predicates. A projection path is made of a sequence of steps composed by '`/`'. Each step contains an axis and a node test. Projection paths are described by the following grammar and have the same semantics as in XPath 2.0 [34].

$$\begin{array}{lcl}
SimplePath & ::= & Axis\ NodeTest \\
 & \mid & SimplePath\ \texttt{/}\ Axis\ NodeTest \\
Axis & ::= & \texttt{child::} \\
 & \mid & \texttt{self::} \\
 & \mid & \texttt{descendant::} \\
 & \mid & \texttt{descendant-or-self::} \\
 & \mid & \texttt{attribute::} \\
NodeTest & ::= & ((NCName \mid \texttt{*})\texttt{:})?(NCName \mid \texttt{*}) \\
 & \mid & \texttt{node()} \\
 & \mid & \texttt{text()}
\end{array}$$

Definition 3.2: [Projection Path] A projection path always starts from the root of the document[5], and contains a simple path expression followed by an optional '#' flag.

$$Path \ ::= \ \texttt{/}SimplePath\ \ \texttt{\#}?$$

■

The '#' flag indicates whether the descendants of the nodes returned by the path expression should be kept in the projected document. The '#' flag is merely a convenience, as the relevant paths could always be enumerated.

[4]Note that we do not currently support the `parent` axis, but rewriting techniques such as those presented in [23] should apply.
[5]For simplicity, the presentation assumes there is only one document.

3.4 Projected Document

Projection is an operation that takes a document and a set of projection paths as input and returns a projected document. To define projection, we need the following operation: $PathNodes(P, D)$ returns the set of nodes in the document D that result from the evaluation of the path P on D. Projection is then defined as follows:

Definition 3.3: [Projection] Given an input document D, and a set of projection paths $(ProjPath_1, ..., ProjPath_n)$. The projected document D' is a 5-tuple $(N', tag', child', attr', root')$ such that:

1. $N' \subset N$;
2. $n \in N'$ if:
   - $\exists k : PathNodes(ProjPath_k, D) = n$,
   - or $\exists n_1 : n_1 \in N'$ and $n_1 \in child(n)$,
   - or $\exists n_2, k : PathNodes(ProjPath_k, D) = n_2$ and $n_2 \in ancestor(n)$ and $projPath_k$ has the flag #.
3. tag', $child'$, $attr'$ are the restrictions of tag,$child$, and $attr$ to N';
4. $root' = root$.

■

4 Static Path Analysis

We now present the path analysis algorithm, which computes a set of projection paths from an arbitrary XQuery expression. In Section 4.1, we illustrate some of the problems involved in the development of the algorithm through some examples, and introduce some basic notations. Section 4.2 gives the main algorithm. Section 4.3 states the correctness theorem for the algorithm. It turns out the algorithm in Section 4.2 is not optimal for an important class of path expressions, in Section 4.4 we propose an optimization of the main algorithm which addresses that issue.

4.1 Analyzing an XQuery Expression

Analyzing an arbitrary XQuery expression is not a simple task. In particular, the algorithm must be robust under the syntactic variations supported by XQuery, and must deal with variables and XQuery composability.

4.1.1 XQuery Syntax and the XQuery Core

XQuery often offers several ways to write the same operation. For instance, the following two XQuery expressions are equivalent to the XMark Query 1 given in Section 3.1, but are constructed in very different ways.

Query 1 (a)

```
for $b in /site/people/person
where $b/@id="person0"
return $b/name
```

Query 1 (b)

```
for . in / return
 for . in child::site return
  for . in child::people return
   for . in child::person return
    if ((some $id in (attribute::id) satisfies
         typeswitch ($id)
           case $n as node return data($n)
           default $d return $d) = "person0")
    then child::name
    else ()
```

Query 1(a) is identical to the original XMark Query 1, except that the condition predicate has been expressed with a `where` clause. Query 1(b) seems more complex, but it is the same query, in which some implicit XPath operations have been replaced by explicit XQuery expressions. Path navigation is done step by step, using XPath unabbreviated syntax (`child::`), and binding the current node (`.`) explicitly in a `for` expression. The `where` clause has been replaced by a conditional (`if..then..else`). Finally, a `typeswitch` is used to extract the attribute value.

The path analysis algorithm has to be robust under XQuery syntactic variations. To address this problem, the analysis is performed after the query has been *normalized* in the XQuery core [37]. The XQuery core is a subset of XQuery in which all implicit operations are made explicit. In fact, Query 1 (b) above is very similar to the normalized version of XMark Query 1. An additional advantage is that the paths analysis only has to be defined on the XQuery core instead of the whole language.

Notations. The following grammar gives the XQuery core expressions used in the rest of the paper.

$$
\begin{array}{lcl}
Var & ::= & \texttt{\$}QName \\
Expr & ::= & Literal \\
 & | & \texttt{()} \\
 & | & Expr\texttt{,}\ Expr \\
 & | & \texttt{/} \\
 & | & \texttt{\$}Var \\
 & | & \texttt{for \$}Var\ \texttt{in}\ Expr\ \texttt{return}\ Expr \\
 & | & \texttt{let \$}Var\ \texttt{:=}\ Expr\ \texttt{return}\ Expr \\
 & | & AxisNodeTest \\
 & | & \texttt{if (}Expr\texttt{) then}\ Expr\ \texttt{else}\ Expr \\
 & | & \texttt{typeswitch (}Expr\texttt{)}\ Cases \\
 & | & Expr\ (\texttt{=} \mid \texttt{>})\ Expr \\
 & | & Expr\ \texttt{sortby (}SortList\texttt{)} \\
 & | & \texttt{cast as}\ Datatype\ \texttt{(}Expr\texttt{)} \\
 & | & \texttt{(element} \mid \texttt{attribute) (QName} \mid \{Expr\}\texttt{)}\ \{Expr\} \\
 & | & \texttt{document}\ \{Expr\} \\
 & | & \texttt{text}\ \{String\} \\
 & | & \texttt{QName (}Expr\texttt{?)} \\
Cases & ::= & \texttt{default return}\ Expr \\
 & | & \texttt{case}\ Type\ \texttt{return}\ Expr\ \ Cases \\
SortList & ::= & Expr\ \texttt{(ascending} \mid \texttt{descending) (,}\ SortList\texttt{)?}
\end{array}
$$

This grammar contains: literal values (e.g., strings, integers), the empty sequence (`()`), sequence construction, the root path (`/`), variables, `for` and `let` expressions, XPath steps, conditionals, typeswitch, comparisons, `sort by`, casting, element and attribute constructors, and function calls. This grammar is sufficient [37] to capture all of XPath 1.0 plus XQuery FLWR expressions. We use this grammar to illustrate the technical problems involved in the development of the algorithm.

4.1.2 Variables and Environments

XQuery supports variables which can be bound using, for instance, `let` or `for` expressions. Once a variable is bound, it can be used in a subexpression. During the static analysis, we need to be able to retrieve the set of projection paths that correspond to a given variable, in order to apply further navigation steps. For example, consider the query:

```
for $x in /site/people
return $x/person/name
```

During the analysis, we need to remember that the variable `$x` has been bound to nodes resulting from the evaluation of the path `/site/people`.

To address this problem, we use environments which store bindings between variable and their corresponding projection paths. We will see in Section 4.2 how the environment is maintained for each expression.

Notations. We write

$$Paths = Env(Var)$$

if Var is mapped to $Paths$ in the environment Env. And we write

$$Env' = Env + (Var \Rightarrow Paths)$$

to construct an environment Env' with a new binding for variable Var to the projection paths $Paths$.

For example, the following creates a new environment in which the variable `$x` is bound to the one projection path '`/site/people`'.

$$Env' = Env + (\texttt{\$x} \Rightarrow \{\texttt{/site/people}\})$$

4.1.3 XQuery Composability

XQuery expressions can be composed arbitrarily. To address that problem, the analysis algorithm operates in a bottom up fashion: the set of projection paths for a given expression is computed from those of its subexpressions. The analysis must carefully examine the semantics of each kind of XQuery expression for the algorithm to work, as illustrated by the following example. Consider the query:

```
(if (true())
 then /site/people/person
 else /site/open_auctions/open_auction)/@id
```

This query can be analyzed from its sub-expression:

- `true()` does not require any node from the tree;
- the `then` clause requires nodes reachable from the path: `/site/people/person`
- the `else` clause requires nodes reachable from the path: `/site/open_auctions/open_auction`

Therefore, the conditional requires the set of two paths:

$$\{\ \texttt{/site/people/person}, \\ \texttt{/site/open\_auctions/open\_auction}\ \}$$

The conditional is itself a subexpression of the path expression (`(if ...)/@id`). The path step can be applied to the previous result, giving us the following two paths:

$$\{\ \texttt{/site/people/person/@id}, \\ \texttt{/site/open\_auctions/open\_auction/@id}\}$$

A superficial analysis might conclude that the last navigation step should apply to all paths computed for its input expression. Unfortunately, this does not work in the case wheresome paths are used in the condition, since the corresponding nodes are not actually returned as a result. For instance consider the following variation of the previous query:

```
(if (count(/site/regions/*) = 3)
 then /site/people/person
 else /site/open_auctions/open_auction)/@id
```

By applying the same reasoning as before, we would end up with the following paths:

```
{ /site/region/@id,
  /site/people/person/@id,
  /site/open_auctions/open_auction/@id }
```

However, the path expression `/@id` is never applied to the path `/site/region/*`, the resulting path may not even exist. Nevertheless, the path `/site/regions/*` is indeed necessary to answer the query, but the last step `@id` should not be applied to it.

As a consequence, the algorithm must differentiate paths that are only used during the query, on which no further navigation step will apply, from paths returned by the query. Paths describing nodes which are returned by the query are called *returned* paths. Paths describing nodes which are necessary to compute an intermediate result but are not actually returned as result are called *used* paths.

For the above expression, the set of returned paths is:

```
{ /site/people/person/@id,
  /site/open_auctions/open_auction/@id}
```

and the set of used path is:

```
{ /site/region/* }
```

Notations. We are now ready to introduce the main judgment used during the path analysis. The judgment:

$$Env \vdash Expr \Rightarrow Paths_1 \texttt{ using } Paths_2$$

holds iff, under the environment Env, the expression $Expr$ returns the set of paths $Paths_1$, and uses the set of paths $Paths_2$. Whether this judgment holds or not is defined through the path analysis algorithm itself.

4.2 Paths Analysis Algorithm

We now give the path analysis rules, starting with the simpler expressions. The algorithm is written using the inference rule notation familiar to the fields of programming languages and program analysis [21, 37].

4.2.1 Literal values

Literal values do not require any path.

$$\frac{}{\mathit{Env} \vdash \mathit{Literal} \Rightarrow \{\} \texttt{ using } \{\}}$$

The fact that there is nothing written above the inference rule indicates that this judgment is always true (it does not have any precondition).

4.2.2 Sequences

The empty sequence does not require any path.

$$\frac{}{\mathit{Env} \vdash \texttt{()} \Rightarrow \{\} \texttt{ using } \{\}}$$

Projection paths are propagated in a sequence.

$$\frac{\begin{array}{c}\mathit{Env} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Paths}_1 \texttt{ using } \mathit{UPaths}_1 \\ \mathit{Env} \vdash \mathit{Expr}_2 \Rightarrow \mathit{Paths}_2 \texttt{ using } \mathit{UPaths}_2\end{array}}{\mathit{Env} \vdash \mathit{Expr}_1\texttt{, } \mathit{Expr}_2 \Rightarrow \mathit{Paths}_1 \cup \mathit{Paths}_2 \texttt{ using } \mathit{UPaths}_1 \cup \mathit{UPaths}_2}$$

The two judgments above the rule are preconditions for the judgment below the rule to hold. Computing the projection paths for a sequence of two expressions is done based on the result of computing the projection paths for those two sub-expressions.

4.2.3 Root path

Computing the root expressions requires to keep the root path. The root expression is always the entry point for the query and for the paths analysis.

$$\frac{}{\mathit{Env} \vdash \texttt{/} \Rightarrow \{\texttt{/}\} \texttt{ using } \{\}}$$

4.2.4 Conditionals

Projection paths in the clauses of a conditional expression are propagated. The paths required to compute the condition are added to the final set of used paths.

$$\frac{\begin{array}{c}\mathit{Env} \vdash \mathit{Expr}_0 \Rightarrow \mathit{Paths}_0 \texttt{ using } \mathit{UPaths}_0 \\ \mathit{Env} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Paths}_1 \texttt{ using } \mathit{UPaths}_1 \\ \mathit{Env} \vdash \mathit{Expr}_2 \Rightarrow \mathit{Paths}_2 \texttt{ using } \mathit{UPaths}_2\end{array}}{\begin{array}{c}\mathit{Env} \vdash \texttt{if (}\mathit{Expr}_0\texttt{) then } \mathit{Expr}_1 \texttt{ else } \mathit{Expr}_2 \\ \Rightarrow \mathit{Paths}_1 \cup \mathit{Paths}_2 \texttt{ using } \mathit{Paths}_0 \cup \mathit{UPath}_0 \cup \mathit{UPaths}_1 \cup \mathit{UPaths}_2\end{array}}$$

4.2.5 Comparisons

Comparisons are interesting in that they never return nodes, but a literal (boolean). Therefore, the paths needed for the comparison will not be further modified and are placed in the set of used paths.

$$\frac{\begin{array}{c} Env \vdash Expr_1 \Rightarrow Paths_1 \ \texttt{using} \ UPaths_1 \\ Env \vdash Expr_2 \Rightarrow Paths_2 \ \texttt{using} \ UPaths_2 \end{array}}{Env \vdash Expr_1 \ \texttt{=} \ Expr_2 \Rightarrow \{\} \ \texttt{using} \ Paths_1 \cup Paths_2 \cup UPaths_1 \cup UPaths_2}$$

4.2.6 Variables

The algorithm returns the set of paths to which the variable is bound in the environment, as discussed in Section 4.1.3.

$$\frac{Paths = Env(Var)}{Env \vdash Var \Rightarrow Paths \ \texttt{using} \ \{\}}$$

4.2.7 `for` and `let` expressions

`for` and `let` expressions are binding new variables in the environment.

$$\frac{\begin{array}{c} Env \vdash Expr_1 \Rightarrow Paths_1 \ \texttt{using} \ UPaths_1 \\ Env' = Env + (Var \Rightarrow Paths_1) \\ Env' \vdash Expr_2 \Rightarrow Paths_2 \ \texttt{using} \ UPaths_2 \end{array}}{\begin{array}{c} Env \vdash \texttt{for \$}Var \ \texttt{in} \ Expr_1 \ \texttt{return} \ Expr_2 \\ \Rightarrow Paths_2 \ \texttt{using} \ Paths_1 \cup UPaths_1 \cup UPaths_2 \end{array}}$$

There are two important things to note here. First, the environment is extended with a binding of the variable used in the expression, and passed to the evaluation of $Expr_2$, in order to compute the right set of paths. Then, the returned paths for $Expr_1$ will not be extended any further, unless the variable is used, in which case they will be accessed through the variable, thus these paths are kept as used paths of the `for` expression.

A similar rule applies to `let`.

$$\frac{\begin{array}{c} Env \vdash Expr_1 \Rightarrow Paths_1 \ \texttt{using} \ UPaths_1 \\ Env' = Env + (Var \Rightarrow Paths_1) \\ Env' \vdash Expr_2 \Rightarrow Paths_2 \ \texttt{using} \ UPaths_2 \end{array}}{Env \vdash \texttt{let \$}Var \ \texttt{:=} \ Expr_1 \ \texttt{return} \ Expr_2 \Rightarrow Paths_2 \ \texttt{using} \ UPaths_1 \cup UPaths_2}$$

4.2.8 XPath steps

XPath steps are the most important operation for the path analysis since they actually modify the projection paths. XPath steps are processed by first retrieving the projection paths for the context node `(.)` from the environment, then applying the XPath step to each of the retrieved paths.

$$\frac{\begin{array}{c} Paths = Env(\texttt{.}) \\ Paths = \{Path_1, ..., Path_n\} \end{array}}{\begin{array}{c} Env \vdash Axis\ NodeTest \\ \Rightarrow \{Path_1\texttt{/}Axis\ NodeTest, ..., Path_n\texttt{/}Axis\ NodeTest\}\ \texttt{using}\ \{\} \end{array}}$$

Paths analysis exploits the fact that expressions are normalized into the XQuery core. For instance, the path expression: `/site/people` is normalized as a combination of `for` expressions and path steps:

```
for . in (for . in / return child::site)
return child::people
```

We illustrate the path analysis on this expression step by step. The algorithm starts from the sub-expression '`/`' (matching $Expr_1$ for the inner `for` loop). The name of the inference rule applied is indicated in the prefix.

(ROOT) $Env \vdash \texttt{/} \Rightarrow \{\texttt{/}\}\ \texttt{using}\ \{\}$
(FOR)$_1$ $Env' = Env + (\texttt{.} \Rightarrow \texttt{/})$
(STEP)$_1$ $Env' \vdash \texttt{site} \Rightarrow \{\texttt{/site}\}\ \texttt{using}\ \{\}$
(FOR)$_1$ $Env \vdash \texttt{for . in / ...} \Rightarrow \{\texttt{/site}\}\ \texttt{using}\ \{\texttt{/}\}$
(FOR)$_2$ $Env'' = Env + (\texttt{.} \Rightarrow \texttt{/site})$
(STEP)$_2$ $Env'' \vdash \texttt{people} \Rightarrow \{\texttt{/site/people}\}\ \texttt{using}\ \{\}$
(FOR)$_2$ $Env \vdash \texttt{for . in (for ...} \Rightarrow \{\texttt{/site/people}\}$
$\quad\quad \texttt{using}\ \{\texttt{/}, \texttt{/site}\}$

Note that intermediate paths are bound to the current node (`.`) and retrieved to apply the next XPath step. The resulting set of paths is:

$$\{\ \ \texttt{/}, \texttt{/site}, \texttt{/site/people}\ \}$$

Note that the path analysis keeps all intermediate paths, which can result in the construction of some unnecessary nodes in the projected document. For instance, in the above example, all `site` elements are kept by the projection although we only need the `site` elements which have `people` elements as children to evaluate the query. This is an unwanted side effect of normalization, as all paths expressions are decomposed in `for` loops, resulting in intermediate paths being saved in the set of used paths. We will see in Section 4.4 how to optimize the inference rule for `for` to remove those unwanted intermediate paths.

4.2.9 Typeswitch

Typeswitch, although a complex XQuery operation, is not particularly difficult to analyze. Its inference rule is very similar to the one for conditional, except that it needs to handle multiple branches.

$$\frac{\begin{array}{c} Env \vdash Expr_0 \Rightarrow Paths_0 \ \texttt{using}\ UPaths_0 \\ Env \vdash Expr_1 \Rightarrow Paths_1 \ \texttt{using}\ UPaths_1 \\ \ldots \\ Env \vdash Expr_n \Rightarrow Paths_n \ \texttt{using}\ UPaths_n \end{array}}{\begin{array}{c} Env \vdash \texttt{typeswitch (}Expr_0\texttt{)} \\ \texttt{case}\ Type_1\ \texttt{return}\ Expr_1 \\ \texttt{...} \\ \texttt{default return}\ Expr_n \\ \Rightarrow Paths_1 \cup \ldots \cup Paths_n \ \texttt{using}\ Paths_0 \cup UPath_0 \cup \ldots \cup UPaths_n \end{array}}$$

4.2.10 Sort by

Sort by only returns the expression that is sorted. Expressions used for sorting are not used further in the query and therefore are returned as used paths.

$$\frac{\begin{array}{c} Env \vdash Expr_0 \Rightarrow Paths_0 \ \texttt{using}\ UPaths_0 \\ Env' = Env + (. \Rightarrow Paths_1) \\ Env' \vdash Expr_1 \Rightarrow Paths_1 \ \texttt{using}\ UPaths_1 \\ \ldots \\ Env' \vdash Expr_n \Rightarrow Paths_n \ \texttt{using}\ UPaths_n \end{array}}{\begin{array}{c} Env \vdash \texttt{(}Expr_0\texttt{) sort by} \\ \texttt{(ascending}|\texttt{descending)}\ Expr_1 \\ \texttt{...} \\ \texttt{(ascending}|\texttt{descending)}\ Expr_n \\ \Rightarrow Paths_0 \ \texttt{using}\ UPaths_0 \cup Paths_1 \cup \ldots \cup Paths_n \cup UPaths_1 \cup \ldots \cup UPaths_n \end{array}}$$

4.2.11 Casting

The casting rule is straightforward: the expression being casted is evaluated.

$$\frac{Env \vdash Expr_1 \Rightarrow Paths_1 \ \texttt{using}\ UPaths_1}{Env \vdash \texttt{cast as}\ Type\ \ Expr_1 \Rightarrow Paths_1 \ \texttt{using}\ UPaths_1}$$

4.2.12 Constructors

Constructors always yield a set of empty return paths. As a result, the rest of the analysis always ignore paths applied to a constructed node, since these nodes are created by the query and are not part of the original document. However, in order to keep all information the query may need, the paths returned by the expression used to construct a node must contain the whole subtree.

$$\frac{Env \vdash Expr_1 \Rightarrow Paths_1 \ \texttt{using}\ UPaths_1}{Env \vdash \texttt{(element}|\texttt{attribute)}\ QName\ \ Expr_1 \Rightarrow \{\} \ \texttt{using}\ Paths_1\# \cup UPaths_1}$$

$$\frac{Env \vdash Expr_1 \Rightarrow Paths_1 \texttt{ using } UPaths_1}{Env \vdash \texttt{(document) } Expr_1 \Rightarrow \{\} \texttt{ using } Paths_1\# \cup UPaths_1}$$

$$\frac{}{Env \vdash \texttt{(text) } String \Rightarrow \{\} \texttt{ using } \{\}}$$

4.2.13 Functions

We performed a case by case analysis of built-in functions. Depending on the type of function, the analysis might keep all the function expression paths as used paths, or return some of them. For example, the function `count` returns a literal, so the paths involved in the evaluation of the function are returned as used paths. In contrast, the `union` function returns the nodes resulting from the evaluation of the function, the return paths resulting from the evaluation of the function are then kept as return paths.

Currently, we do not provide user-defined functions analysis as the function itself should be analyzed. For non-recursive functions, we chose the conservative approach and return the function's return paths with their subtrees, and keep the function used paths as used paths.

4.2.14 Wrapping up

Finally, after the set of projection paths has been computed, a # marker must be added at the end of each returned paths, as they corresponds to the actual result of the query.

4.3 Correctness

An essential property of the algorithm is that evaluating the query on the projected document obtained using the paths resulting from the inference must yield the same result than on the original document. The algorithm described in Section 4.2 verifies the following theorem.

Theorem 1 [Correctness] *Let D be an XML document and $Expr$ be an XQuery expression. Let $Paths$ be the result of the static path analysis for $Expr$, i.e., $\vdash Expr \Rightarrow Paths$. Let D' be the projected document of D for the paths $Paths$. Then the evaluation of $Expr$ on D and the evaluation of $Expr$ on D' are the same.*

A proof for the correctness theorem can be constructed by induction on the inference rules for each expressions.

To prove the correctness theorem we need the following results:

Lemma 1: [Return Paths] Applying the *return paths* from the static path analysis of $Expr$ on a document D will return a projected document D'', which contains all nodes that are part of the answer of the evaluation of $Expr$ over D. ■

Proof. [Return Paths] This lemma can be proven by induction over each expression.

- **Literal values:** Literal values never return any nodes, thus the set of nodes returned by a literal is included in the empty set.

- **Root:** The evaluation of the *Root* expression returns the node that is the root of the document, i.e., `root()`, which is the node that can be accessed when applying the return path to the document.

- **Sequences:** Sequences return the union of the nodes of their sub-sequences. Assuming the sub-sequences return paths are correct, then the path analysis rule for sequences will produce *return paths* that contain all nodes that are part of the answer for the sequence expression.

- **Conditionals:** Conditionals return either the nodes that can be reached from the `then` or the `else` expressions. If the set of nodes returned by each expression is included in their return paths, then their union is included in the union of the `then` and `else` return paths.

- **Comparison:** Comparisons return Boolean variables, thus no node are returned by these expression

- **for and let expressions:** for and let return the nodes that can be accessed from their `return` clause expression. If the set of nodes returned by the `return` expression is included in its return paths, then it is included in the `for` (or `let`) expression return paths.

- **Typeswitch:** Typeswitches are very similar to conditionals; they return nodes that can be reached from the `case` and `default` expressions.

- **Sort by:** A Sort by expression only returns the expression that is sorted. If this expression is evaluated correctly, then the sort by is evaluated correctly.

- **Casting:** Casting is straightforward: it returns the nodes from the expression being cast. If this expression is evaluated correctly, then the casting expression is evaluated correctly.

- **Constructors:** Constructors build completely new nodes. Any further operations are applied on these new nodes; thus constructors do not return any node from the original document.

The last two expressions (Variables and XPath steps) also return the correct set of return paths as explained in the following result.

Lemma 2: [Variables] The set of paths associated to a variable allows to reach all nodes that the variable can iterate on. ■

Proof. [Variables] Only two expression can associate a set of paths to a variable: `for` and `let`. These expression associate the set of `return` paths of the expression in their `in` (for) or := (let) clause. The variable is only instanciated with the nodes that are returned from the expression it is associated with. As we saw above, the set of return paths of an expression contains all nodes that are returned by the expression, thus the variable is associated to the correct set of paths.

This result on variables allows to prove that the last XQuery Core expression: **XPath Axis** returns the correct result. XPath Axis applies a path on all nodes associated to the `.` variable. Since we showed that variables to paths mapping are correct, then the XPath Axis expression yields the correct set of *return paths*.

Proof. [Correctness] The correctness theorem states that for an XQuery expression $Expr$, the result of the evaluation of $Expr$ over a document D is the same as the result of the evaluation of $Expr$ over D', where D' is the projected document resulted from applying the projection paths $Paths$ of $Expr$ on D. For this to be correct, D' must contain all the nodes in D that are needed to evaluate the XQuery $Expr$.

We proved that the nodes necessary to evaluate any given sub-expression are contained in the projected document based on the paths for that sub-expression. To prove that the final projection is correct we need to prove that all paths needed by an expression's sub-expressions are present in the expression projection paths:

Lemma 3: [Monotonicity] The projection paths computed for $Expr$ include all projection paths computed by the projection analysis of the sub-expression of $Expr$. ■

Proof. [Monotonicity] Trivial by induction: each inference rule propagates the return paths and used paths of its sub-expression either to its own return paths or used paths. The final projection paths contain both the return paths and the used paths of an expression.

4.4 Optimized Inference Rules

4.4.1 Principle

In this section, we show how to optimize the inference rule of the `for` expression. First we need to understand in which case the original rule computes some unwanted intermediate paths. Recall that in the inference rule for the `for` expression, the set of paths returned by the `in` subexpression is kept as used paths. This is required only when the `for` is applied to certain kinds of sub-expressions during iteration. Distinguishing which sub-expressions can be optimized depend on a whether the expression in the `return` clause yields an observable result when the input is the empty sequence.

For example, consider the following two queries:

```
for $x in //person
return <add>{ $x/address }</add>

for $x in //person return $x/address
```

and assume that some persons have addresses and others do not. In the first query, the persons who do not have an address will still be "visible" in the result as an empty `add` element. In that case, we should then keep the following projection paths:

$\{$ `//person`,
$\{$ `//person/address` $\}$

However, the second query does not return anything for persons which do not have an address. For such a query the projection paths could simply be:

$$\{ \texttt{//person/address} \}$$

The distinction between these two queries is rather subtle. In essence, the reason the second case can be optimized is that when the return clause returns the empty sequence, this does not appear in the final result since sequences are flattened in XQuery.

4.4.2 Revised Inference Rules

We now give the revised inference rules, taking the optimization into account. Our analysis performs in a bottom-up fashion. Thus, each expression, when evaluated, should identify the variables that would evaluate to the empty sequence in that expression, were the variable to itself be the empty sequence. For this we define the optimization variables as:

$$\begin{array}{lcl} OptimVars & ::= & \texttt{\$}Var\texttt{*} \\ & | & AllVariables \\ & | & EmptyVars \end{array}$$

When `OptimVars` is equal to `AllVariables`, the expression can always be optimized, when it is equal to `EmptyVars`, the expression can never be optimized. `OptimVars` otherwise contains a list of variables for which the expression can be optimized.

We define the following operations of `OptimVars`:

- Intersection: intersection with `EmptyVars` is always `EmptyVars`, intersection of `OptimVars` with `AllVariables` is `OptimVars`.
- Union: union with `AllVariables` is always `AllVariables`, intersection of `OptimVars` with `EmptyVars` is `OptimVars`.
- Remove: removes a variable $\$Var$ from `OptimVars`
- Member: checks whether a variable $\$Var$ is in `OptimVars` (all variables are in `AllVariables`)

Notations. The main judgment used during the optimized path analysis is:

$$Env \vdash Expr \Rightarrow Paths_1 \texttt{ using } Paths_2 \texttt{ with } OVars$$

holds iff, under the environment Env, the expression $Expr$ returns the set of paths $Paths_1$, uses the set of paths $Paths_2$, and evaluates to the empty sequences when any of the variables in $OVars$ is binded to the empty sequence. Whether this judgment holds or not is defined through the path analysis algorithm itself.

The optimized inference rules are then:

Literal values

$$\frac{}{\mathit{Env} \vdash \mathit{Literal} \Rightarrow \{\}\ \texttt{using}\ \{\}\ \texttt{with}\ \{\}}$$

Sequences

$$\frac{}{\mathit{Env} \vdash \texttt{()} \Rightarrow \{\}\ \texttt{using}\ \{\}\ \texttt{with}\ \text{AllVariables}}$$

Optimization variables are intersected to evaluate a sequence: the sequence will only evaluate to the empty sequence if both sub-expressions evaluate to the empty sequence.

$$\frac{\begin{array}{c}\mathit{Env} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Paths}_1\ \texttt{using}\ \mathit{UPaths}_1\ \texttt{with}\ \mathit{OVars}_1 \\ \mathit{Env} \vdash \mathit{Expr}_2 \Rightarrow \mathit{Paths}_2\ \texttt{using}\ \mathit{UPaths}_2\ \texttt{with}\ \mathit{OVars}_2\end{array}}{\mathit{Env} \vdash \mathit{Expr}_1\texttt{,}\ \mathit{Expr}_2 \Rightarrow \mathit{Paths}_1 \cup \mathit{Paths}_2\ \texttt{using}\ \mathit{UPaths}_1 \cup \mathit{UPaths}_2\ \texttt{with}\ \texttt{Intersection}(\mathit{OVars}_1, \mathit{Ovars}_2)}$$

Root path

$$\frac{}{\mathit{Env} \vdash \texttt{/} \Rightarrow \{\texttt{/}\}\ \texttt{using}\ \{\}\ \texttt{with}\ \{\}}$$

Conditionals

In conditionals, an empty variable evaluates to the empty sequence if it results in both the `then` and the `else` expressions evaluating to the empty sequence.

$$\frac{\begin{array}{c}\mathit{Env} \vdash \mathit{Expr}_0 \Rightarrow \mathit{Paths}_0\ \texttt{using}\ \mathit{UPaths}_0\ \texttt{with}\ \mathit{OVars}_0 \\ \mathit{Env} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Paths}_1\ \texttt{using}\ \mathit{UPaths}_1\ \texttt{with}\ \mathit{OVars}_1 \\ \mathit{Env} \vdash \mathit{Expr}_2 \Rightarrow \mathit{Paths}_2\ \texttt{using}\ \mathit{UPaths}_2\ \texttt{with}\ \mathit{OVars}_2\end{array}}{\begin{array}{c}\mathit{Env} \vdash \texttt{if (}\mathit{Expr}_0\texttt{) then}\ \mathit{Expr}_1\ \texttt{else}\ \mathit{Expr}_2 \\ \Rightarrow \mathit{Paths}_1 \cup \mathit{Paths}_2\ \texttt{using}\ \mathit{Paths}_0 \cup \mathit{UPath}_0 \cup \mathit{UPaths}_1 \cup \mathit{UPaths}_2\ \texttt{with}\ \texttt{Intersection}(\mathit{OVars}_1, \mathit{Ovars}_2)\end{array}}$$

Comparisons

Comparisons cannot be optimized, they never evaluate to the empty sequence.

$$\frac{\begin{array}{c}\mathit{Env} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Paths}_1\ \texttt{using}\ \mathit{UPaths}_1\ \texttt{with}\ \mathit{OVars}_1 \\ \mathit{Env} \vdash \mathit{Expr}_2 \Rightarrow \mathit{Paths}_2\ \texttt{using}\ \mathit{UPaths}_2\ \texttt{with}\ \mathit{OVars}_2\end{array}}{\mathit{Env} \vdash \mathit{Expr}_1\ \texttt{=}\ \mathit{Expr}_2 \Rightarrow \{\}\ \texttt{using}\ \mathit{Paths}_1 \cup \mathit{Paths}_2 \cup \mathit{UPaths}_1 \cup \mathit{UPaths}_2\ \texttt{with}\ \text{EmptyVars}}$$

Variables

A variable can be optimized with itself.

$$\frac{\mathit{Paths} = \mathit{Env}(\mathit{Var})}{\mathit{Env} \vdash \mathit{Var} \Rightarrow \mathit{Paths}\ \texttt{using}\ \{\}\ \texttt{with}\ \mathit{Var}}$$

`for` expressions

The `for` expression may compute unwanted paths. The optimization step was added specifically to take care of this problem. Thus, this expressions is the core of the optimization.

Two different judgement correspond to the `for` expression depending whether or not the expression can be optimized.

The `for` expression can be optimized if $Expr_2$ always evaluate to the empty sequence when the `for` variable evaluates to the empty sequence, which happens when $\$Var$ is part of $Expr_2$'s $OptimVars$. In such a case, the return paths of $Expr_1$ can be omitted as they will appear through a call to the variable if needed:

$$\frac{\begin{array}{c} Env \vdash Expr_1 \Rightarrow Paths_1 \texttt{ using } UPaths_1 \texttt{ with } OVars_1 \\ Env' = Env + (Var \Rightarrow Paths_1) \\ Env' \vdash Expr_2 \Rightarrow Paths_2 \texttt{ using } UPaths_2 \texttt{ with } OVars_2 \\ \texttt{Member}(\$\text{Var}, OVars_2) \\ OVars_3 = \texttt{Remove}(\$\text{Var}, OVars_2) \end{array}}{\begin{array}{c} Env \vdash \texttt{for \$}Var \texttt{ in } Expr_1 \texttt{ return } Expr_2 \\ \Rightarrow Paths_2 \texttt{ using } UPaths_1 \cup UPaths_2 \texttt{ with Union}(OVars_1, OVars_3) \end{array}}$$

When the `for` expression cannot be optimized the rule is as before:

$$\frac{\begin{array}{c} Env \vdash Expr_1 \Rightarrow Paths_1 \texttt{ using } UPaths_1 \texttt{ with } OVars_1 \\ Env' = Env + (Var \Rightarrow Paths_1) \\ Env' \vdash Expr_2 \Rightarrow Paths_2 \texttt{ using } UPaths_2 \texttt{ with } OVars_2 \\ \texttt{Not Member}(\$\text{Var}, OVars_2) \end{array}}{\begin{array}{c} Env \vdash \texttt{for \$}Var \texttt{ in } Expr_1 \texttt{ return } Expr_2 \\ \Rightarrow Paths_2 \texttt{ using } Paths_1 \cup UPaths_1 \cup UPaths_2 \texttt{ with Union}(OVars_1, OVars_2 \end{array}}$$

`let` expressions

The let expression binds a variable. The variable needs to be removed from the $OptimVars$ as it will not be binded above the let expression:

$$\frac{\begin{array}{c} Env \vdash Expr_1 \Rightarrow Paths_1 \texttt{ using } UPaths_1 \texttt{ with } OVars_1 \\ Env' = Env + (Var \Rightarrow Paths_1) \\ Env' \vdash Expr_2 \Rightarrow Paths_2 \texttt{ using } UPaths_2 \texttt{ with } OVars_2 \\ OVars_3 = \texttt{Remove}(\$\text{Var}, OVars_2) \end{array}}{Env \vdash \texttt{let \$}Var \texttt{ := } Expr_1 \texttt{ return } Expr_2 \Rightarrow Paths_2 \texttt{ using } UPaths_1 \cup UPaths_2 \texttt{ with } OVars_3}$$

XPath steps

XPath steps applied on the empty sequence yield the empty sequence. Since XPath steps are applied on the context node (`.`), if the corresponding variable is empty, the the XPath step evaluates to the empty sequence.

$$\frac{\begin{array}{c} Paths = Env(\texttt{.}) \\ Paths = \{Path_1, ..., Path_n\} \end{array}}{\begin{array}{c} Env \vdash Axis\ NodeTest \\ \Rightarrow \{Path_1\texttt{/}Axis\ NodeTest, ..., Path_n\texttt{/}Axis\ NodeTest\}\ \texttt{using}\ \{\}\ \texttt{with .} \end{array}}$$

Typeswitch

Typeswitch can be optimized if all their return expressions evaluates to the empty sequence.

$$\frac{\begin{array}{c} Env \vdash Expr_0 \Rightarrow Paths_0\ \texttt{using}\ UPaths_0\ \texttt{with}\ OVars_0 \\ Env \vdash Expr_1 \Rightarrow Paths_1\ \texttt{using}\ UPaths_1\ \texttt{with}\ OVars_1 \\ ... \\ Env \vdash Expr_n \Rightarrow Paths_n\ \texttt{using}\ UPaths_n\ \texttt{with}\ OVars_n \end{array}}{\begin{array}{c} Env \vdash \texttt{typeswitch (}Expr_0\texttt{)} \\ \texttt{case}\ Type_1\ \texttt{return}\ Expr_1 \\ \texttt{...} \\ \texttt{default return}\ Expr_n \\ \Rightarrow Paths_1 \cup ... \cup Paths_n\ \texttt{using}\ Paths_0 \cup UPath_0 \cup ... \cup UPaths_n\ \texttt{with Intersection}(OVars_1, ..., OVars_n) \end{array}}$$

Sort by

Sort bys can be optimized if their return expression evaluates to the empty sequence.

$$\frac{\begin{array}{c} Env \vdash Expr_0 \Rightarrow Paths_0\ \texttt{using}\ UPaths_0\ \texttt{with}\ OVars_0 \\ Env' = Env + (. \Rightarrow Paths_1) \\ Env' \vdash Expr_1 \Rightarrow Paths_1\ \texttt{using}\ UPaths_1\ \texttt{with}\ OVars_1 \\ ... \\ Env' \vdash Expr_n \Rightarrow Paths_n\ \texttt{using}\ UPaths_n\ \texttt{with}\ OVars_n \end{array}}{\begin{array}{c} Env \vdash \texttt{(}Expr_0\texttt{) sort by} \\ \texttt{(ascending}|\texttt{descending)}\ Expr_1 \\ \texttt{...} \\ \texttt{(ascending}|\texttt{descending)}\ Expr_n \\ \Rightarrow Paths_0\ \texttt{using}\ UPaths_0 \cup Paths_1 \cup ... \cup Paths_n \cup UPaths_1 \cup ... \cup UPaths_n\ \texttt{with}\ OVars_0 \end{array}}$$

Casting

$$\frac{Env \vdash Expr_1 \Rightarrow Paths_1\ \texttt{using}\ UPaths_1\ \texttt{with}\ OVars_1}{Env \vdash \texttt{cast as}\ Type\ \ Expr_1 \Rightarrow Paths_1\ \texttt{using}\ UPaths_1\ \texttt{with}\ OVars_1}$$

Constructors

Constructed nodes can never be optimized as they never return the empty sequence: even if the expression used in the constructor evaluates to empty, a node will be created.

$$\frac{Env \vdash Expr_1 \Rightarrow Paths_1\ \texttt{using}\ UPaths_1\ \texttt{with}\ OVars_1}{Env \vdash \texttt{(element}|\texttt{attribute)}\ QName\ \ Expr_1 \Rightarrow \{\}\ \texttt{using}\ Paths_1\# \cup UPaths_1\ \texttt{with}\ \text{EmptyVars}}$$

$$\frac{Env \vdash Expr_1 \Rightarrow Paths_1 \texttt{ using } UPaths_1 \texttt{ with } OVars_1}{Env \vdash \texttt{(document) } Expr_1 \Rightarrow \{\} \texttt{ using } Paths_1\# \cup UPaths_1 \texttt{ with } \text{EmptyVars}}$$

$$\frac{}{Env \vdash \texttt{(text) } String \Rightarrow \{\} \texttt{ using } \{\} \texttt{ with } \text{EmptyVars}}$$

Functions

In the absence of more detailed analysis, functions are never optimized. Their `OptimVars` are then always equal to `Emptyvars`.

5 Loading Algorithm

This section describes the loading algorithm used to create a projected document from an original XML document and a set of projection paths. The original document is parsed using a SAX API [29]. For this discussion, we only consider the following SAX events:

$$\begin{array}{lcl} SAXEvent & ::= & \texttt{Characters}\,(String) \\ & | & \texttt{OpeningTag}\,(QName) \\ & | & \texttt{ClosingTag} \end{array}$$

The loading algorithm operates in a left-deep recursive fashion. It takes a set of projection paths as input and operates on a stream of SAX events returned by the parser. As the SAX events are being processed, the algorithm maintains a set of paths to apply to the current XML document node. For each node, the algorithm decides on one of four actions to apply:

- Skip the node and its subtree (`Skip`);
- Keep the node and its subtree (`KeepSubtree`);
- Keep the node without its subtree (`Keep`);
- Keep processing the paths (`Move`).

The loading algorithm is illustrated on Figure 3 for a set two projection paths: `/a/b/c#` and `/a/d`, over the following document fragment:

```
<a>
  <g><b></b></g>
  <b>
    <c><f></f></c>
  </b>
  <d><e></e></d>
  <b></b>
  <c></c>
<a>
```

The loading algorithm processes one SAX event at a time, and maintains a set of current paths, corresponding to the parts of the original projection paths that apply to the current node. Note that nodes are only loaded (if needed) when their `ClosingTag` tokens are encountered, i.e., after all of their children have been processed. In the first step shown on Figure 3, the processed token is `OpeningTag(a)` (or `<a>`). The loading algorithm's current node is `<a>`, which is the first node for both projection paths. Given this and the projection paths information, the algorithm only needs to load descendants of the current node `<a>` that can be accessed through the *current paths*: `/b/c#` and `/d`. The loading algorithm then recursively processes the stream, loading children before their parents. When a projection path is verified, the corresponding node is loaded (with its subtree in case the # flag is present). On Figure 3, the node `<c>` that is a descendant of `/a/b` is loaded with its subtree `<f></f>` as specified by the projection path `/a/b/c#`. In contrast, the node `<e>` is not kept in the projected document. When a node does not verify a projection path, its entire subtree is skipped, i.e., the loading algorithm ignores the corresponding SAX tokens until corresponding closing tag is encountered (e.g., node `<g>` on Figure 3).

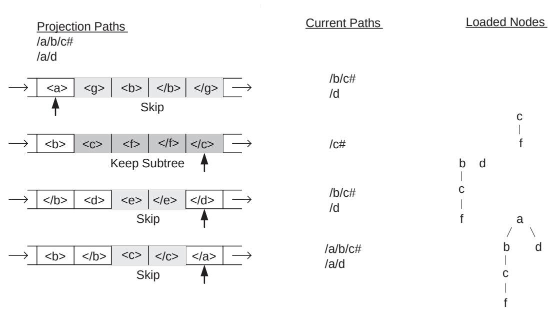


Figure 3: Loading algorithm

Dealing with the `descendant` axis significantly complicates the loading algorithm, since it might result in one projection path spawning into two new projection paths when moving down the tree. For example, consider the paths expression `//a`, which is expanded as `/descendant-or-self::node()/a`. Assuming the current node is an element `b`, both the path `/a`, since we might have a node `a` that is a child of `b` (`self::`), and the original path `/descendant-or-self::node()/a` can lead to nodes verifying the path expression. For this reason, the set of current paths the algorithm maintains can become larger during the loading process.

6 Experimental Evaluation

Experiments were run on a modified version of Galax [17] in which we implemented projection. We performed several kinds of experiments which were selected to evaluate the following aspects of projection:

Correctness: We used Galax regression tests to check that the implementation of our projection algorithm

Configuration	*CPU*	*Cache size*	*RAM*
A	1GHz	256Kb	256Mb
B	550MHz	512Kb	768Mb
C	1.4GHz	256Kb	2Gb

Table 3: Hardware configurations

does preserve the semantics of the query. (Section 6.1.)

Effectiveness: Projection is effective for a large family of queries. We evaluate the relative size of the projected document using using the XMark [31] benchmark, as well as queries over the XML version of DBLP. (Section 6.2.)

Maximal document size: As expected, projection allows to process queries on much larger documents than was previously possible (Section 6.3.)

Processing time: Measures of the evaluation time before and after projection show that projection also improves run-time performances. (Section 6.4.)

In order to understand the effect of projection on different hardware configurations, we used three different machines with varying CPU speed and RAM size. The first configuration (A) is a modern IBM laptop with 256M memory and a 1GHz CPU. The second configuration (B) is a desktop PC with more memory but a slower CPU. Finally, configuration (C) is a high-end server with a large 2Gb memory and a fast CPU. All three machines were running RedHat Linux. Configurations (A), (B) and (C) are summarized on Table 3

6.1 Correctness

Before evaluating the performance of the projection technique, we used Galax infrastructure for regression tests to check that the implementation of our projection algorithm is indeed working correctly. Galax regression tests are composed of a large number of queries, each with its corresponding expected result according to the XQuery semantics. A simple perl script runs all queries using the Galax interpretor, and verifies that the actual result returned by the interpretor matches the expected result. The set of tests contains more than 1000 queries which include simple atomic tests, the set of XQuery use cases, and queries from additional sources including queries from the XMark benchmark. We run those regression tests using Galax without projection, with projection, and with optimized projection to confirm that projection preserves the original semantics of each query. The regression tests and the corresponding scripts come with the Galax code itself and can be downloaded at [17].

6.2 Effectiveness

A second set of experiments was conducted to evaluate the actual reduction of memory usage for a various queries. We present experiments on all the XMark benchmark queries, and on queries over a real document, namely the XML version of the DBLP database[6].

6.2.1 XMark Queries

The XMark benchmark [31] contains of a broad range of queries, including simple lookups, joins, aggregations, queries with long path traversals, and publishing queries. XMark queries run over a single document about auctions. XMark comes with a document generator that can create auction documents of any size and can be downloaded from the XMark Web site[7].

For this experiment, we generated documents of varying sizes (from 500Kb to 2Gb) and run the 20 XMark queries on documents of increasing size for the three configurations. We then compared the size of the projected document against the size of the original document: as expected projection results in similar relative improvement for all sizes. Figures in the rest of the section report on the evaluation of all XMark Queries on a 50Mb document over Configuration C (Table 3).

Projected document size in file: Figure 4 shows the size of the projected documents as a percentage of the size of the original document. We report results for both versions of the projection. The projected document is less than 5% of the size of the document for most of the queries. On Query 19, *Projection* only reduces the size of the document by 40%, and it has no effects for Queries 6, 7, and 14. In contrast, *Optimized Projection* results in projected documents of at most 5% of the document for all queries but Query 14 (33%). The reason for this difference is that Queries 6, 7, 14 and 19 evaluate descendant-or-self() (//) path expressions for which projection without optimization performs poorly. Query 14 is a special case since it selects a large fragment of the original auction document. Obviously projection cannot perform as well for this kind of query.

Projected document size in memory: Figure 5 shows the memory used by the query processor for the projected document as a percentage of the memory used. Memory usage for the projected document is consistent with the relative size of the projected documents on file. Projected documents tend to use relatively slightly more memory than their size, due to some overhead in the XML data model representation.

6.2.2 DBLP Queries

The DBLP document contains a bibliography of over 325,000 publications. Its size stored as text is 144Mb. The schema of the DBLP document is very simple and result in shallow trees, therefore we could not evaluate complicated queries, such as queries with descendant axis, on it. We considered two queries. The

[6]`http://dblp.uni-trier.de/xml/`

[7]`http://monetdb.cwi.nl/xml/downloads.html`

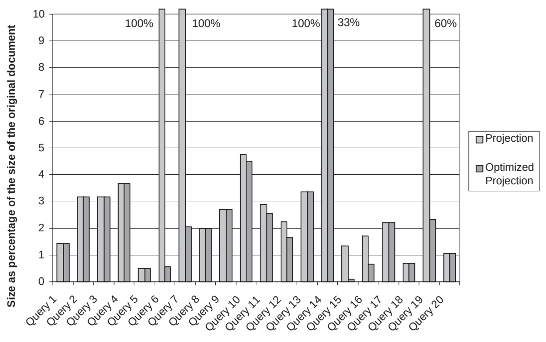


Figure 4: Projected documents size as a percentage of the size of the original documents

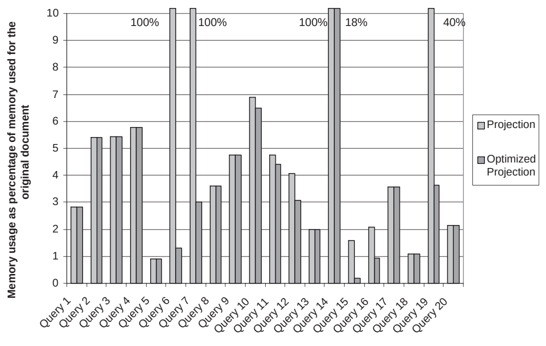


Figure 5: Projected document memory usage

first query asks for the titles of the books written by Jim Gray. This query is very selective in terms of the projection, as only 0.25% of the publications in the document are books.

DBLP Query 1

```
for $a in $dblp/dblp/book
where $a/author/text()="Jim Gray"
return $a/title/text()
```

The second query asks for the titles of the journal articles written by Jim Gray. This query is not as selective in terms of the projection, as 35% of the publications in the document are journal articles.

DBLP Query 2

```
for $a in $dblp/dblp/articles
where $a/author/text()="Jim Gray"
return $a/title/text()
```

We run the two DBLP queries on Configuration C (see Table 3). We were not able to load the complete document in memory without projection. Therefore, we report only on the memory needed for DBLP Queries 1 and 2 in Table 4 for *Projection* and *Optimized Projection*.

Query	*Projection*	*Optimized Projection*
DBLP Query 1	0.85Mb	0.76Mb
DBLP Query 2	97Mb	84Mb

Table 4: Projection on a real data XML document.

Configuration		*A*	*B*	*C*
Query 3	NoProj	33Mb	220Mb	520Mb
	OptimProj	1Gb	1.5Gb	1.5Gb
Query 14	NoProj	20Mb	20Mb	20Mb
	OptimProj	100Mb	100Mb	100Mb
Query 15	NoProj	33Mb	220Mb	520Mb
	OptimProj	1Gb	2Gb	2Gb

Table 5: Document size limits for three XMark Queries with or without projection

6.3 Maximal Document Size

The main objective of projection is to overcome the strong memory limitations that were reported in the introduction. We compare the size of the largest document we were able to process without projection, and with optimized projection on our three hardware configurations.

Table 5 gives for three XMark queries (3, 14 and 15) the size of the largest document for which we could evaluate the query, with or without optimized projection. We selected these three query as they result in different decreases in the document size (Figure 4), from 82% (Query 14) to 99.9% (Query 15). We see that our projection approach makes it possible to evaluate queries on significantly larger documents (up to 30 times larger for Query 15) than without any projection.

6.4 Processing Time

Finally, we study the impact of projection on processing time. We now show that: (a) projection does not have a significant impact on parsing and loading time, and (b) it reduces, sometimes significantly, query evaluation time.

Parsing and Loading time: Figure 6 shows the impact of path analysis for projection on the parsing and loading time of the query (path analysis time included). For most queries, the path analysis does not slow down document loading, but actually speeds it up. This might look surprising, but can be explained by the fact that less nodes have to be created in the document data model. However, for queries that contain descendant-or-self() axis, loading is more expensive with *Projection*, due to the more complex computation required during loading. The queries for which *Projection* result in high loading times are actually the ones

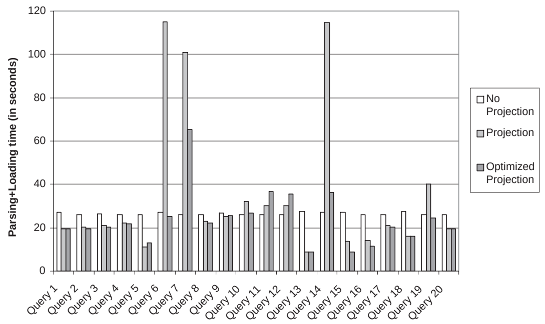


Figure 6: Parsing and loading time, in seconds

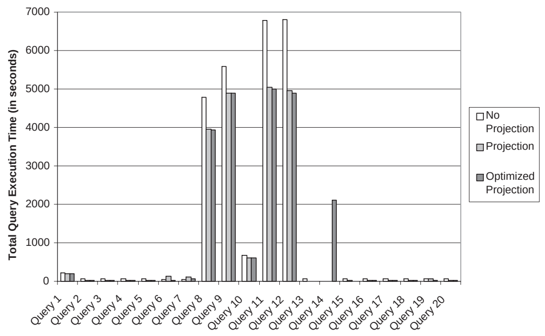


Figure 7: Query execution time, in seconds

for which *Projection* does not perform well in terms of memory reduction. For these queries, loading with *Optimized Projection* is still a little more expensive than without any projection, but it is much faster than with *Projection*, and results in decreased memory usage.

Query Execution time: Figure 7 shows the impact of projection on query execution time. Projection actually speeds up query processing. The reason is that query evaluation has less unnecessary nodes to process since those have been discarded in advance by the loader. Some XMark queries are very expensive to evaluate, because of expensive join operations. For these queries, while projection still speed up processing, query processing time is dominated by the cost of the join. Note that for query 14, the reason why the optimized projection seem more expensive is only because the query fails without projection, therefore the figures for query execution show up as zero.

As a conclusion, we see that *Optimized Projection* results in significant savings in memory usage (more than 95% for all queries but one), and does not increase document parsing and loading time significantly. In fact, for most queries, parsing and loading are actually faster when *Optimized Projection* is applied. Additionally, *Optimized Projection* results in lower query execution times for all XMark queries.

7 Related Work

Projection operations have been proposed in previous algebras for XML and for semistructured data. The TAX tree algebra for XML [18] includes a projection operator, which differs from ours in that it supports omitting intermediate nodes while we require to keep all nodes from the root of the document, and it only support simple wilcards * while we support all XPath node tests. The SAL algebra [5] has a quite different projection operation based on regular-expressions. Both work focus on the expressiveness of the projection operation, while our notion of projection is simpler but designed to support an efficient physical implementation on XML files and streams.

Our loading algorithm has some similarity with work on filtering XML documents [1, 9]. However, they focus on processing efficiently subsets of XPath without building intermediate data structures, while we support the construction of a data model instance that can be used to process arbitrary XQuery expressions.

Finally, we have studied the impact of projection in isolation from other optimization techniques. However, we believe work on XML indexes [11, 19] and XML joins [8, 10] could be used in conjunction with projection.

8 Conclusion

In this paper, we have presented projection techniques that can be used to support main-memory XQuery evaluation over large XML documents. The main contribution of the paper is a path analysis technique that infers the set of paths used for an arbitrary XQuery expression. Our experiments show that this technique can be used to evaluate queries on files up to two Gigabyte even on machines with limited memory. Our implementation is fully functional and available for download on the Web at [17]. As future work, we plan to work on a tighter integration between the query evaluation and the loading, which are currently done in separate steps, and investigate methods to quantify the precision of our projection algorithm compared to the *optimal* projection. Finally, we believe the techniques presented here should be integrated with other forms of optimization, including XML join algorithms and query rewritings.

Acknowledgments. We want to thank Sihem Amer-Yahia, Irini Fundulaki, and the Database Group at Columbia University for their feedback on earlier versions of the paper.

References

[1] M. Ahmet and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2000.

[2] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Querying XML sources using an ontology-based mediator. In *Proceedings of International Conference on Cooperative Information Systems (CoopIS)*, pages 429–448, Irvine, California, Oct. 2002.

[3] S. Amer-Yahia and P. Case. XQuery and XPath full-text use cases. W3C Working Draft, Feb. 2003.

[4] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1 specification. W3C Recommendation, Oct. 1998.

[5] C. Beeri and Y. Tzaban. SAL: An algebrar for semistructured data and XML. In *International Workshop on the Web and Databases (WebDB'99)*, Philadelphia, Pennsylvania, June 1999.

[6] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2002.

[7] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. W3C Recommendation, Jan. 1999.

[8] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2002.

[9] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2002.

[10] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Hong Kong, China, Aug. 2002.

[11] C. Chun, J. Min, and K. Shim. Apex: An adaptative path index for XML data. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2001.

[12] D. K. Daniela Florescu, Andreas Grünhagen. XL: an XML programming language for web service specification and composition. In *Proceedings of International World Wide Web Conference*, pages 65–76, May 2002.

[13] XQuery 1.0 and XPath 2.0 data model. W3C Working Draft, Nov. 2002.

[14] Enosys software. `http://www.enosys.com/`.

[15] P. Fankhauser, T. Groh, and S. Overhage. Xquery by the book: The ipsi xquery demonstrator. In *Proceedings of the International Conference on Extending Database Technology*, 2002.

[16] XQEngine. `http://www.fatdog.com/`.

[17] Galax: An implementation of xquery.
`http://db.bell-labs.com/galax/optimization/`.

[18] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proceedings of International Workshop on Database Programming Languages*, 2001.

[19] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2002.

[20] C. Minoux. Kweelt backend, July 2001.
`http://cheops.cis.upenn.edu/`
`~sahuguet/PUB/X98/cyril_minoux.ps.gz`.

[21] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[22] Nimble technology. `http://www.nimble.com/`.

[23] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *In Proceedings of Workshop on XML-Based Data Management (XMLDM) at EDBT 2002*, LNCS 2490, Prague, Mar. 2002. Springer-Verlag.

[24] P. F. Patel-Schneider and J. Siméon. The Yin/Yang web: XML syntax and RDF semantics. In *Proceedings of International World Wide Web Conference*, pages 443–453, May 2002.

[25] Quip. `developer.softwareag.com/tamino/quip`.

[26] J. Robie. The syntactic web: Syntax and semantics on the web. In *XML'2001*, Orlando, Florida, Dec. 2001.

[27] M. Rys. State-of-the-art XML support in RDBMS: Microsoft SQL Server's XML features. *Bulletin of the Technical Committee on Data Engineering*, 24(2):3–11, June 2001.

[28] A. Sahuguet, L. Dupont, and T.-L. Nguyen. Kweelt.
`http://kweelt.sourceforge.net/`.

[29] Simple API for XML. `http://www.saxproject.org/`.

[30] Saxon. `http://saxon.sourceforge.net/`.

[31] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 974–985, Hong Kong, China, Aug. 2002. `http://monetdb.cwi.nl/xml/`.

[32] I. Sosnoski Software Solutions. Java XML models benchmark. `www.sosnoski.com/opensrc/xmlbench`.

[33] Xalan. `http://xml.apache.org/xalan-j/`.

[34] XPath 2.0. W3C Working Draft, Nov. 2002.

[35] XQRL, Inc. `http://www.xqrl.com/`.

[36] XQuery 1.0: An XML query language. W3C Working Draft, Nov. 2002.

[37] XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, Nov. 2002.

[38] Oracle XQuery prototype: Querying XML the XQuery way. `http://technet.oracle.com/tech/xml/xmldb`.