Pruning Nested XQuery Queries

Bilel Gueni, Talel Abdessalem, Bogdan Cautis
LTCI - TELECOM ParisTech
Paris – France
First.Last@Telecom-ParisTech.fr

Emmanuel Waller
LRI - Université de Paris-Sud
Orsay – France
First.Last@lri.fr

ABSTRACT

We present in this paper an approach for XQuery optimization that exploits minimization opportunities raised in composition-style nesting of queries. More precisely, we consider the simplification of XQuery queries in which the intermediate result constructed by a subexpression is queried by another subexpression. Based on a large subset of XQuery, we describe a rule-based algorithm that recursively prunes query expressions, eliminating useless intermediate results. Our algorithm takes as input an XQuery expression that may have navigation within its subexpressions and outputs a simplified, equivalent XQuery expression, and is thus readily usable as an optimization module in any existing XQuery processor. We demonstrate by experiments the impact of our rewriting approach on query evaluation costs and we prove formally its correctness.

Categories and Subject Descriptors: H.2 [Information Systems]: Database Management—*Query Language*

General Terms: Algorithms, Language, Performance

Keywords: XML, XQuery, query rewriting

1. INTRODUCTION

XML is by now the de facto standard format for data exchange on the Web. It is also used as a data model for native XML databases and as a common language in systems that integrate data coming from heterogenous sources. It is thus essential to have effective and efficient tools for querying and manipulating XML data. Consequently, query languages such as XPath and XQuery have been receiving a great deal of attention from the research community lately. And, unsurprisingly, query optimization, one of the most important (and most studied) topics in relational databases, has seen a revival in the semi-structured context.

The XQuery language plays a key role in XML data management and has many powerful features such as nesting and composition of *for-let-where-return (FLWR)* query blocks, the construction of hierarchical XML results and the navigation in documents by means of XPath expressions. Unfortunately, its expressive power and operational semantics make the reasoning about query optimization quite difficult and have been the main obstacles in establishing a comprehensive framework for query optimization, although significant progress has been made in this direction.

We study in this paper a novel aspect of XQuery optimization that exploits minimization opportunities raised in a composition-style nesting of XQuery queries. More precisely, we consider the simplification of XQuery expressions in which the intermediate results constructed by a subexpression is queried by another subexpression. In other words, given an XQuery expression with navigation over some documents, we consider a setting in which some of these documents may in fact be *intentional*, defined as the result of other XQuery subexpressions. Our approach is similar in spirit to the one of Marian and Simeon [16], of projecting XML documents w.r.t. a given XQuery query. Instead of XML documents, we project XQuery subexpressions with respect to other subexpressions querying them.

This kind of composition is common in many scenarios of data exchange, mediation or integration, or in view-based security. Before discussing in more detail these scenarios, and several others, let us first illustrate the problem we study and the main challenges by a data integration example. The example deals with the reformulation of queries over heterogenous, interconnected sources.

EXAMPLE 1.1. *Our example is based on the XMark benchmark data [20]. Let us consider three interconnected XML sources S_1, S_2 and S_3, with S_1 being somehow complemented by S_2 and S_3: S_1 integrates data coming from these two sources under a unified schema that can be transparently queried by users. To this end, mappings between the schema of S_1 and the ones of S_2 and S_3 are defined by means of transformation XQuery queries, Q_2 and Q_3, as follows*[1]*:*

Q_2 :
```
<site>
{for $i in (docA@S2/site)
 where $i/people/person/@id = "X"
 return ($i/open_auctions/open_auction,
         $i/closed_auctions/closed_auction,
         $i/people/person)
}
</site>
```
Q_3 :
```
<site>
{let $l := for $i in docB@S3/site/closed_auctions/closed_auction
           where ($i/itemref/@item = "car" or
                  $i/buyer/@id = "X") and $i/seller/@id = "Y"
           return $i
 return $l
}
</site>
```

The query Q_2 returns all open_auction, closed_auction and person data from the sites containing a person identified by "X". The query Q_3 computes the sequence of the closed_auction elements having

[1]We use $doc@S_i$ as short notation for a document URL at S_i.

CIKM'08, October 26–30, 2008, Napa Valley, California, USA.

either a buyer identified by "X" or an item car, and a seller identified by "Y". In both queries, the result is wrapped in a site element.

In this scenario, the role of Q_2 and Q_3 is to define the relationship between S_1, on one hand, and S_2 and S_3, on the other hand. They formulate a transfer that is only virtual and the data remains at the sources S_2 and S_3. Moreover, the source S_1 may also have its own data. Intuitively, S_1 could be defined by a virtual document "$source_1 := docC@S_1 \cup Q_2 \cup Q_3$" having an extensional component (for S_1's own data) and two intentional ones (for the data residing at S_2 and S_3).

Let us consider now the following user query Q_1, specified over $source_1$, which returns the open_auction elements that have some person data in common with another document, $docD@S_1$:

Q_1 :
```
for $j in source1
return
  for $k in docD@S1/site
  where $j/person = $k/people/person
  return
    <common-auction>{$j/open_auction}</common-auction>
```

S_1's wrapper module would have no difficulty in executing Q_1 over the extensional part of $source_1$. For the intentional ones, there are two possible approaches: (a) Q_2 and Q_3 are executed at S_2 and S_3, their results are transferred to S_1 and then Q_1 is evaluated over them, or (b) Q_1 is "pushed" to both S_2 and S_3, which evaluate it locally over their respective transformation queries and send back to S_1 the result. Unsurprisingly, the latter approach can have significant advantages, especially when Q_1 uses only a small portion from the output of the transformation queries.

Now, by the second approach, S_2's wrapper module has to execute Q_1 over Q_2. This could be done by first evaluating Q_2, then evaluating Q_1 over the intermediate result. But since XQuery is compositional, it is more preferable to interpret this step as a single XQuery expression $Q_{1.2} = Q_1 \circ Q_2$, by simply substituting the virtual variable by its XQuery definition. In this way, the query optimizer module can chose the best execution strategy. For instance, $Q_{1.2}$ can be considered as the following XQuery expression:

$Q_{1.2}$:
```
for $j in (the definition of Q2)
return
  for $k in docD@S1/site
  where $j/person = $k/people/person
  return
    <common-auction>{$j/open_auction}</common-auction>
```

At this point, instead of the straightforward execution plan, an efficient query optimizer module should detect that Q_2 is only partially useful in $Q_{1.2}$, since only open_auction and person elements are queried. Hence the following equivalent yet less expensive query can be executed instead:

$Q'_{1.2}$:
```
for $j in (<site>
            {for $i in (doc2@S2/site)
              where $i/people/person/@id = "X"
              return
                ($i/open_auctions/open_auction, $i/people/person)
            }</site>)
return
  for $k in doc1@S1/site
  where $j/person = $k/people/person
  return
    <common-auction>{$j/open_auction}</common-auction>
```

In the case of S_3, the efficient optimizer would have even greater impact, since the equivalent yet simplified query should in fact replace the content of Q_3's site element by the empty sequence, because only closed_auction elements are outputted in the Q_3's site element :

$Q'_{1.3}$:
```
for $j in <site>{()}</site>
return
  for $k in doc1@S1/site
  where $j/person = $k/people/person
  return
    <common-auction>{$j/open_auction}</common-auction>
```

Our contribution. We study in this paper the simplification of queries that have a composition-style nesting as the one illustrated in Example 1.1. We adopt a static-analysis approach, based on detecting and projecting out the useless parts in subexpressions, keeping only what is needed in order to compute the end result. This task is made difficult by potentially complex relationships between the query blocks. We describe a set of rewrite rules that apply such pruning steps recursively over the blocks of an XQuery query, not only at the uppermost level but at any nesting level in the query. Each rule application will output a strictly simpler (i.e., with less navigation steps) yet equivalent XQuery expression. Our rule-based algorithm applies to a large subset of XQuery and we formally prove its correctness. Importantly, the algorithm takes as input an XQuery that may have navigation within subexpressions and outputs a simplified, equivalent XQuery expression. It can be thus easily plugged as an optimization module in any existing XQuery processor. We implemented and tested our rewriting approach on top of the Galax engine and we demonstrate by experiments its impact on query evaluation costs.

In the remaining of this section we further motivate our work and we discuss related research. Queries with this composition-style of nesting are very useful in practice. Transformation XQuery queries for mapping between heterogenous XML sources in integration and mediation scenarios are quite common [12, 22, 1]. The Clio project [12] provides a graphical editor for defining schema mapping definition, generating complex XSLT or XQuery transformations. In peer-to-peer settings, such as the Piazza PDMS [22], a peer can refer to data hold by another peer by means of an XQuery mapping. In this setting, it is crucial to minimize the amount of actual data that is transferred between peers. The Active XML system [1] introduces a flexible framework for peer-to-peer XML integration, by combining in one *active* document materialized (extensional) XML parts with intentional parts defined by calls to Web services. Importantly these services can be defined by XQuery expressions and evaluating a query over an active document amounts essentially to query pushing and composition.

Another important use is in queries posed on security views. In many applications that rely on sensitive data, like medical or juridic applications, access to XML documents may be granted only by querying views over these documents. The views define what data the user can access, and the system may accept only queries formulated over these views. It can either evaluate the global query (i.e., the composition of user query and the views) or can first materialize the views and then evaluate the user query. Obviously, in the case of a large number of views, materializing and maintaining these views can be too costly.

It is also common to cache and reuse the *definitions* of queries but not necessarily their results. This can for instance guide inexperienced users, allowing them to query XQuery expressions that are already available and well-understood. Finally, our simplification technique can be used to optimize automatically generated queries, e.g. for graphical editors in the style of query-by-example.

Related work. Several works on XQuery processing and optimization adopt an approach based on rewrite rules. In [15, 18, 17, 10, 19, 21], the authors discuss various rules for XQuery normalization or for transformation tasks such as XQuery-to-SQL translation, elimination of unnecessary ordering operations or introduc-

tion of a tree-pattern operator in query plans. These approaches are orthogonal to the query simplification technique presented here. [3] introduces rewrite rules for reducing the number of nesting levels in XQuery expressions but does not consider the elimination of useless navigation and result construction. In [5], the authors introduce a logical framework for optimization in the OptXQuery subset of XQuery, the Nested XML Tableaux. They present a set of rewrite rules for normalization and grouping of repeated navigation steps within a query by means of a group-by operator.

More germane to this work is [16], which introduces XML document projection for query optimization. It gives a set of rewrite rules for the following task: starting from an XQuery expression Q over a document D, identify and project out the parts of D that are not useful for the evaluation of Q. This is very effective to reduce in-memory computations such as node construction. The technique was later refined and extended to take into account the schema of the document in [2]. Although very close in spirit, our approach subsumes the idea of [16] of projecting XML documents, as we consider the projection-based simplification of arbitrary XQuery blocks, and not only plain XML documents. In this context, special attention has to be paid to preserving query equivalence.

In [22, 6], the authors consider the minimization of queries obtained by following semantic paths (mappings) in the Piazza system. To this end, they study the complexity of query containment for a restricted XQuery flavor, that of conjunctive XML queries (c-XQueries). The role of composition in XQuery evaluation was considered in [13]. For an XQuery fragment strictly smaller than the one we consider here, a formal study of the computational complexity of XQuery without composition is provided. Moreover, [13] shows that, under restrictions, composition can be eliminated and describes a set of rewrite rules to this end.

A problem similar to ours was also studied in the context of publishing relation data in XML format, in projects such as XPeranto [4] and SilkRoute [8]. In Silkroute, the composition of XQuery expressions represented by so called *view forests* over relational sources was considered, where a view forest is a mix of XML structure and SQL expressions representing XQuery-to-SQL translations. These techniques are specific to the XML-over-relational setting and do not transfer to XQuery minimization.

The paper is organized as follows. In Section 2, we present preliminary notions. Section 3 details our rule-based algorithm and some extensions of the algorithm are presented in Section 4. In Section 5 we describe the experimental analysis we have conducted. We conclude in Section 6.

2. PRELIMINARIES

We describe in this section the data model and XQuery expressions we consider, as well as additional assumptions.

Data model. For the sake of simplicity we present our techniques using a slightly simplified version of the XQuery data model. We consider an XML document as an unranked rooted tree t modeled by a set of edges $\text{EDGES}(t)$, a set of nodes $\text{NODES}(t)$, a distinguished root node $\text{ROOT}(t)$, a labeling function over nodes λ_t assigning to each node a label (or text value) from an infinite alphabet Σ, and a typing function τ_t assigning to each node one of the following kinds: *{document, element, text}*. The *document* type can only be given to the root of the XML document and *text* nodes can only appear as leaves. This simplified model can be extended in straightforward manner to other components of the XQuery data model such as attributes.

XQuery fragment. We focus our study on a significant subset of XQuery, described by the grammar of Figure 1.

$$
\begin{array}{lcl}
exp & := & () \\
 & & \mid literal \\
 & & \mid exp, exp \\
 & & \mid exp\ Op\ exp \\
 & & \mid Path \\
 & & \mid (forClause \mid letClause)+ \\
 & & \ (where\ exp)?\ return\ exp \\
 & & \mid (some \mid every)\ \$QName\ in\ exp\ return\ exp \\
 & & \mid if(exp)\ then\ exp\ else\ exp \\
 & & \mid <QName>\{exp\}</QName> \\
 & & \mid element\{QName\}\{exp\} \\
forClause & := & for\ \$QName\ in\ exp\ (,\$QName\ in\ exp)* \\
letClause & := & let\ \$QName := exp\ (,\$QName\ :=\ exp)* \\
Path & := & (doc(uri) \mid \$QName)(/Step)* \mid exp/Step \\
Op & := & < \mid > \mid = \mid + \mid - \mid * \mid << \mid >> \mid \text{“}is\text{”} \\
Step & := & NodeTest(/Step)? \mid text() \\
NodeTest & := & QName \mid \text{”}*\text{”}
\end{array}
$$

Figure 1: *The XQuery fragment*

This grammar captures the main XQuery constructs used in practice, such as literal values, sequence construction, variables, FLWR blocks, conditionals, quantifiers, comparisons operators, logical or arithmetic operations, element constructions. For clarity and space reasons, we consider in this paper XPath navigation only along the `child axis` (/). Extensions to other navigation axis such as attribute (/@) and descendant (//) are presented in an extended version of this work [11]. We also ignore path qualifiers, which can always be reformulated away using *where* clauses.

XQuery normalization. Before applying our technique for query simplification, we assume that some of the standard normalization steps, usually employed to reduce XQuery expressions to equivalent expressions in the simpler language XQuery Core [7], are first applied. This normalization phase will allow us to present our inference algorithm based on a uniform syntactic formulation. We give in Figure 2 the set of normalization rules we consider, each of them being self-explanatory. In short, they either facilitate the extraction of XPath expressions referencing a variable or reformulate nested expressions in order to have one variable per clause.

$$
\begin{array}{c}
e/step_1/\ldots/step_n \Rightarrow let\ \$v := e\ return\ \$v/step_1/\ldots/step_n \\
for\ \$v_1\ in\ e_1, \ldots, \$v_n\ in\ e_n\ return\ e \Rightarrow \\
for\ \$v_1\ in\ e_1\ return\ for \ldots\ for\ \$v_n\ in\ e_n\ return\ e \\
let\ \$v_1 := e_1, \ldots, \$v_n := e_n\ return\ e \Rightarrow \\
let\ \$v_1 := e_1\ return\ let \ldots\ let\ \$v_n := e_n\ return\ e \\
some\ \$v_1\ in\ e_1, \ldots, \$v_n\ in\ e_n\ satisfies\ e \Rightarrow \\
some\ \$v_1\ in\ e_1\ satisfies\ some \ldots\ some\ \$v_n\ in\ e_n\ satisfies\ e \\
every\ \$v_1\ in\ e_1, \ldots, \$v_n\ in\ e_n\ satisfies\ e \Rightarrow \\
every\ \$v_1\ in\ e_1\ satisfies\ every \ldots\ every\ \$v_n\ in\ e_n\ satisfies\ e \\
<QName>\{e\}</QName> \Rightarrow element\{QName\}\{e\}
\end{array}
$$

Figure 2: *Normalization rules*

Inference rules notation and environment. We present our algorithm via a set of inference rules, and we adopt standard programming languages notation similar to the one used in [16]. Inference rules are based on *judgements*, which denote statements of the form: $Env \vdash f(p_1, \ldots, p_n) \Rightarrow res$.

Such a statement reads as follows: *the judgement holds iff in the environment Env, by calling the function f with parameters $p_1, \ldots, p_n$ we obtain the result res.*

Inference rules are represented as follows:

$$\frac{premise_1 \ldots premise_n}{Env \vdash f(p_1, \ldots, p_n) \Rightarrow res}$$

where each premise is a judgement. Such a rule reads as follows: *the judgement $Env \vdash f(p_1, \ldots, p_n) \Rightarrow res$ holds if the premises $premise_1 \ldots premise_n$ hold.* The functions we consider in our inference rules will be defined in Section 3.

In XQuery, any variable $\$v$ is associated to a subexpression, by either $\$v$ *in exp* or $\$v$:= *exp*, in this way being bound to the intermediate XML values returned by the subexpression.

EXAMPLE 2.1. *For instance, in the query Q_2 of the running example, variable $\$i$ is bound to elements produced by the XPath docA@S2/site. Similarly, in the query Q_3, variable $\$l$ is bound to some elements produced by variable $\$i$. This is because the FLWR block to which $\$l$ is bound returns elements over which $\$i$ iterates, those that satisfy certain conditions. In $Q_{1,2}$, the variable $\$j$ is bound to a constructed site element wrapping some content returned by Q_2's FLWR expression.*

For a variable $\$v$, by the *bound expression* associated with $\$v$ (in short, $exp_b(\$v)$) we denote the expression *exp* appearing in the statement declaring $\$v$, be it for $\$v$ *in exp*, *let* $\$v := exp$ or *some* $\$v$ *in exp*. By the *return expression* of $\$v$ (in short, $exp_r(\$v)$) we denote the associated *where*, *return* or $satisfies$ parts.

In the presentation of our rule-based algorithm, we will rely on a memory space (denoted *environment*) that records for each variable the expressions that produce the intermediate results to which it is bound. The environment will contain a set of mappings from variables to sets of objects. Formally, this is written $\$v \Rightarrow \{o_1, \ldots, o_m\}$. We distinguish three possible kinds of such objects: (i) results of an XPath expression (represented in the environment by the XPath expression itself), (ii) element constructors with some element content (can be any XQuery subexpression), (iii) text values (denoted simply `#text`).

Going back to the example, we write $\$i \Rightarrow \{docA@S2/site\}$ for Q_2, $\$i \Rightarrow \{docB@S3/site/closed\_auctions/closed\_auction\}$ and $\$l \Rightarrow \{\$i\}$ for Q_3, or $\$j \Rightarrow \{<site>\ldots</site>\}$ for $Q_{1.2}$.

For the construction of the environment, we determine by a static analysis for each variable the objects returned as intermediate XML values by its bound expression. This is done using the function $varRes()$, which infers the output kind of a subexpression by the following exhaustive and straightforward case analysis:

$varRes(for\ \$v\ in\ e_1\ (where\ e_2)?\ return\ e_3) \Rightarrow varRes(e_3)$
$varRes(let\ \$v := e_1\ (where\ e_2)?\ return\ e_3) \Rightarrow varRes(e_3)$
$varRes(if(e_1)\ then\ e_2\ else\ e_3) \Rightarrow varRes(e_2) \cup varRes(e_3)$
$varRes(e_1, \ldots, e_n) \Rightarrow varRes(e_1) \cup \ldots \cup varRes(e_n)$
$varRes(step_1/\ldots/step_2) \Rightarrow \{step_1/\ldots/step_2\}$
$varRes(element\{QName\}\{e\}) \Rightarrow \{element\{QName\}\{e\}\}$
$varRes(\$v) \Rightarrow \{\$v\}$
$varRes(literal) \Rightarrow \{\texttt{\#text}\}$
$varRes(some\ \$v\ in\ e_1\ satisfies\ e_2) \Rightarrow \{\texttt{\#text}\}$
$varRes(every\ \$v\ in\ e_1\ satisfies\ e_2) \Rightarrow \{\texttt{\#text}\}$

Since an XPath expression can be relative to a named variable (i.e., starting with a variable name), the environment will also allow us to keep track of the relationship between variables within the query (for example, the fact that $\$l$ is bound to $\$i$). For convenience, for the manipulation of the environment we also define a function called $saturate()$, which refines the bindings by making explicit all the XPath navigation.

EXAMPLE 2.2. *After obtaining by varRes that $\$l \Rightarrow \$i$ and $\$i \Rightarrow \{docB@S3/site/closed\_auctions/\ closed\_auction\}$, we can refine the information on variable $\$l$, using the saturate function, as $\$l \Rightarrow \{\$i, docB@S3/site/closed\_auctions/closed\_auction\}$.*

Finally, for a variable $\$v$ and its bound expression $exp_b(\$v)$, the addition of $\$v$ to the environment is performed in the inference rules by the following statement:

$$Env = +(\$v \Rightarrow (Env.saturate(varRes(exp_b(\$v))))).$$

For convenience, the following functions will also be used in the algorithm to access the pre-computed environment:

- $getBind(\$v)$: retrieves from the environment the set of objects associated to the input variable $\$v$.
- $getXPathBind(\$v)$: among the objects to which the variable $\$v$ is bound, it retrieves those corresponding to XPath expressions (if any exist).

3. THE REWRITING ALGORITHM

Overview. We start by giving an overview of our rule-based algorithm, which takes as input an XQuery expression Q and outputs an equivalent simplified XQuery expression Q'.

As the various bound expressions in Q compute intermediate results that may only be partially useful to Q's end result, our algorithm identifies and prunes their irrelevant parts. The output is an equivalent query Q' obtained from Q by substituting each subexpression $exp_b(\$v)$ by a subexpression $exp'_b(\$v)$ that has the advantage of computing only the needed intermediate results.

For a given variable $\$v$, the algorithm retrieves from $exp_r(\$v)$ all the XPath expressions that access the result of $exp_b(\$v)$. This task is performed by the *extractPaths* function. The paths are then used to retrieve and project out the useless parts in $exp_b(\$v)$. This is the role of the *projectPaths* function. A simpler subexpression $exp'_b(\$v)$ is obtained in this way.

This process is applied recursively, in bottom-up manner, by the *Prune* function over Q. More precisely, for a given variable $\$v$ in Q, the pruning is first applied recursively within its bound and return expressions, then for $\$v$ itself, as described above.

We continue the presentation of the algorithm, starting with the rule-based functions for path analysis (*extractPaths*) and query projection (*projectPaths*) (in Section 3.1 and Section 3.2 respectively). We wrap-up the presentation with the *Prune* function, that applies in bottom-up manner the steps for path extraction and projection (in Section 3.3). The section ends with the formal results on correctness (in Section 3.4).

3.1 Path analysis

The $extractPaths$ function takes as input a variable $\$v$ and its return expression $exp_r(\$v)$, analyses $exp_r(\$v)$ and extracts the paths that navigate through the variable $\$v$. These paths start with $\$v$ (either explicitly, or via other declared variables), and are denoted the *projection paths* of $\$v$.

Similar to [16], in our analysis we will distinguish between two kinds of projection paths: (i) *used paths* and (ii) *returned paths*. The former kind denotes paths for which the descendants of returned nodes are not necessarily relevant for the end result and no navigation in the subtrees of these nodes is required. These are the paths that simply bind a variable $\$v$, appearing only in its bound expression $exp_b(\$v)$.

The latter kind denotes paths for which descendants of the nodes reached by the path may be relevant and must be kept in the end result. Paths are by default considered of the *returned* kind, unless some conditions for the *used* kind are verified.

We now present the inference rules for the $extractPaths$ function. The result of each rule application will be two sets of paths, $\mathcal{P}$ and $\mathcal{P}^{\#}$, for the *used* and *returned paths* respectively.

Literal, empty sequence. When the input expression $exp_r(\$v)$ is a literal (rule $ep1$) or an empty sequence (rule $ep2$), no paths can be extracted.

$$\frac{}{Env \vdash extractPaths(\$v, literal) \Rightarrow \emptyset, \emptyset}\ (ep1)$$

$$\frac{}{Env \vdash extractPaths(\$v, ()) \Rightarrow \emptyset, \emptyset}\ (ep2)$$

Sequence, conditional, comparison, element construction. In this case (rules $ep3$ to $ep6$), the analysis of the input expression

amounts to analyzing its subexpressions, and the output sets $\mathcal{P}$ and $\mathcal{P}^{\#}$ are obtained from the union of the *used* and *returned paths* extracted from the subexpressions.

$$\frac{Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \quad Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#}}{Env \vdash extractPaths(\$v, (e_1, e_2)) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#}}\ (ep3)$$

$$\frac{\begin{array}{c} Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \\ Env \vdash extractPaths(\$v, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#} \end{array}}{\begin{array}{c} Env \vdash extractPaths(\$v, if\ e_1\ then\ e_2\ else\ e_3) \Rightarrow \\ \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#} \end{array}}\ (ep4)$$

$$\frac{Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \quad Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#}}{Env \vdash extractPaths(\$v, e_1\ Op\ e_2) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#}}\ (ep5)$$

$$\frac{Env \vdash extractPaths(\$v, exp) \Rightarrow \mathcal{P}, \mathcal{P}^{\#}}{Env \vdash extractPaths(\$v, element\{QName\}\{exp\}) \Rightarrow \mathcal{P}, \mathcal{P}^{\#}}\ (ep6)$$

Variable reference. When the input expression $exp_r(\$v)$ is a variable reference, we have two alternative cases. If $exp_r(\$v) = \$v$ (rule $ep7$), the output result is a *returned path* $\$v$ (a one step path). If $exp_r(\$v) = \$v'$, with $\$v' \neq \$v$, we extract from the XPath expressions bound to $\$v'$ those that are relative to $\$v$ (i.e., $\$v$ is their first step). These paths constitute the output set of *returned paths* $\mathcal{P}^{\#}$ (rule $ep8$). The output set of *used paths* $\mathcal{P}$ is empty in both cases.

$$\frac{}{Env \vdash extractPaths(\$v, \$v) \Rightarrow \emptyset, \{\$v\}}\ (ep7)$$

$$\frac{\$v' \neq \$v, \quad Env.getXPathBind(\$v') \Rightarrow B \quad P^{\#} = \{p \in B, p = \$v/\ldots\}}{Env \vdash extractPaths(\$v, \$v') \Rightarrow \emptyset, P^{\#}}\ (ep8)$$

XPath expression. Here, due to the normalization process described in Section 2, we have only three alternatives (rules $ep9$ to $ep11$). If the input expression $exp_r(\$v)$ is a path relative to $\$v$, then this path formes the output set of *returned paths* $\mathcal{P}^{\#}$ (rule $ep9$). If $exp_r(\$v) = \$v'/\ldots/s_n$, with $\$v' \neq \$v$, then we extract from the XPath expressions bound to $\$v'$ those that are relative to $\$v$ (if any exist). These are used to substitute $\$v'$ and create the output set of *returned paths* $\mathcal{P}^{\#}$ (rule $ep10$). Otherwise (rule $ep11$), no paths relative de $\$v$ can be extracted from the input expression and the output set of *returned paths* is empty. The output set of *used paths* $\mathcal{P}$ is empty in the three cases.

$$\frac{}{Env \vdash extractPaths(\$v, \$v/\ldots/s_n) \Rightarrow \emptyset, \{\$v/\ldots/s_n\}}\ (ep9)$$

$$\frac{Env.getXPathBind(\$v') \Rightarrow B \quad P^{\#} = \{p' = p/s_1/\ldots/s_n, p \in B \wedge p = \$v/\ldots\}}{Env \vdash extractPaths(\$v, \$v'/s_1/\ldots/s_n) \Rightarrow \emptyset, P^{\#}}\ (ep10)$$

$$\frac{}{Env \vdash extractPaths(\$v, doc(uri)/\ldots/s_n) \Rightarrow \emptyset, \emptyset}\ (ep11)$$

FLWR expression, quantifier. Here, the input expression $exp_r(\$v)$ is a FLWR or quantifier expression: for instance, $for\ \$v'\ in\ e_1\ where\ e_2\ return\ e_3$.

The first three premises (or the first two, in the case of the quantifier expression) in the inference rules $ep12$ to $ep14$ will simply apply recursively the path analysis process to the subexpressions e_1, e_2 and e_3.

The role of the remaining premises is to transform some of the returned paths $\mathcal{P}_1^{\#}$ from $e1$ into used paths for $\$v$. This would obviously allow for more drastic query simplifications in the later stages. The transformation happens when certain conditions are verified for the projection paths $\mathcal{P}_1^{\#}$, more precisely when (1) the paths are bound to variable $\$v'$ (this can be checked by testing if they appear in $Env.getXPathBind(\$v')$), and (2) the paths are relative to $\$v$ (i.e., $\$v$ is their first step). The paths verifying these two conditions are moved from the set of *returned paths* to the one of *used paths*.

Consider the following example:

EXAMPLE 3.1. *Given the query:*
for $\$v$ *in* ...
return for $\$v'$ *in* $(\$v/A/B, \ldots)$
return ...
the path $\$v/A/B$ *is bound to variable* $\$v'$. *Although* $\$v/A/B$ *is initially a* returned *path, the fact that it is relative to* $\$v$ *enables us to safely consider it a* used path. *In this case, the descendants of B elements are deemed not necessary to compute the end result (unless some other path overwrites this fact).*

$$\frac{\begin{array}{c} Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \\ Env \vdash extractPaths(\$v, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#} \\ Env.getXPathBind(\$v') \Rightarrow P \\ \mathcal{P}_{1.bis} = \{p \in P, p = \$v/\ldots\}, \mathcal{P}_1^{\#} - \mathcal{P}_{1.bis} \Rightarrow \mathcal{P}_{1.bis}^{\#} \end{array}}{\begin{array}{c} Env \vdash extractPaths(\$v, for\ \$v'\ in\ e_1\ where\ e_2\ return\ e_3) \\ \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_{1.bis}, \mathcal{P}_{1.bis}^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#} \end{array}}\ (ep12)$$

$$\frac{\begin{array}{c} Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \\ Env \vdash extractPaths(\$v, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#} \\ Env.getXPathBind(\$v') \Rightarrow P \\ \mathcal{P}_{1.bis} = \{p \in P, p = \$v/\ldots\}, \mathcal{P}_1^{\#} - \mathcal{P}_{1.bis} \Rightarrow \mathcal{P}_{1.bis}^{\#} \end{array}}{\begin{array}{c} Env \vdash extractPaths(\$v, let\ \$v' := e_1\ where\ e_2\ return\ e_3) \\ \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_{1.bis}, \mathcal{P}_{1.bis}^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#} \end{array}}\ (ep13)$$

$$\frac{\begin{array}{c} Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \\ Env.getXPathBind(\$v') \Rightarrow P \\ \mathcal{P}_{1.bis} = \{p \in P, p = \$v/\ldots\}, \mathcal{P}_1^{\#} - \mathcal{P}_{1.bis} \Rightarrow \mathcal{P}_{1.bis}^{\#} \end{array}}{\begin{array}{c} Env \vdash extractPaths(\$v, some\ \$v'\ in\ e_1\ satisfies\ e_2) \\ \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_{1.bis}, \mathcal{P}_{1.bis}^{\#} \cup \mathcal{P}_2^{\#} \end{array}}\ (ep14)$$

EXAMPLE 3.2. *We give below the outcome of the extractPaths function on the variable* $\$j$ *and its return expression in* $Q_{1.2}$. *The prefix indicates the rules that were applied.*

$$\begin{array}{rll} (ep11)\ Env & \vdash & extractPaths(\$j, docD@S_1/site) \Rightarrow \emptyset, \emptyset \\ (ep5, ep9, ep10)\ Env & \vdash & extractPaths(\$j, \$j/person = \$k\ldots) \\ & & \Rightarrow \emptyset, \{\$j/person\} \\ (ep6, ep9)\ Env & \vdash & extractPaths(\$j, <common\text{-}auction> \\ & & \{\$j/open\_auction\}</common\text{-}auction>) \\ & & \Rightarrow \emptyset, \{\$j/open\_auction\} \\ (ep12)\ Env & \vdash & extractPaths(\$j, for\ \$k\ \ldots) \\ & & \Rightarrow \emptyset, \{\$j/open\_auction, \$j/person\} \end{array}$$

3.2 Path projection

In this section, we present the function *projectPaths*. Recall that for each variable $\$v$, this function projects out the useless parts of $exp_b(\$v)$ (the bound expression for $\$v$) based on the sets of paths $\mathcal{P}$ and $\mathcal{P}^{\#}$ extracted from $exp_r(\$v)$ (the return expression for $\$v$). *projectPaths* takes as input a set of *used paths*, a set of *returned paths* and an XQuery expression, and returns a new, simplified XQuery expression. The output expression is obtained by projecting out any subexpression producing an intermediary result that is not in the scope of these paths. For each extracted path p and the given expression e, *projectPaths* determines if a matching is possible between p and the expected result of e (i.e., if we can expect a non empty result for the evaluation of p on the result of e).

Next, we detail the inference rules for *projectPaths*.

Literal, comparison, quantifier. When matching a set of paths with a literal value, the only cases that yield a non empty result are when there is at least one path p such that $p = text()$ (a final step of a *projection path*) or $p = \$v$ (rule $pp1$). Otherwise (rule $pp2$), the output expression is empty.

We use the same reasoning when the input query is a comparison (e_1 *Op* e_2) or a quantifier expression (*some* $\$v$ *in* e_1 *satisfies* e_2), as the result of their evaluation is necessarily a numeric or boolean value. The corresponding rules ($pp3$ to $pp6$) are similar to $pp1$ and $pp2$. They are detailed in [11].

$$\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^{\#}), p = (\$v \mid text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, literal) \Rightarrow literal} (pp1)$$

$$\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^{\#}),\ p \neq (\$v \mid text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, literal) \Rightarrow ()} (pp2)$$

Sequence. Matching an input path p with a sequence (e_1, e_2) amounts to matching p with the subexpressions (e_2 and e_3) composing the sequence. Then, the output expression is a sequence composed of the obtained elementary results (rule $pp7$).

$$\frac{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_1) \Rightarrow e'_1 \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_2) \Rightarrow e'_2 \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, (e_1, e_2)) \Rightarrow e'_1, e'_2} (pp7)$$

Variable reference. The evaluation of paths on a variable $\$v$ amounts to their evaluation on the objects to which $\$v$ is bound. Then, if a matching is possible between at least an input path p and one of the objects bound to $\$v$, the output expression is $\$v$ itself (rule $pp8$). Otherwise (rule $pp9$), the output expression is empty.

$$\frac{\begin{array}{c} (Env.getBind(\$v) \Rightarrow \{o_1, \ldots, o_n\}) \\ \exists i, 1 \leq i \leq n, projectPaths(\mathcal{P}, \mathcal{P}^{\#}, o_i) \Rightarrow o'_i \ s.t.\ o'_i \neq () \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, \$v) \Rightarrow \$v} (pp8)$$

$$\frac{\begin{array}{c} (Env.getBind(\$v) \Rightarrow \{o_1, \ldots, o_n\}) \\ \forall i, 1 \leq i \leq n, projectPaths(\mathcal{P}, \mathcal{P}^{\#}, o_i) \Rightarrow () \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, \$v) \Rightarrow ()} (pp9)$$

XPath expression. Here, the input expression e is an XPath $s_1/\ldots/s_n$. Matching an input path p with e depends on the nature of p and on the expected result of e. The necessary information on this expected result can be deduced from the last step s_n of e (rule $pp10$).

More precisely, if $s_n = text()$, the result of e will be composed of literal values. So, the only case where a matching is possible is when $p = text()$ or $p = \$v$.

If $s_n \neq text()$ and $s_n \neq *$, hence the nodes returned by $e = s_1/\ldots/s_n$ are element nodes, it is sufficient to have one input path in $\mathcal{P} \cup \mathcal{P}^{\#}$ that starts with $\$v$, $*$ (which corresponds to any element test) or s_n (i.e., the first step of p matches with the elements returned by e). Since we do not have enough information about the eventual descendants of s_n, we only check if a matching is possible between e and the first step of p.

If $s_n = *$, hence the label of the returned element nodes is unknown, the only case where a matching is not possible is when $p = text()$.

Otherwise (rule $pp11$), no matching is possible and the output expression is empty.

$$\frac{\begin{array}{c} \exists p \in (\mathcal{P} \cup \mathcal{P}^{\#}), ((s_n = text()) \wedge (p = text() | \$v)) \ \vee \\ ((s_n \neq text()) \wedge (s_n \neq *) \wedge head^2(p) = (*|s_n|\$v)) \ \vee \\ ((s_n = "*") \wedge (p \neq text())) \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, s_1/\ldots/s_n) \Rightarrow s_1/\ldots/s_n} (pp10)$$

$$\frac{otherwise}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, s_1/\ldots/s_n) \Rightarrow ()} (pp11)$$

[2] *head(p)* is a function that retrieves from a path p its first step.

FLWR expression. According to the XQuery semantics, the result of a FLWR expression is computed by its *return* subexpression. So, matching an input path p with a FLWR expression e amounts to matching p with the *return* subexpression of e (e_3 in $pp12$). If, for at least one path $p \in \mathcal{P} \cup \mathcal{P}^{\#}$, the result expression e'_3 is not empty, the composed output is a new FLWR expression obtained by substituting the initial *return* expression by e'_3 (rule $pp12$). Otherwise (rule $pp13$), the output expression is empty. For space reason, we do not detail the rules for let expressions ($pp14$ and $pp15$). These rules are similar to those given here, and can be found in [11].

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow e'_3}{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, for\ \$v\ in\ e_1\ where\ e_2\ return \\ e_3) \Rightarrow for\ \$v\ in\ e_1\ where\ e_2\ return\ e'_3 \end{array}} (pp12)$$

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow ()}{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, for\ \$v\ in\ e_1\ where\ e_2\ return \\ e_3) \Rightarrow () \end{array}} (pp13)$$

Conditional. Matching an input path p with a conditional expression ($if\ (e_1)\ then\ e_2\ else\ e_3$) amounts to matching p with the subexpressions of the $true$ and $false$ branches. The output expression is then a new conditional expression obtained by substituting in the input expression the initial $true$ and $false$ branches by the two potentially simplified subexpressions (see rule $pp16$).

$$\frac{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_2) \Rightarrow e'_2 \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow e'_3 \end{array}}{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, if\ (e_1)\ then\ e_2\ else\ e_3) \Rightarrow \\ if\ (e_1)\ then\ e'_2\ else\ e'_3 \end{array}} (pp16)$$

Element construction. Here, the input expression is an element construction expression $expr = element\{QName\}\{e\}$. Matching the input path p with $expr$ depends on the nature of p, as described in the following case analysis. In order to simplify the presentation, we assume that the application of the rules is attempted according to the order in which they are presented below:

Rule $pp17$: if the input set $\mathcal{P}^{\#}$ contains a path p such as $p = \$v$ or $p = QName$, then a matching is possible between $expr$ and at least one *returned path*. In this case, nothing can be projected out and the returned expression is identical to the input one.

Rule $pp18$: if there is no input path $p = s_1/\ldots$ with the first step $s_1 = \$v$ or $s_1 = QName$, then no matching is possible between $expr$ and the input paths. In this case, the output expression is empty (i.e., the input expression is projected out).

Rule $pp19$: if for all the input paths $p = s_1/s_2/\ldots$ with a first step $s_1 = \$v$ or $s_1 = QName$, there is no matching possible between their remaining suffixes $s_2/\ldots$ and the subexpression e, then no matching is possible between $expr$ and the input paths as well. The output expression is empty in this case.

Rule $pp20$: if there is at least one *used path* p such that $p = \$v$ or $p = QName$, and no other input path has a first step equal to $\$v$ or $QName$, then we deduce that the descendent nodes of the expected element $QName$ are useless. In this case, the subexpression e is projected out and the output expression is a new element construction $QName$ with an empty content.

Rule $pp21$: if there is at least one input path $p = s_1/s_2/\ldots$ with a first step $s_1 = \$v$ or $s_1 = QName$, and a matching is possible between the remaining part $s_2/\ldots$ of p and the subexpression e, then we deduce that a matching is possible with $expr$. In this case the output is a new element construction expression for $QName$ with a new subexpression e' obtained by the recursive application of $projectPaths$ for these $s_2/\ldots$ path suffixes over e.

$$\frac{\exists p \in \mathcal{P}^{\#},\ p = (QName \mid \$v)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e\}}\ (pp17)$$

$$\frac{\forall p \in \mathcal{P} \cup \mathcal{P}^{\#}, head(p) \neq (QName \mid \$v)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow ()}\ (pp18)$$

$$\frac{\begin{array}{c}\nexists p \in \mathcal{P},\ p = (QName \mid \$v) \\ \mathcal{P}' = \{p', [QName \mid \$v]/p' \in \mathcal{P}\} \\ \mathcal{P}'^{\#} = \{p', [QName \mid \$v]/p' \in \mathcal{P}^{\#}\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^{\#}, e) \Rightarrow ()\end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow ()}\ (pp19)$$

$$\frac{\begin{array}{c}(\exists p \in \mathcal{P},\ p = (QName \mid \$v)) \wedge \\ (\forall \overline{p} \in (\mathcal{P} \cup \mathcal{P}^{\#}) - \{p\}, head(\overline{p}) \neq (QName \mid \$v))\end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{\}}\ (pp20)$$

$$\frac{\begin{array}{c}\exists p \in \mathcal{P} \cup \mathcal{P}^{\#}, head(p) = (QName \mid \$v) \\ \mathcal{P}' = \{p', [QName \mid \$v]/p' \in \mathcal{P}\} \\ \mathcal{P}'^{\#} = \{p', [QName \mid \$v]/p' \in \mathcal{P}^{\#}\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^{\#}, e) \Rightarrow e'\end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e'\}}\ (pp21)$$

Empty sequence. If the input expression is empty, then the output expression is also empty.

$$\frac{}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, ()) \Rightarrow ()}\ (pp22)$$

3.3 The pruning process

The pruning process is applied recursively by the *Prune* function, using *extractPaths* and *projectPaths* in bottom-up manner. This function takes as input a query Q and returns a new, simplified query Q'. It is defined by the following inference rules.

Literal, variable reference, XPath expression and empty sequence. When the input expression Q is a literal, a variable reference, an XPath expression or an empty sequence, the pruning has no effect and the output expression is the same as the input one.

$$\frac{}{Env \vdash Prune(literal) \Rightarrow Literal}\ (p1)$$

$$\frac{}{Env \vdash Prune(\$v) \Rightarrow \$v}\ (p2)$$

$$\frac{}{Env \vdash Prune(s_1/\ldots/s_n) \Rightarrow s_1/\ldots/s_n}\ (p3)$$

$$\frac{}{Env \vdash Prune(()) \Rightarrow ()}\ (p4)$$

Sequence, comparison, element construction. The pruning of a sequence of subexpressions returns as a result the sequence of the pruned subexpressions (rule $p5$). We use the same approach to prune comparison expressions and element construction expressions (rules $p6$ and $p7$).

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env \vdash Prune(e_2) \Rightarrow e'_2}{Env \vdash Prune(e_1, e_2) \Rightarrow e'_1, e'_2}\ (p5)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env \vdash Prune(e_2) \Rightarrow e'_2}{Env \vdash Prune(e_1\ Op\ e_2) \Rightarrow e'_1\ Op\ e'_2}\ (p6)$$

$$\frac{Env \vdash Prune(e) \Rightarrow e'}{Env \vdash Prune(element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e'\}}\ (p7)$$

Conditional. In this case, the pruning operation is propagated to the condition subexpression, and to the *true* and *false* branches. The output expression is obtained by substituting the subexpressions by their corresponding pruned results.

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash Prune(e_3) \Rightarrow e'_3\end{array}}{Env \vdash Prune(if(e_1)\ then\ e_2\ else\ e_3) \Rightarrow if(e'_1)\ then\ e'_2\ else\ e'_3}\ (p8)$$

FLWR expressions, quantifier. When the input Q is a FLWR expression ($for\ \$v\ in\ e_1\ return\ e_2$), the pruning operation is first applied on the bound expression e_1 (with result e'_1). Then, the variable $\$v$ is added to the environment Env with its bound objects computed by *varRes*. The *saturate* function refines the bindings stored in the environment (as described in Section 2).

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}, \mathcal{P}^{\#} \\ Env \vdash projectPaths(\mathcal{P} \cup \{\$v\}, \mathcal{P}^{\#}, e'_1) \Rightarrow e''_1\end{array}}{Env \vdash Prune(for\ \$v\ in\ e_1\ return\ e_2) \Rightarrow for\ \$v\ in\ e''_1\ return\ e'_2}\ (p9)$$

Then, the pruning step is applied on the return subexpression e_2. The result is a pruned expression e'_2, from which the *extractPaths* function extracts the paths relative to $\$v$. Finally, the extracted paths ($\mathcal{P} \cup \{\$v\}$, $\mathcal{P}^{\#}$) are applied on e'_1 in order to eliminate its useless parts. Here, $\$v$ is added as a used path to ensure that the number of iterations remains the same. The resulting expressions e''_1 and e'_2 replace e_1 and e_2 in the FLWR expression.

The pruning of a for expression can lead to the following interesting special cases ($p10$ and $p11$):

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(for\ \$v\ in\ e_1\ return\ e_2) \Rightarrow ()}\ (p10)$$

Here, the pruning of the bound expression generates an empty result, which means that the number of iterations is equal to 0. So, the pruning result of the entire FLWR expression is empty.

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e'_1 \\ RootPath(e'_1) \Rightarrow PathB_1 \\ Env = Env + (\$v \Rightarrow PathB_1) \quad Env = Env.saturate(\$v) \\ Env \vdash Prune(e_2) \Rightarrow ()\end{array}}{Env \vdash Prune(for\ \$v\ in\ e_1\ return\ e_2) \Rightarrow ()}\ (p11)$$

This case corresponds to the situation in which the pruning of the return expression e_2 leads to the empty result. This means that whatever the number of iterations is, the result is always empty. In such a case, the pruning of the entire for expression gives an empty result.

When the input for expression contains a *where* clause, we follow the same reasoning and handle the *where* clause in the same way we handled the *return* subexpression (rules $p12$ to $p14$).

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash Prune(e_3) \Rightarrow e'_3 \\ Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^{\#}{}_2 \\ Env \vdash extractPaths(\$v, e'_3) \Rightarrow \mathcal{P}_3, \mathcal{P}^{\#}{}_3 \\ Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3 \cup \{\$v\}, \mathcal{P}^{\#}_2 \cup \mathcal{P}^{\#}_3, e'_1) \Rightarrow e''_1\end{array}}{\begin{array}{c}Env \vdash Prune(for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow \\ for\ \$v\ in\ e''_1\ where\ e'_2\ return\ e'_3\end{array}}\ (p12)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p13)$$

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash Prune(e_3) \Rightarrow ()\end{array}}{Env \vdash Prune(for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p14)$$

When the for expression contains a *where* clause, an additional rule is used each time the pruning of the *where* gives an empty result (rule p15). In this case, the condition is obviously $false$ (empty sequence) and the pruning of the whole for expression yields an empty result.

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \quad Env \vdash Prune(e_2) \Rightarrow () }{Env \vdash Prune(for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p15)$$

The pruning process of *let* expressions is similar to the pruning of *for* expressions (see rules $p16$ to $p20$). The only notable difference is that we do not have to add the path $\$v$ to the set of used paths $\mathcal{P}$.

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \quad Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}, \mathcal{P}^{\#} \quad Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e'_1) \Rightarrow e''_1}{Env \vdash Prune(let\ \$v\ :=\ e_1\ return\ e_2) \Rightarrow let\ \$v\ :=\ e''_1\ return\ e'_2}\ (p16)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad RootPath(e'_1) \Rightarrow PathB_1 \quad Env = Env + (\$v \Rightarrow PathB_1) \quad Env = Env.saturate(\$v) \quad Env \vdash Prune(e_2) \Rightarrow ()}{Env \vdash Prune(let\ \$v\ :=\ e_1\ return\ e_2) \Rightarrow ()}\ (p17)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \quad Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash Prune(e_3) \Rightarrow e'_3 \quad Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^{\#}{}_2 \quad Env \vdash extractPaths(\$v, e'_3) \Rightarrow \mathcal{P}_3, \mathcal{P}^{\#}{}_3 \quad Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}^{\#}_2 \cup \mathcal{P}^{\#}_3, e'_1) \Rightarrow e''_1}{Env \vdash Prune(let\ \$v\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow let\ \$v\ :=\ e''_1\ where\ e'_2\ return\ e'_3}\ (p18)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \quad Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash Prune(e_3) \Rightarrow () }{Env \vdash Prune(let\ \$v\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p19)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \quad Env \vdash Prune(e_2) \Rightarrow () }{Env \vdash Prune(let\ \$v\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p20)$$

For quantifier expressions, we use the same reasoning, with the notable difference that, in the special cases, instead of returning an empty result, we return $false()$ (since a quantifier returns a boolean value).

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1)))) \quad Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}, \mathcal{P}^{\#} \quad Env \vdash projectPaths(\mathcal{P} \cup \{\$v\}, \mathcal{P}^{\#}, e'_1) \Rightarrow e''_1}{Env \vdash Prune(some\ \$v\ in\ e_1\ satisfies\ e_2) \Rightarrow some\ \$v\ in\ e''_1\ satisfies\ e'_2}\ (p21)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(some\ \$v\ in\ e_1\ satisfies\ e_2) \Rightarrow false()}\ (p22)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad RootPath(e'_1) \Rightarrow PathB_1 \quad Env = Env + (\$v \Rightarrow PathB_1) \quad Env = Env.saturate(\$v) \quad Env \vdash Prune(e_2) \Rightarrow ()}{Env \vdash Prune(some\ \$v\ in\ e_1\ satisfies\ e_2) \Rightarrow false()}\ (p23)$$

3.4 Correctness

We prove that the rule-based algorithm is correct, showing that its input and output queries are *equivalent* (i.e., the evaluation of the output query yields the same result as the evaluation of the initial query). More precisely, we prove that the algorithm *Prune* described in Section 3.3 verifies the following.

THEOREM 3.1. *[Equivalence] Let q be an XQuery query, let $I = \{d_1, \ldots, d_k\}$ be the set of XML documents used in q, let Env be the environment, and let q' be the XQuery query obtained from the pruning of q in Env (i.e., $Env \vdash Prune(q) \Rightarrow q'$). Then, the results of q and q' over I are equal: $q(I) = q'(I)$, where "=" denotes the deep equality defined for XML values in [14].*

PROOF SKETCH. A proof for this theorem can be constructed by induction on the inference rule for each expression. Due to space limitations, we were not able to include all the details of the proof in this paper. The detailed proof (including the proof of the two lemmas presented below), can be found in the extended version of this work [11].

As it was shown in Section 3.3, when the input query q is a literal value, a variable name, an XPath expression or an empty sequence (rules $p1$ to $p4$), the pruning process will produce an output query q' identical to its input q.

When the input query q is a conditional expression, a non empty sequence, a comparison expression, an arithmetic or a logical expression (pruning rules $p5$ to $p8$), the pruning process is simply applied recursively to the subexpressions of q, replacing them by the obtained pruned expressions.

Assuming that the pruned subexpressions are equivalent to the initial ones, then according to the semantics of XQuery[7] the output query q' is equivalent to q.

For instance, considering a conditional expression

$$q = if(e_1)\ then\ e_2\ else\ e_3$$

the pruning process yields

$$q' = if(e'_1)\ then\ e'_2\ else\ e'_3$$

where $Env \vdash Prune(e_1) \Rightarrow e'_1, Env \vdash Prune(e_2) \Rightarrow e'_2$, and $Env \vdash Prune(e_3) \Rightarrow e'_3$. Assuming that e_1 is equivalent to e'_1, e_2 is equivalent to e'_2 and e_3 is equivalent to e'_3, it is easy to check that expressions q and q' are equivalent as well.

FLWR expressions. When the input query q is a FLWR expression (pruning rules $p9$ to $p20$ in Section 3.3), the pruning process can be summarized in three main steps : (1) the pruning is applied recursively to the subexpressions of q, substituting them by the obtained pruned expressions (e'_1, e'_2 and e'_3), (2) *extractPaths* is called to extract from the *return* and *where* subexpressions (e'_2 and e'_3) the *used* and *return paths* (P and $P^{\#}$), and (3) *projectPaths* is called to apply the extracted paths on the bound subexpression (e'_1) in order to simplify it.

We argue that the first step preserves the equivalence: assuming that e_1 is equivalent to e'_1, e_2 is equivalent to e'_2 and e_3 is equivalent to e'_3, then *for* $\$v$ *in* e_1 [*where* e_2] *return* e_3 is equivalent to *for* $\$v$ *in* e'_1 [*where* e'_2] *return* e'_3 (similar for the *let* expressions *let* $\$v$:= e_1 [*where* e_2] *return* e_3 and *let* $\$v$:= e'_1 [*where* e'_2] *return* e'_3).

To prove equivalence between the input query q and the output one q', we need the following result: the projected subexpression e''_1, obtained by applying the *used* and *return paths* on the bound subexpression e'_1, generates all the nodes necessary in the evaluation of e'_2 and e'_3 (this is similar to the *Return Paths Lemma* of [16]). More precisely, we need to prove the following two properties.

LEMMA 3.1. *[Paths Extraction] Let e be an XQuery expression and $v be a variable. The sets of paths $\mathcal{P}$ and $\mathcal{P}^{\#}$ extracted from e ($Env \vdash extractPaths(\$v, e) \Rightarrow \mathcal{P}, \mathcal{P}^{\#}$) satisfy the following:*

- *$\mathcal{P}$, $\mathcal{P}^{\#}$ contain all the paths in e referencing $v, and nothing else.*
- *only* used paths *are contained in $\mathcal{P}$.*

LEMMA 3.2. *[Paths Projection] Let e_1 be an XQuery expression, let $\mathcal{P}$ be a set of* used paths *and $\mathcal{P}^{\#}$ a set of* return paths. *The XQuery expression e_2, obtained by the application of $\mathcal{P}$ and $\mathcal{P}^{\#}$ paths on e_1 ($Env \vdash ProjectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_1) \Rightarrow e_2$), satisfies the following properties:*

- $\forall p \in \mathcal{P} : \text{root}(eval(p, e_2)) = \text{root}(eval(p, e_1))$
- $\forall p \in \mathcal{P}^{\#} : eval(p, e_2) = eval(p, e_1)$

where the *root* function retrieves the root nodes of its input XML data, and the *eval* function is defined as follows:

DEFINITION 3.1 (EVAL). *For an XQuery expression q and an XPath expression p, $eval(p, q)$ denotes the following XQuery expression:*

- *q/p_1, if $p = \$var/p_1$ (i.e., p starts with a variable reference);*
- *$q/self :: p$, otherwise.*

Lemmas 3.1 and 3.2 presented above can be proven by induction over each expression. The details can be found in [11]. □

4. OPTIMIZED PRUNING

We discuss in this section extensions on the algorithm presented previously, which may further simplify an XQuery expression.

Path refinements. Let us illustrate a first improvement by an example. Let q be the following XQuery expression :

```
for $j in (for $i in <A><B/><C/></A> return $i)
return $j/B
```

By applying the pruning rule on q, no simplifications would apply and the query remains unchanged. However, one can easily notice that the C elements constructed by the inner for are not necessary for the end result and can thus be projected out.

This kind of pruning is not possible using the bottom-up inference rules of Section 3.3. This is mainly due to the fact that when we prune some inner subexpression, we have no information about the outer subexpressions, potentially missing such further refinements. For instance, in the query q, when we prune the inner for subexpression, the projection path $\$i$ suggests to keep all the bound expression of $\$i$ (by rule $pp1$). Then, when we apply the path $\$j/B$ on the inner for, we apply it in fact only on the variable $\$i$ (by rule $pp12$), and we conclude that we must keep the variable $\$i$ (by rule $pp8$). In this way, we fail to refine the path $\$i$ to $\$i/b$ and to detect that only B elements, children of As, are needed from $\$i$'s content.

In order to detect this kind of pruning opportunity, we have to use information of the entire query during the pruning process.

We are currently extending our algorithm to take into account such simplification opportunities, using a two-step pruning. In the first step, we apply the pruning as described previously, starting with empty sets of used and returned paths for each variable. In addition, when we have situations in which we apply a path p on a variable reference $\$v$ or on a path p' starting by a variable name $\$v$, we keep in a separate structure a mapping from the variable to the projection path that is applied, i.e. ($\$v \Rightarrow p$) or ($\$v \Rightarrow p'/p$), along with its kind (used or returned). Then, these mappings are used to initialize the sets $\mathcal{P}$ and $\mathcal{P}^{\#}$ for a second pruning pass.

Going back to the example, in the first step, q remains unchanged but we extract a mapping $\$i \Rightarrow \{\$i/B\}$ (as returned path). In the second step, when pruning the inner for, having this path enables us to prune the C elements as well.

Elimination of useless XPath expressions. Notice that the query $Q'_{1.3}$ presented in Section 1 contains a path, $\$j/open\_auction$, whose evaluation is not necessary for the end result. This is because it always returns the empty sequence $()$. Our second improvement addresses this issue, eliminating irrelevant navigation by directly substituting such paths by the empty sequence $()$.

The simplified query $Q''_{1.3}$ would now be the following:

```
for $j in <site>{()}</site>
return
   for $k in doc1@S1/site
   where $j/person = $k/people/person
   return
      <common-auction>{()}</common-auction>
```

Intuitively, we can find these paths during the *projectPaths* process, when applying paths over expressions. In the same way we retrieve the expressions parts that do match some path, we can also retrieve the paths that do not match anything in these expressions. For space reasons, the modified *projectPaths* rules that take into account such cases are presented in the extended version of this paper. Further details are omitted.

5. EXPERIMENTS

We implemented our algorithm for XQuery projection as a separate module on top of the Galax query processor [9] (version 0.7.2). Our choice was motivated by the robustness of this processor and its conformance with the W3C XQuery specifications.

We describe in this section the impact of our approach, measuring the gain in evaluation time obtained by eliminating the computation of irrelevant intermediate results. In our experiments, we varied the nature and complexity of the pruned subexpressions. More precisely, we considered three kinds of subexpressions widely used in practice : FLWR blocks, XPath expressions relative to a given document or XPath expressions relative to a variable. For each kind of subexpression, we varied the amount of intermediate results produced by the pruned subexpression: 25%, 50%, 75% or 100% of the total intermediate results. We used in our experiments the following template for test queries:

```
let $q := <personInf>
            {for $i in doc("xmark.xml")/site/people/person
             return
               (< name > {test_exp} < /name >,
                < age > {test_exp} < /age >,
                < gender > {test_exp} < /gender >,
                < email > {test_exp} < /email >)}
          </personInf>
for $j in $q
return($j/names?,$j/age?,$j/gender?,$j/email?)
```

where the question mark indicates optional parts that could be missing from one test query to another. By the first *let* clause in the template we create a set of intermediate results. The *let* binds the variable $\$q$ to a *personInf* element that contains four child elements *name*, *age*, *gender* and *email*. The four elements have the same content, produced by a *test_exp* expression (to be defined for each test query).

The number of children nodes of *personInf* depends on the size of the sequence to which the variable $\$i$ is bound (*person* elements) and varies with the size of the document on which the test is performed. The percentage of useless intermediate results is simply tuned by deciding which XPath expressions appear in the query, among the four expressions given in the *return* of the outer for

clause. For example, when testing the gain for 100% of irrelevant intermediate results, we can use the path $\$j/names$, because it does not follow any child element of the *personInf* element. When testing the gain for 50% of irrelevant intermediate results, we can use two paths, such as $\$j/age$ and $\$j/gender$.

Finally, the kind of expression that is pruned along with its wrapping element was also varied (*test_exp*).

We show in Figures 3, 4, and 5 the improvements when *test_exp* is a FLWR block, an XPath expression relative to a variable or an XPath expressions relative to a document. The measures were conducted on a Pentium 3.2 GHz Linux PC with 2Gb of RAM.

Results & Discussion. The experiments show that our approach ensures a gain of time whatever is the nature of the pruned subexpressions. The gain varies according to the amount of pruned intermediate results and the complexity of the irrelevant subexpression.

In Figure 3, where the pruned subexpressions correspond to FLWR blocks, the savings in evaluation time are determined by the amount of pruned intermediate results. These savings increase slightly when the document size increases. They increase significantly when the pruned subexpressions correspond to XPath expressions relative to a document (Figure 4). We believe that this is mainly due to the specificity of the XQuery processor we used. In Figure 5, the pruned subexpressions correspond to XPath expressions relative to a variable. In this case, we measured savings of time less important than in the two previous cases. It seems that in this kind of scenarios we save only the time needed to retrieve the element returned by the path, which is normally done in main memory.

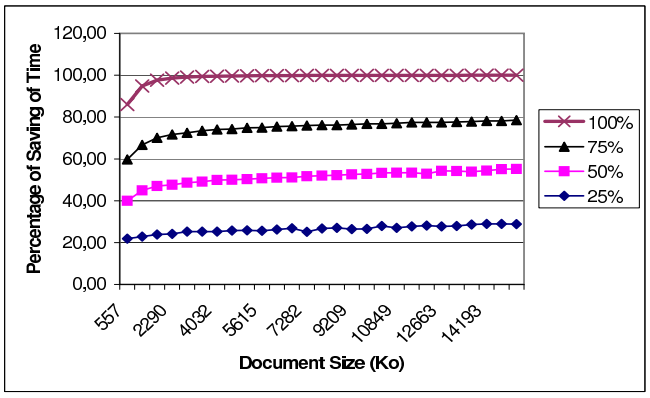

Figure 3: Test results for queries pruning FLWR blocks

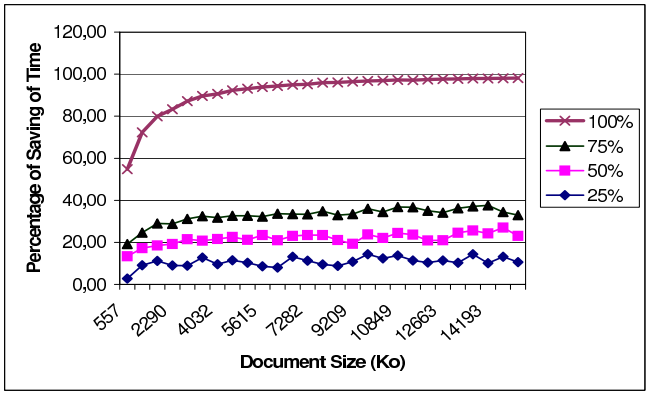

Figure 4: Test results for queries pruning variable XPaths

6. CONCLUSION

We present in this paper a rewriting algorithm for XQuery queries which detects and prunes the computations that are irrelevant for the overall result. For each input query, our algorithm outputs an equivalent, simplified query. We show by extensive experiments important savings in evaluation time, and we prove formally the correctness of our algorithm. An important direction for future research is to extend the algorithm by taking into account schema information.

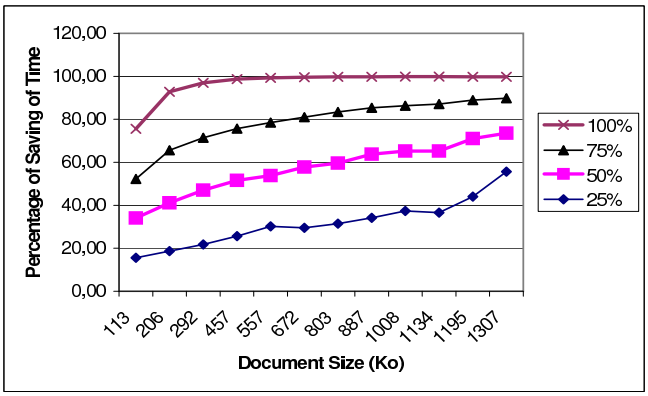

Figure 5: Test results for queries pruning document XPaths

7. REFERENCES

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *SIGMOD Conf*, 2004.

[2] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based XML projection. In *VLDB Conf*, 2006.

[3] M. Brantner, C-C. Kanne, and G. Moerkotte. Let a Single FLWOR Bloom (to improve XQuery plan generation). In *XSym Workshop*, 2007.

[4] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. Xperanto: Middleware for publishing object-relational data as XML documents. In *VLDB Conf*, 2000.

[5] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Logical Framework for XQuery. In *VLDB Conf*, 2004.

[6] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of nested XML queries. In *VLDB Conf*, 2004.

[7] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation*, 2007.

[8] M. F. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. Silkroute: A framework for publishing relational data in XML. *ACM Trans. Database Syst.*, 27(4), 2002.

[9] M. F. Fernández and J. Siméon. *The Galax System "The XQuery Implementation for Discriminating Hackers" Version 0.7.2*, 2007.

[10] M. Grinev. XQuery Optimizing Based on Rewriting. In *ADBIS*, 2004.

[11] B. Gueni, T. Abdessalem, B.Cautis, and E.Waller. Pruning Nested XQuery Queries. Technical report, Telecom ParisTech, http://www.tsi.enst.fr/publications/enst/techreport-2008-8307.pdf, 2008.

[12] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD Conf*, 2005.

[13] C. Koch. On the role of Composition in XQuery. In *WebDB Workshop*, 2005.

[14] A. Malhotra, J. Melton, and N. Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation*, 2007.

[15] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB Conf*, 2001.

[16] A. Marian and J. Siméon. Projecting XML Documents. In *VLDB Conf*, 2003.

[17] P. Michiels. XQuery Optimization. In *VLDB PhD Workshop*, 2003.

[18] P. Michiels, G. A. Mihaila, and J. Siméon. Put a tree pattern in your algebra. In *ICDE Conf*, 2007.

[19] P. Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *SIGMOD Conf*, 2002.

[20] A. Schmidt, F. Waas, M. Kirsten, M. J.Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB Conf*, 2002.

[21] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB Conf*, 2001.

[22] I. Tatarinov and A. Y. Halevy. Efficient Query Reformulation in Peer-Data Management Systems. In *SIGMOD Conf*, 2004.