XQuery Optimization Based on Program Slicing [*]

Jesús M. Almendros-Jiménez
Dpto. de Lenguajes y Computación
Universidad de Almería
Carretera de Sacramento s/n
E-04120 Almería, Spain
jalmen@ual.es

Josep Silva
Dpto. de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera s/n
E-46022 Valencia, Spain
jsilva@dsic.upv.es

Salvador Tamarit
Dpto. de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera s/n
E-46022 Valencia, Spain
stamarit@dsic.upv.es

ABSTRACT

XQuery has become the standard query language for XML. The efforts put on this language have produced mature and efficient implementations of XQuery processors. However, in practice the efficiency of XQuery programs is strongly dependent on the ability of the programmer to combine different queries which often affect several XML sources that in turn can be distributed in different branches of the organization. Therefore, techniques to reduce the amount of data loaded and also to reduce the intermediate structures computed by queries is a necessity. In this work we propose a novel technique that allows the programmer to automatically optimize a query in such a way that unnecessary intermediate computations are avoided, and, in addition, it identifies the paths in the source XML documents that are really required to resolve the query.

Categories and Subject Descriptors

H.2 [**DATABASE MANAGEMENT**]: H.2.3 Languages—*Query languages*; F.3 [**LOGICS AND MEANINGS OF PROGRAMS**]: F.3.2 Semantics of Programming Languages—*Program analysis*

General Terms

Languages

Keywords

XQuery, Slicing, Query Optimization

[*] This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grants TIN2008-06622-C03-02 and TIN2008-06622-C03-03, by the *Generalitat Valenciana* under grant PROMETEO/2011/052, and by the *Junta de Andalucía* under grant TIC-6114. Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.
Copyright 2011 ACM 978-1-4503-0717-8/11/10 ...$10.00.

1. INTRODUCTION

XQuery [21, 6] is a typed functional language devoted to express queries against XML documents. It contains *XPath* [2] as a sublanguage. *XPath* supports navigation, selection and extraction of fragments from XML documents. *XQuery* also includes *flowr* expressions (i.e. *for-let-orderby-where-return* expressions) to construct new XML values and to join multiple documents. The design of *XQuery* has been influenced by group members with expertise in the design and implementation of other high-level languages. *XQuery* has static typed semantics which is part of the *W3C* standard [6, 21].

XQuery has evolved into a widely accepted query language for XML processing and many XQuery engines have been developed [12, 8, 3, 14, 10, 16]. However, *memory consumption* and *execution time* remains a crucial bottleneck in query evaluation. Queries against large data sources require the improving of data loading and buffering together with XQuery optimization. Sometimes the refactoring of the XQuery or the pre-filtering of the source XML files is mandatory to be able to process large XML documents. As a matter of fact, standard XML processors have a maximum size in the XML documents they can process. Nevertheless, some XML documents such as the XML version of DBLP are very large and could require pre-processing and query optimization to be handled.

In this work we introduce a novel technique able to automatically project the portion of the input XML document that is needed to resolve a given XQuery expression (thus the remaining parts of the document are not loaded), and at the same time, it also allows us to optimize the XQuery itself with a refactoring process based on *program slicing*.

Program slicing is a general technique of program analysis and transformation whose main aim is to extract the part of a program (called slice) that influences or is influenced by a given point of interest [22, 20]. Program slicing has been traditionally based on a data structure called *program dependence graph* (PDG) [9] that represents all statements in a program with nodes and their control and data dependences with edges. Once the PDG is computed, slicing is reduced to a graph reachability problem, and slices can be computed in linear time.

Our slicing-based technique aims to provide an optimization method for XQuery. Given a (composed) query, we are able to detect those parts of the XQuery code which are not relevant for the output of the query. In addition, we are able to rewrite the query into a new one in which the irrelevant parts have been removed. Unfortunately, the PDG cannot be used with XQuery because the notion of statement is not applicable to functional languages. Therefore, firstly, we will define a notion of *XQuery Dependence Graph* (XDG) which is a labelled graph able to represent XQuery expressions. Secondly, we will describe how to transform an XDG in order to optimize an XQuery expression in such a way that only those expressions that contribute to the final result remain in the graph. The transformation consists on forward and backward propagation of data dependences together with an slicing procedure in order to detect the required paths in an XQuery expression. From the transformed XDG we extract a new optimized XQuery and, additionally, we deduce the parts of the documents used by the XQuery expression.

Example 1. *The following composed query requests the* `customer` *elements obtained from a nested query in which all* `customer`*, and some* `provider` *elements are computed.*

```
Q1 =  for $i in (doc(‘File’)/company)
      return <sales>{($i/customer, if ($i/provider=‘X’)
                        then () else $i/provider)}</sales>
Q2 =  for $j in Q1
      return $j/customer
```

This query shows an example of query optimization. Given that the `provider` *elements are never used by Q2, they can be pruned from Q1 when they are composed. This produces an equivalent optimized query.*

The structure of the paper is as follows. Section 2 presents the related work. Section 3 introduces some preliminaries. The XDG is introduced in Section 4, and the slicing algorithms are presented in Section 5. Section 6 describes the implementation. Section 7 presents our experimental results. Finally, Section 8 concludes and presents future work.

2. RELATED WORK

There exist two major research lines concerning the optimization of XQuery. The first line tries to improve the processing of the XML input data. The second line operates over the source XQuery expressions transforming them to improve efficiency.

On one hand, XML document *loading* and *buffering* techniques have been studied in [17, 1, 4, 19]. The static *projection* technique of [17] of input XML documents implemented in the *Galax* [8] processor and refined in [1, 4] proposes that only the parts of the input documents relevant to query evaluation are loaded into memory. The projected documents are computed before query evaluation starts. In the case of [19], they distinguish *bulk* input data only used to generate the output from input data which are traversed in query evaluation, improving XML projection.

On the other hand, XQuery *optimization* techniques have been proposed. In [13], they describe a technique for *pruning* XQuery in order to improve composition based queries. Rather than projecting XML documents, they project XQuery sub-expressions with respect to other sub-expressions querying them. In other words, they propose the pruning of queries in which an (intermediate) result computed by means of a query is used as input of another query.

Our work follows the same line as [13] and [17], providing a query optimization technique based on query transformation which combines prunning and projection. However, our technique is much more precise because it uses a data dependence analysis that is performed bottom-up and top-down in the XQuery expression. Contrarily, their pruning technique is a bottom-up analysis that fails to prune many useless expressions as shown in the following XQuery expression:

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

Clearly, in this expression, the `C` elements are not necessary and can be removed. However, this simplification cannot be detected by their pruning technique (that would leave the query unchanged). The reason is that they only use a bottom-up analysis. Hence, when some inner subexpression is pruned, they do not have information about the outer subexpressions, thus missing pruning opportunities.

Example 2. *The following query presented in [13] requests the* `close_auction` *elements obtained from a nested query in which the elements computed are* `open_auction` *elements enclosed by means of the label* `site`*.*

```
for $j in <site>{for $i in (doc(‘File1’)/site)
                 return $i/open_auctions/open_auction
          }</site>
return for $k in (doc(‘File2’)/site)
       where $j/person = $k/people/person
       return <common_auction>{$j/closed_auction}
              </common_auction>
```

The pruning technique in [13] would produce the following simplified query:

```
for $j in <site>{()}</site>
return for $k in (doc(‘File2’)/site)
       where $j/person = $k/people/person
       return <common-auction>{()}</common-auction>
```

However, this simplification is suboptimal and it can be further optimized in our approach: the nested query does not compute `close_auction` *nor* `person` *elements and therefore the query can be completely pruned because the* `where` *clause cannot be satisfied (i.e., the final query should be the empty sequence* `()`*).*

In our technique, the same analysis performed to prune XQuery expressions provides the information needed to project the source XML documents. This means that with the XDG we can also project the XML documents that participate in the XQuery expression as it is done in [17]. Let us remark that the projecting technique of [17] is also improved here by means of our query optimization technique. For instance, in Example 1 `provider` elements will not be projected from the input XML document.

There has been previous attempts to define a PDG-like data structure for functional languages. The first attempt to adapt the PDG to the functional paradigm was [18] where they introduced the *functional dependence graph* (FDG). Unfortunately, FDGs are useful at a high abstraction level (i.e., they can slice modules or functions), but they cannot slice expressions and thus they are useless for XQuery. Another approach is based on the *term dependence graphs* (TDG) [7]. However, these graphs only consider term rewriting systems with function calls and data constructors (i.e., no complex structures such as let-expressions, for-expressions, if-then-else, etc. are considered). Finally, another use of program slicing has been done in [5] for Haskell. But in this case, no new data structure was defined and the abstract syntax tree of Haskell was used with extra annotations about data dependences.

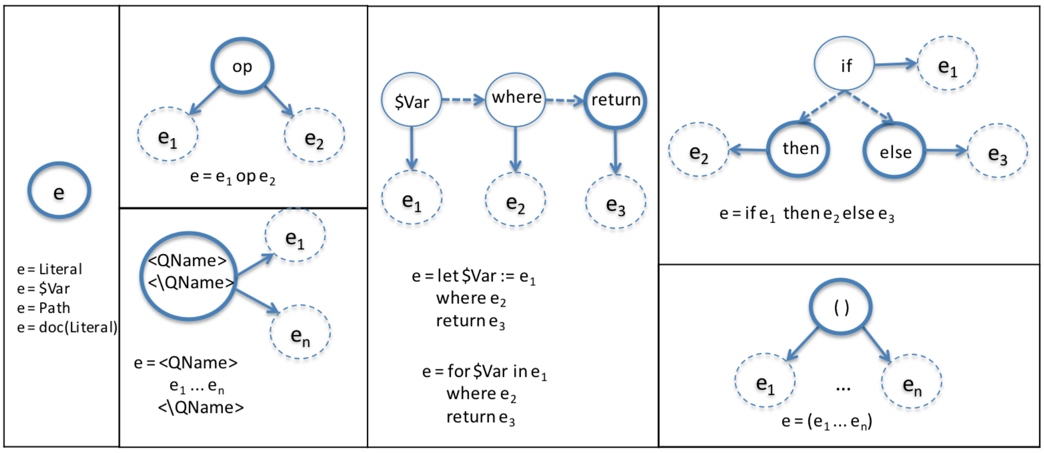


Figure 1: Graph Representation of XQuery Expressions

$$
\begin{array}{lll}
\mathit{Expr} & ::= & \mathit{Literal} \\
 & | & (\mathit{Expr}_1, \ldots, \mathit{Expr}_n) \\
 & | & \mathit{Var} \\
 & | & \mathit{doc}(\mathit{Literal}) \\
 & | & \mathit{Path} \\
 & | & \text{for Var in } \mathit{Expr} \text{ (where } \mathit{Expr}\text{)? return } \mathit{Expr} \\
 & | & \text{let Var := } \mathit{Expr} \text{ (where } \mathit{Expr}\text{)? return } \mathit{Expr} \\
 & | & \text{if } (\mathit{Expr}) \text{ then } \mathit{Expr} \text{ else } \mathit{Expr} \\
 & | & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
 & | & \mathit{Tag} \\
\mathit{Tag} & ::= & <\mathit{QName}>\ \mathit{LExpr}_1 \ldots \mathit{LExpr}_n\ <\backslash \mathit{QName}> \\
\mathit{LExpr} & ::= & \{\mathit{Expr}\} \mid \mathit{Literal} \mid \mathit{Tag} \\
\mathit{Path} & ::= & \mathit{Expr}\ \ (/\mathit{QName})^+ \\
\mathit{Var} & ::= & \$\mathit{VarName} \\
\mathit{Op} & ::= & < \mid > \mid = \mid + \mid - \mid * \mid / \mid \mathit{and} \mid \mathit{or}
\end{array}
$$

Figure 2: Syntax of XQuery expressions

3. PRELIMINARIES

For the sake of concreteness, in the rest of the paper we will consider the subset of the XQuery core language shown in Figure 2. We need to introduce a normalization process for XQuery expressions. This process ensures that (1) all variables defined in both let and for expressions are pairwise different, that is, they are renamed when they coincide; and (2) all *Path* expressions start with a variable. The normalization substitutes $e\ (/QName)^+$ by *let* $\$x := e$ *return* $\$x\ (/QName)^+$, whenever e is not a Var expression, where $\$x$ is a new variable, and e is recursively normalized.

Example 3. *The query of Example 1:*

```
for $j in for $i in (doc('File')/company)
      return <sales>{($i/customer, if ($i/provider='X')
                        then () else $i/provider)}</sales>
return $j/customer
```

is normalized as follows:

```
for $j in for $i in let $v := doc('File')
                     return $v/company
      return <sales>{($i/customer, if ($i/provider='X')
                        then () else $i/provider)}</sales>
return $j/customer
```

We define functions $first$, *suffix* and $last$ to extract a portion of a normalized path as follows:
$first(\$x\ (/QName)^+) = \$x$,
$\mathit{suffix}(\$x\ (/QName)^+) = (/QName)^+$,
$last(\$x\ /QName_1 \ldots /QName_n) = QName_n$.

4. XQUERY DEPENDENCE GRAPHS

In this section we define the *XQuery dependence graph* (XDG). Such data structure is one of the main contributions of this work. It allows us to graphically represent XQuery expressions establishing data and control relations between subexpressions. Therefore, it is very useful for refactoring and, in particular, it is the basis of our slicing algorithms for XQuery. First, we define the *graph representation* of a XQuery expression.

Definition 1 (Graph Representation). *Given a normalized XQuery expression e, we represent e with a labelled graph* $(\mathcal{N}, \mathcal{E}, \mathcal{F})$ *where* $\mathcal{N}$ *are the nodes,* $\mathcal{E} = (\mathcal{C}, \mathcal{S})$ *are edges of two types:* $\mathcal{C}$ *the control edges, and* $\mathcal{S}$ *the structural edges, and* $\mathcal{F}$ *is a set of partial functions:*

$$
\begin{array}{ll}
\mathit{type}: & \mathcal{N} \rightarrow \mathcal{T} \\
\mathit{literal}: & \mathcal{N} \rightarrow \mathit{Literal} \\
\mathit{children}: & \mathcal{N} \rightarrow \mathcal{P}(\mathit{Nat} \times \mathcal{N}) \\
\mathit{var}: & \mathcal{N} \rightarrow \mathit{Var} \\
\mathit{path}: & \mathcal{N} \rightarrow \mathit{Path} \\
\mathit{op}: & \mathcal{N} \rightarrow \mathit{Op} \\
\mathit{tag}: & \mathcal{N} \rightarrow \mathit{Tag}
\end{array}
$$

The set $\mathcal{F}$ of partial functions defines the labels of each node in the graph. Function *type* returns the type of a node. $\mathcal{T}$ is the set of node types: *literal*, *seq*, *var*, *doc*, *path*, *letBinding*, *forBinding*, *where*, *return*, *if*, *then*, *else*, *op* and *tag*. Type *seq* represents a sequence of elements in a tuple or in a *Tag* element. Function *literal* is defined for nodes of type *literal* including elements *doc(literal)*. The partial function *children* is defined for nodes of type *seq* and *op*. It returns a set of pairs of the form (pos, n) where *pos* is the position in the sequence or operation of the expression represented

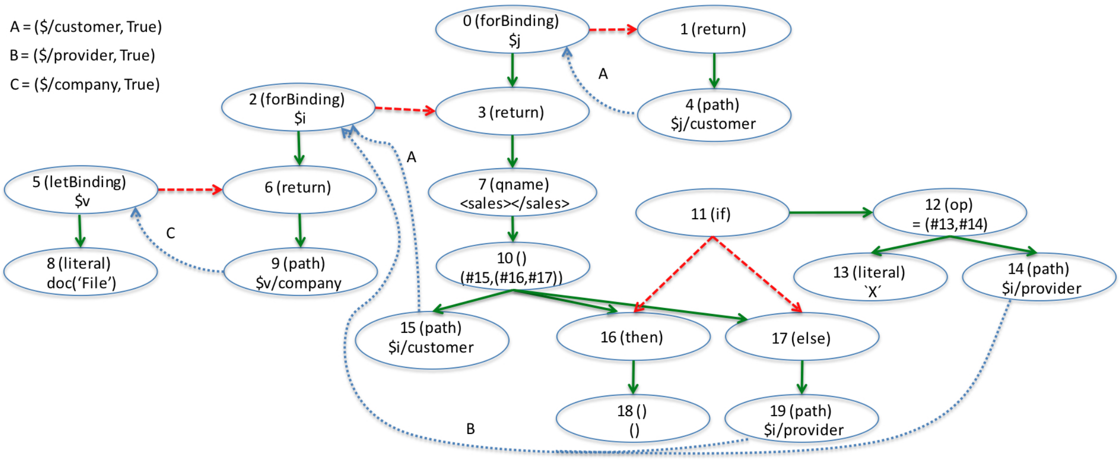


Figure 3: XQuery Dependence Graph of the running example

by node n. Function *var*, is defined for binding nodes (i.e, *forBinding* and *letBinding*) and represents the bound variable. The other functions are defined for nodes of type *path*, *op* and *tag*, respectively, and their results are straightforward. We denote by $f^{\mathcal{F}}$ the meaning that $\mathcal{F}$ assigns to the function f.

The graph representation of a XQuery expression is constructed compositionally according to the cases of Figure 1. In such representation, each graph has a *final node* (which has been graphically distinguished with a bold line) except *if-then-else* that has two final nodes. The nodes with a dashed line represent the graph associated to their subexpression, and all nodes connected to a dashed node are linked to the final nodes of the graph represented by the dashed node. We have graphically distinguished the two kinds of edges: *control edges* with a solid line, and *structural edges* with a dashed line. In addition, nodes are graphically represented including the information provided by means of the associated partial functions.

We are now in a position to define our main data structure called XQuery Dependence Graph (XDG). Essentially, the XDG augments the graph representation of a XQuery expression with the standard notion of data dependence of static analysis (see, e.g., [20]). Formally,

Definition 2 (XQuery Dependence Graph). *Given a XQuery expression e, the* XQuery Dependence Graph (XDG) *of e is a directed labelled graph $\mathcal{X} = (\mathcal{N}, \mathcal{E}, \mathcal{F})$ where $\mathcal{N}$ are the nodes and $\mathcal{E} = (\mathcal{C}, \mathcal{S}, \mathcal{D})$ are the edges. $(\mathcal{N}, \mathcal{E}', \mathcal{F}')$ is the graph representation of e being $\mathcal{E}' = (\mathcal{C}, \mathcal{S})$ with $\mathcal{C}$ the control edges, and $\mathcal{S}$ the structural edges. The set $\mathcal{D}$ represents* data edges. *We have a data edge from node n to n' iff $first(path(n)) = var(n')$. $\mathcal{F}$ is a set containing the functions in $\mathcal{F}'$ plus a partial function called* label *which returns a set of pairs of the form $(Ppath, boolean)$ for each data edge, that is, $label : \mathcal{D} \rightarrow \mathcal{P}(Ppath \times boolean)$.*

In the previous definition *Ppath* denotes *partial paths* of the form: $Ppath = (\$ \mid QName)\ (/QName)^*$. Functions *first*, *suffix* and *last* can be also defined for partial paths. Given $pp = (\$ \mid QName)\ (/QName)^*$ we have: $first(pp) = (\$ \mid QName)$ and $suffix(pp) = (/QName)^*$; in addition, $last((\$ \mid QName)\ /QName_1 \ldots /QName_n) = QName_n$ if $n \geq 1$, and $(\$ \mid QName)$, otherwise.

Observe that the definition of XDG uses the standard notion of data dependence (see, e.g., [9]). In the case of XQuery the notion of variable definition and variable usage is similar to the other languages. In particular, variables are always defined in $forBinding$ and $letBinding$ nodes and they are used in any node that contains this variable (i.e., there exists a data edge from node n to n' when $first(path(n)) = \$v$ and $var(n') = \$v$). Observe that, thanks to the normalization process, no redefinition of a variable is possible, thus, all uses of a particular variable data-depend on the same variable definition.

The labels of the data edges are useful to know what information (i.e., partial paths) is required and provided by each node (i.e., **true** means that the partial path could[1] be provided by the node, and **false** means that it cannot be provided). Such labels represent complete or incomplete paths from a bound variable. The name of the bound variable does not need to be specified in edges, for this reason we have used the notation $ in partial paths.

Example 4. *The XDG of the normalized XQuery in Example 3 is shown in Figure 3 where nodes are identified with numbers. We have graphically represented the function children by means of sequences $(\#n_1, \ldots, \#n_k)$ representing the order of the children. The example contains five data dependence edges (the dotted edges) representing the request of* `$j/customer` *from the outermost* `for` *(A), the request of* `$i/customer` *(A) and* `$i/provider` *(B) from the innermost* `for`*, and the request of* `$v/company` *from the* `let` *expression (C). Let us remark that initially they are marked as* **true***, but the proposed slicing algorithms will update such boolean values.*

5. SLICING XQUERY

In this section we describe how to transform a XDG in order to optimize an XQuery expression in such a way that only those expressions that contribute to the final result remain in the graph. From the final transformed XDG we can

[1] Whether the path is actually provided or not depends on the XML source.

(i) extract a new optimized query and (ii) deduce the parts of the source documents used by the query that are really needed.

The transformation is divided into two independent stages. In the first stage, all data dependences are *propagated* forwards and backwards in order to determine which expressions are needed and which of them are available. This process mainly affects data edges: labels of data edges are updated, and some new data edges can be also added or deleted. In the second stage, by means of an *slicing* procedure, those expressions that are useless are removed, and the graph is further transformed to ensure that the final XQuery expression is syntactically correct. In addition, there is a *garbage removal* procedure that can be applied before and after propagation and slicing procedures.

Now, we define some notation that will be used in the slicing algorithms. Given a XDG $\mathcal{G} = (\mathcal{N}, (\mathcal{C}, \mathcal{S}, \mathcal{D}), \mathcal{F})$:

(1) We use $ine_{\mathcal{E}}(n)$ and $oute_{\mathcal{E}}(n)$ to denote, respectively, the incoming and outgoing *edges* of a node $n \in \mathcal{N}$ belonging to a certain set $\mathcal{E}$:

$$ine_{\mathcal{E}}(n) = \{(n' \to n) \mid (n' \to n) \in \mathcal{E}\}$$
$$oute_{\mathcal{E}}(n) = \{(n \to n') \mid (n \to n') \in \mathcal{E}\}$$

Note that the set $\mathcal{E}$ is parameterizable; e.g., $ine_{\mathcal{S}}(n)$ denotes the incoming structural edges of n. Analogously, we use $inn_{\mathcal{E}}(n)$ and $outn_{\mathcal{E}}(n)$ to denote, respectively, the input and output *nodes* of a node n w.r.t. a certain set $\mathcal{E}$:

$$inn_{\mathcal{E}}(n) = \{n' \mid (n' \to n) \in \mathcal{E}\}$$
$$outn_{\mathcal{E}}(n) = \{n' \mid (n \to n') \in \mathcal{E}\}$$

(2) Function $reachable^{\mathcal{G}}(n)$, denotes the set of data and structural *edges* reachable from a node $n \in \mathcal{N}$:

$$reachable^{\mathcal{G}}(n) = \bigcup_{(n \to n') \in oute_{\mathcal{D} \cup \mathcal{S}}(n)} \{(n \to n')\} \ \cup \ reachable^{\mathcal{G}}(n')$$

(3) Finally, we denote by $init(\mathcal{G})$ the set of initial nodes of $\mathcal{G}$, which is defined as the set of binding nodes (i.e., *forBinding* and *letBinding*) that are not reachable from other binding nodes by traversing forwards data and structural edges. Formally:

$$\begin{array}{l} init(\mathcal{G}) = \{n \in \mathcal{N} \mid type^{\mathcal{F}}(n) \in \{letBinding, forBinding\} \\ \quad \wedge \ (\nexists n_{prev} \in \mathcal{N} \ : \\ \qquad type^{\mathcal{F}}(n_{prev}) \in \{letBinding, forBinding\} \\ \qquad\quad \wedge \ n \in reachable^{\mathcal{G}}(n_{prev}))\} \end{array}$$

Observe that initial nodes of a graph $\mathcal{G}$ can be computed in linear time by traversing $\mathcal{G}$. As an example, in Figure 3 the only initial node is node 0.

5.1 Garbage removal

We can remove from the XDG all the *letBinding* nodes that are not the target of a data edge. Such bindings are useless in the XQuery expression: they represent variables that are declared and not used. Note that *forBinding* nodes cannot be removed because they can be useful for iteration even if they do not have incoming data edges. It provides our first optimization step, and in addition, it avoids to traverse such nodes in the next stages. Such garbage removal can be done in linear time with respect to the size of the XDG and it must be done before and after both stages of the transformation because the propagation of dependences and the slicing process itself could remove the incoming data edges of a *letBinding* node producing new garbage. The Algorithm 1 implements the garbage removal process.

Algorithm 1 Garbage Removal

```
Input: A XDG $\mathcal{G} = (\mathcal{N}, \mathcal{E} = (\mathcal{C}, \mathcal{S}, \mathcal{D}), \mathcal{F})$
Output: A XDG $\mathcal{G}'$
repeat
    $Garbage := \{n \in \mathcal{N} \mid type^{\mathcal{F}}(n) = letBinding$
                                   $\wedge\ ine_{\mathcal{D}}(n) = \emptyset\}$
    for each node $n \in Garbage$
        $\mathcal{G} := deleteFrom(n, \mathcal{G})$
until $Garbage = \emptyset$
return $\mathcal{G}$
```

Algorithm 2 deleteFrom Function

```
Function $deleteFrom(n, \mathcal{G} = (\mathcal{N}, \mathcal{E} = (\mathcal{C}, \mathcal{S}, \mathcal{D}), \mathcal{F}))$
    for each node $n_s \in (outn_{\mathcal{S}}(n) \cup inn_{\mathcal{C}}(n))$
        $\mathcal{G} := deleteFrom(n_s, \mathcal{G})$
    $\mathcal{E} := \mathcal{E} \backslash (ine_{\mathcal{S} \cup \mathcal{D}}(n) \cup oute_{\mathcal{D} \cup \mathcal{C}}(n))$
    $\mathcal{N} := \mathcal{N} \backslash \{n\}$
return $\mathcal{G}$
```

Note that, the removal of a *letBinding* node (and all its related nodes) is implemented with function *deleteFrom* in Algorithm 2. This function starts from a given node and removes recursively all the nodes reachable from them, following structural edges forwards and control edges backwards; it also removes all their structural/data edges. Observe also that the application of this function could produce new garbage and thus the process is repeated until no garbage exists in the XDG.

5.2 Propagating dependences

This phase propagates data dependences through the XDG. Such propagation must be done forwards and backwards.

Roughly speaking, the forward propagation says what (sub) paths *are required* by the expressions in the XQuery. And the backward propagation says which of these (sub)paths *could be provided* to the expressions that required them. Basically, propagation is as follows: the data dependences are represented by means of labelled edges in which a partial path is requested by a certain (sub)expression. The forward propagation starts from initial nodes and follows structural and data edges in order to (1) update labelled data edges with **false** whenever the partial path cannot be obtained, (2) delete useless data edges, and (3) add new data edges. The backward propagation updates the data edges from the forward propagation.

1. **Forward Algorithm** (see Algorithm 3):

   (i) The forward algorithm starts from the initial nodes and propagates forward partial paths of data edges.

Algorithm 3 Propagating Dependences Forwards

```
Input: A XDG $\mathcal{G}$
Output: A XDG $\mathcal{G}'$

for each node $n \in init(\mathcal{G})$ (i)
  $\mathcal{G} := propagateForward(n, \mathcal{G})$
return $\mathcal{G}$

Function propagateForward(n,$\mathcal{G}$=($\mathcal{N}$,($\mathcal{C}$,$\mathcal{S}$,$\mathcal{D}$),$\mathcal{F}$))
  $nl := \emptyset$
  for each edge $e \in ine_{\mathcal{D}}(n)$
    for each tuple $(pp, bool) \in label^{\mathcal{F}}(e)$
      $fst := first(pp)$
      $suf := suffix(pp)$
      case $type^{\mathcal{F}}(n)$ of (ii)
      $tag$: if $fst \in \{tag^{\mathcal{F}}(n), \$\}$
                then  $nl := nl \cup \{(suf, \mathbf{true}) \mid suf \neq ``"\}$
                                $\cup\{(\$, \mathbf{true}) \mid suf = ``"\}$
                else $label^{\mathcal{F}}(e) := (label^{\mathcal{F}}(e) \backslash \{(pp, bool)\})$
                                $\cup\{(pp, \mathbf{false})\}$
      $path$: if $fst \in \{last(path^{\mathcal{F}}(n)), \$\}$
              then $nl := nl \cup \{(path^{\mathcal{F}}(n)/suf, \mathbf{true})\}$
              else  $label^{\mathcal{F}}(e) := (label^{\mathcal{F}}(e) \backslash \{(pp, bool)\})$
                                $\cup\{(pp, \mathbf{false})\}$
      $literal, op$: if  $pp \neq \$$
                then  $label^{\mathcal{F}}(e) := (label^{\mathcal{F}}(e) \backslash \{(pp, bool)\})$
                                $\cup\{(pp, \mathbf{false})\}$
      $seq$: if $oute_{\mathcal{S}}(n) = \emptyset$
             then $label^{\mathcal{F}}(e) := (label^{\mathcal{F}}(e) \backslash \{(pp, bool)\})$
                                $\cup\{(pp, \mathbf{false})\}$
      $otherwise$: $nl := nl \cup \{(pp, \mathbf{true})\}$
  if $nl = \emptyset$ then $\mathcal{D} := \mathcal{D} \setminus oute_{\mathcal{D}}(n)$ (iv)
                  for each node $n_{child} \in outn_{\mathcal{S}}(n)$
                    $deleteFrom(n_{child}, \mathcal{G})$
                  return $\mathcal{G}$
  else for each edge $e \in oute_{\mathcal{S} \cup \mathcal{D}}(n)$ (iii)
          $\mathcal{D} := \mathcal{D} \cup \{e\}$
          $label^{\mathcal{F}}(e) := nl$
       for each node $n_{child} \in outn_{\mathcal{S} \cup \mathcal{D}}(n)$
          $\mathcal{G} := propagateForward(n_{child}, \mathcal{G})$
       return  $\mathcal{G}$
```

- **(ii)** For each data edge and each partial path of a data edge, it proceeds depending on the type of the node. Whenever the requested partial path does not match with the node, that is, for instance, the requested partial path is $p/...$ and the node has the form $/.../q$ or $< q > ... < /q >$, $p \neq q$, then it updates the partial path $p/...$ to **false**, that is, it adds the label ($p/...$,**false**) to the edge. Otherwise, it propagates forward **true** following the data and structural edges.
- **(iii)** In addition, the forward propagation has to update partial paths. For instance, $p/q/r/...$ is propagated to the children of p/q by means of the partial path $r/....$
- **(iv)** When the forward propagation following data edges is not possible (e.g., we require a path p/q from an element p/r with $q \neq r$), the algorithm uses the auxiliary function *deleteFrom* (see Algorithm 2) to remove those nodes that are known to be useless.

Algorithm 4 Propagating Dependences Backwards

```
Input: A XDG $\mathcal{G} = (\mathcal{N}, (\mathcal{C}, \mathcal{S}, \mathcal{D}), \mathcal{F})$
Output: A XDG $\mathcal{G}'$

$Pending := \{n \in \mathcal{N} \mid ine_{\mathcal{D}}(n) \neq \emptyset \wedge oute_{\mathcal{D}}(n) \neq \emptyset\}$ (i)
for each node $n \in Pending$  :  $Pending \cap outn_{\mathcal{D}}(n) = \emptyset$
(ii)
  $\mathcal{G} := propagateBackward(n, \mathcal{G})$
  $Pending$  $:= Pending \backslash \{n\}$
return $\mathcal{G}$

Function propagateBackward(n,($\mathcal{N}$,($\mathcal{C}$,$\mathcal{S}$,$\mathcal{D}$),$\mathcal{F}$))
  nl:=$\{(pp, \mathbf{true}) \in label^{\mathcal{F}}(e) \mid e \in oute_{\mathcal{D}}(n)\} \cup$
      $\{(pp, \mathbf{false}) \in label^{\mathcal{F}}(e) \mid e \in oute_{\mathcal{D}}(n) \wedge$
                $\forall e' \in oute_{\mathcal{D}}(n) : \nexists (pp, \mathbf{true}) \in label^{\mathcal{F}}(e')\}$
  for each edge $e \in ine_{\mathcal{D}}(n)$ (iii)
    for each tuple $(pp, bool) \in label^{\mathcal{F}}(e)$
      $suf := suffix(pp)$
        case $type^{\mathcal{F}}(n)$ of
        $tag$: $new := \{(pp, bool') \mid (suf, bool') \in nl\}$
        $path$: $new := \{(pp, bool') \mid$
                      $(path^{\mathcal{F}}(n)/suf, bool') \in nl\}$
        $otherwise$: $new := \{(pp, bool') \mid (pp, bool') \in nl\}$
      $label^{\mathcal{F}}(e) := (label^{\mathcal{F}}(e) \backslash \{(pp, bool)\}) \cup new$
  return $(\mathcal{N}, (\mathcal{C}, \mathcal{S}, \mathcal{D}), \mathcal{F})$
```

2. **Backward Algorithm** (see Algorithm 4):

   - **(i)** The backward algorithm updates backwards the labelled data edges from the forward propagation. It traverses all the nodes which are source and target of a data edge (the set represented with *Pending*).
   - **(ii)** However, it must be done with a certain order. Concretely, a node is analysed whenever its adjacent nodes have been already analysed (i.e., $n \in Pending$ such that $Pending \cap outn_{\mathcal{D}}(n) = \emptyset$).
   - **(iii)** The algorithm updates to **true** and **false** the dependences obtained with the forward algorithm. In addition, the updating has to rebuild partial paths, e.g., a partial path $r/...$ in a node p/q is propagated as $p/q/r/....$

Let us remark that both propagation algorithms can be performed in linear time with respect to the size of the XDG.

EXAMPLE 5. *In Figure 4 we can see the forward and backward propagation of the normalized query in Example 3. $init(\mathcal{G}) = 0$, thus the forward propagation starts in node 0, propagating A until node 7. Then, B is propagated until nodes 15, 18 and 19 because only* `customer` *is required (i.e., the variable $j is paired with* sales*). Because node 19 only provides* `provider` *elements, and node 18 is the empty sequence, they cannot provide* `customer` *and thus they are updated to* **false** *(C). Moreover, because these nodes cannot provide required elements, the dependences that start from them are deleted. Note that the data edge from node 19 to node 2 has been deleted. The dependences (A) and*

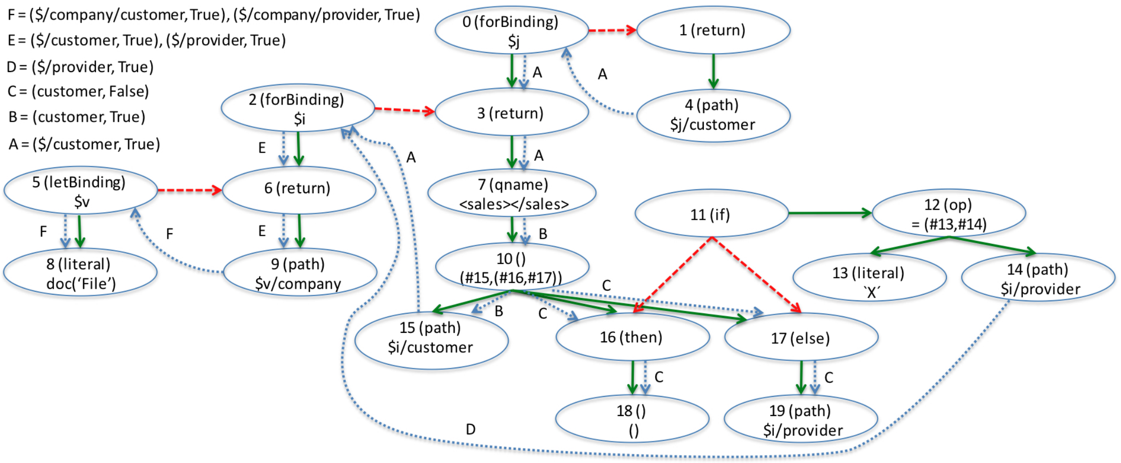


Figure 4: Forward and backward propagation of the running example

```
Algorithm 5 Slicing
  Input: A XDG $\mathcal{G} = (\mathcal{N}, (\mathcal{C}, \mathcal{S}, \mathcal{D}), \mathcal{F})$
  Output: A pruned XDG $\mathcal{G}'$
  $Pending := \{n \in \mathcal{N} \mid$
  $(ine_{\mathcal{D}}(n) \neq \emptyset \wedge \forall e \in ine_{\mathcal{D}}(n) : \nexists(pp, \mathbf{true}) \in label^{\mathcal{F}}(e))$
  $\bigvee$
  $(oute_{\mathcal{D}}(n) \neq \emptyset \wedge \forall e \in oute_{\mathcal{D}}(n) : \nexists(pp, \mathbf{true}) \in label^{\mathcal{F}}(e))\}$
  (i)
  for each node $n \in Pending : Pending \cap outn_{\mathcal{S} \cup \mathcal{D}}(n) = \emptyset$
  (ii)
      $(\mathcal{G}, Pending') := slicingFrom(n, \mathcal{G})$ (iii)
      $\mathcal{C} := \mathcal{C} \backslash \{(n' \rightarrow n'') \in \mathcal{C} \mid n' \notin \mathcal{N} \vee n'' \notin \mathcal{N}\}$
      $Pending := (Pending \cup Pending') \backslash \{n\}$
  return $\mathcal{G}$
```

(D) arriving to node 2 are also propagated forward (E) until node 9. This means that we only require elements `customer` *and* `provider` *from the* `company` *elements provided by node 9. Therefore, the dependence F is updated from E to express that node 9 only needs* `company/customer` *and* `company/provider` *elements. This is then propagated until node 8. In the backward propagation, the (B) dependences between nodes 10-16 and 10-17 are updated to* **false** *(C).*

5.3 Slicing algorithm

Once the dependences of the XDG have been propagated, the optimization technique uses a program slicing-based algorithm to produce a new optimised query.

3. **Slicing Algorithm** (see Algorithm 5):

   (i) The slicing algorithm removes nodes and edges from the XDG according to the partial paths that have been set to **false** by the forward/backward propagation algorithm. Those nodes that have at least one data edge and have all incoming or outgoing data edges with all paths labeled with **false** are removed. And, moreover, all nodes reachable from these nodes following structural and control edges are also removed.

   (ii) Nodes have to be analysed in a certain order to ensure efficiency: a node is analysed whenever the adjacents have been already analysed (i.e., $Pending \cap outn_{\mathcal{S} \cup \mathcal{D}}(n) = \emptyset$).

   (iii) The slicing algorithm uses function *slicingFrom* shown in Algorithm 6 (which in turn uses the auxiliary functions of Algorithms 2 and 7). Function *slicingFrom* is the responsible to remove nodes and edges, and it updates the set *Pending*. It distinguishes cases depending on the type of the node.

   Basically, it accurately removes the nodes of *Pending* following backward control edges (e.g., if a node of type *return* must be removed, then the associated nodes of types *where* and *forBinding* are also removed). An exception is *if-then-else* (removing a node of type *then* does not implies removing its associated node of type *if*). When a node is removed, all nodes reachable from it following structural edges are also removed. In some cases, the elimination of a node requires to rebuild the graph by replacing the children in the position of the parent, this is the functionality of the *replaceByChildren* function (see Algorithm 7).

Example 6. *In the XDG of Figure 4 the slicing process starts from nodes 18 and 19 because all incoming data edges are labelled with* **false***. In particular, the set Pending contains nodes 18 and 19 and thus Algorithm 5 performs, e.g., a call* $slicingFrom(18, \mathcal{G})$ *being* $\mathcal{G}$ *the XDG of Figure 4. Then, the case* $type^{\mathcal{F}}(n) =$ **seq** *of Algorithm 6 is executed with* $noNext =$ **true** *and hence, node 18 is removed and node 16 is included in pending. Then, e.g., it performs a call* $slicingFrom(19, \mathcal{G})$ *and the last* **if** *of Algorithm 6 is executed because* $allTrue = \emptyset$ *and hence, function* $deleteFrom$ *removes node 19. The next calls* $slicingFrom(16, \mathcal{G})$ *and* $slicingFrom(17, \mathcal{G})$ *produce the complete removal of the whole if-then-else. After unnormalization, the final result produced is optimal:*

```
Q1 =    for $i in (doc('File')/company)
        return <sales>{$i/customer}</sales>
Q2 =    for $j in Q1
        return $j/customer
```

Algorithm 6 SlicingFrom Function

```
Function slicingFrom(n,$\mathcal{G}$=($\mathcal{N}$,$\mathcal{E}$=($\mathcal{C}$,$\mathcal{S}$,$\mathcal{D}$),$\mathcal{F}$))
  $next := outn_{\mathcal{S}}(n)$
  $noNext := (next = \emptyset$
                  $\bigvee(next = \{n_{next}\} \wedge type^{\mathcal{F}}(n_{next}) = \mathsf{seq}$
                    $\wedge outn_{\mathcal{S}}(n_{next}) = \emptyset))$
  $allTrue := \{e \in (ine_{\mathcal{D}}(n) \cup oute_{\mathcal{D}}(n)) \mid \forall(pp, bool) \in$
                  $label^{\mathcal{F}}(e) : bool = \mathbf{true}\}$
  if $type^{\mathcal{F}}(n) = \mathsf{forBinding}$
    $\bigwedge(noNext \vee (allTrue = \emptyset \wedge ine_{\mathcal{D}}(n) \neq \emptyset))$
   then let $n_r \in \mathcal{N} : ((n \rightarrow^* n_r) \in \mathcal{C} \wedge type^{\mathcal{F}}(n_r) = \mathsf{return})$
        return $(deleteFrom(n_r, \mathcal{G}), inn_{\mathcal{S}}(n_r))$
  if $type^{\mathcal{F}}(n) = \mathsf{letBinding} \wedge noNext$
  then if $\exists n_w, n_r \in \mathcal{N} : (n \rightarrow n_w), (n_w \rightarrow n_r) \in \mathcal{C}$
        then $\mathcal{G} := deleteFrom(n_w, \mathcal{G})$
        else $\mathcal{G} := deleteFrom(n, \mathcal{G})$
              let $n_r \in \mathcal{N} : (n \rightarrow n_r) \in \mathcal{C}$
        return $(replaceByChildren(n_r, \mathcal{G}), \emptyset)$
  if $type^{\mathcal{F}}(n) = \mathsf{where} \wedge noNext$
  then let $n_p, n_r \in \mathcal{N} : (n_p \rightarrow n), (n \rightarrow n_r) \in \mathcal{C}$
        if $type^{\mathcal{F}}(n_p) \in \{\mathsf{forBinding}, \mathsf{letBinding}\}$
        then return $(deleteFrom(n_r, \mathcal{G}), inn_{\mathcal{S}}(n_r))$
        else $\mathcal{G} = deleteFrom(n, \mathcal{G})$
        return $(replaceByChildren(n_r, \mathcal{G}), \emptyset)$
  if $type^{\mathcal{F}}(n) = \mathsf{return} \wedge noNext$
  then return $(deleteFrom(n, \mathcal{G}), inn_{\mathcal{S}}(n))$
  if $type^{\mathcal{F}}(n) = \mathsf{if} \wedge noNext$
  then let $n_t \in \mathcal{N} : ((n \rightarrow n_t) \in \mathcal{C} \wedge type^{\mathcal{F}}(n_t) = \mathsf{then})$
        let $n_e \in \mathcal{N} : ((n \rightarrow n_e) \in \mathcal{C} \wedge type^{\mathcal{F}}(n_e) = \mathsf{else})$
        let $n_{child} \in \mathcal{N} : (n_t \rightarrow n_{child}) \in \mathcal{S}$
        $\mathcal{G} := deleteFrom(n_{child}, \mathcal{G})$
        $\mathcal{G} := (\mathcal{N} \backslash \{n, n_t\}, \mathcal{E} \backslash oute_{\mathcal{E}}(n), \mathcal{F})$
        return $(replaceByChildren(n_e, \mathcal{G}),\ \emptyset)$
  if $type^{\mathcal{F}}(n) \in \{\mathsf{then}, \mathsf{else}\} \wedge noNext$
  then $type^{\mathcal{F}}(n_f) := \mathsf{seq}$ and $children^{\mathcal{F}}(n_f) = \emptyset$
        where $n_f$ is a new node
        $\mathcal{G} = (\mathcal{N} \cup \{n_f\}, (\mathcal{C}, \mathcal{S} \cup \{(n \rightarrow n_f)\}, \mathcal{D}), \mathcal{F})$
        let $n_{if} \in \mathcal{N} : (n_{if} \rightarrow n) \in \mathcal{C}$
       let $n_t \in \mathcal{N} : ((n_{if} \rightarrow n_t) \in \mathcal{C} \wedge type^{\mathcal{F}}(n_t) = \mathsf{then})$
       let $n_e \in \mathcal{N} : ((n_{if} \rightarrow n_e) \in \mathcal{C} \wedge type^{\mathcal{F}}(n_e) = \mathsf{else})$
        let $n_{child_t} \in \mathcal{N} : (n_t \rightarrow n_{child_t}) \in \mathcal{S}$
        let $n_{child_e} \in \mathcal{N} : (n_e \rightarrow n_{child_e}) \in \mathcal{S}$
        let $n_{parent_t} \in \mathcal{N} : (n_{parent_e} \rightarrow n_t) \in \mathcal{S}$
        if $type^{\mathcal{F}}(n_{child_t}) = type^{\mathcal{F}}(n_{child_e}) = \mathsf{seq}$
          $\wedge\ children^{\mathcal{F}}(n_{child_t}) = children^{\mathcal{F}}(n_{child_e}) = \emptyset$
        then $\mathcal{G} := deleteFrom(n_t, \mathcal{G})$
              $\mathcal{G} := deleteFrom(n_e, \mathcal{G})$
              $\mathcal{G} := (\mathcal{N} \cup \{n_f\},$
       $(\mathcal{E} \backslash ine_{\mathcal{E}}(n_t) \cup ine_{\mathcal{E}}(n_e)) \cup (parent_t \rightarrow n_f) \in \mathcal{S}, \mathcal{F})$
        return $(\mathcal{G}, \emptyset)$
  if $type^{\mathcal{F}}(n) = \mathsf{op}$
  then if $next = \{n_{op}\} \wedge op^{\mathcal{F}}(n) = or$
        then return $(replaceByChildren(n, \mathcal{G}), \emptyset)$
        if $next \neq \{n_{op1}, n_{op2}\}$
       then return $(deleteFrom(n, (\mathcal{N}, \mathcal{E}, \mathcal{F})), inn_{\mathcal{S}}(n))$
  if $type^{\mathcal{F}}(n) = \mathsf{seq}$
  then if $noNext$
        then return $((\mathcal{N} \backslash \{n\}, \mathcal{E} \backslash ine_{\mathcal{E}}(n), \mathcal{F}), inn_{\mathcal{S}}(n))$
        if $next = \{n_{child}\}$
        then return $(replaceByChildren(n, \mathcal{G}), \emptyset)$
        return $(\mathcal{G}, \emptyset)$
  if $allTrue = \emptyset$
  then return $(deleteFrom(n, \mathcal{G}), inn_{\mathcal{S}}(n))$
return $(\mathcal{G}, \emptyset)$
```

Algorithm 7 replaceByChildren Function

```
Function $replaceByChildren(n, (\mathcal{N}, \mathcal{E}, \mathcal{F}))$
  $\mathcal{E} := \mathcal{E} \backslash ine_{\mathcal{C}}(n)$
  for each $n' \in \mathcal{N} : (n \rightarrow n') \in \mathcal{S}$
    $\mathcal{E} := \mathcal{E} \cup \{(n_p \rightarrow n') | (n_p \rightarrow n) \in \mathcal{E}\}$
    $\mathcal{E} := \mathcal{E} \backslash \{(n \rightarrow n') \in \mathcal{E}\}$
  $\mathcal{N} := \mathcal{N} \backslash \{n\}$
return $(\mathcal{N}, \mathcal{E}, \mathcal{F})$
```

5.4 Projecting source documents

When the forward propagation process is finished, we can check all the data dependences of those nodes that represent an XML document (i.e., those labeled with *doc(Literal)*). These data dependences are a collection of paths that represent the information required by the query from this particular XML document. The projection $\mathcal{P}$ of the XML documents can be extracted from the XDG as follows:

$$\mathcal{P} = \{(literal^{\mathcal{F}}(n), \{pp \mid e \in ine_{\mathcal{D}}(n) \wedge (pp, true) \in label^{\mathcal{F}}(e)\}) \mid n \in \mathcal{N} \wedge type^{\mathcal{F}}(n) = doc\ \}$$

Each one of the computed pairs contains an input XML file and the paths required from this file.

The projection of XML documents can be done at any stage after the forward propagation. If it is computed immediately after the forward propagation, the result is equivalent to the projecting technique in [17]. If we compute it after the slicing phase, the result is much more precise because the projection takes advantage of the pruning analysis.

For instance, with the query of Example 2, the projection information that we get after the forward propagation is: $\{(File1, \emptyset), (File2, \{site/people/person\})\}$

In contrast, the projecting information after the slicing phase is the empty set (no information is really needed from the XML sources).

6. IMPLEMENTATION

All the algorithms proposed have been implemented and integrated into a tool called *XQSlicer*. This tool allows us to automatically generate an XDG from a given XQuery expression. The tool has been implemented in Haskell. It has about 1000 LOC and generates XDGs in `dot` and `jpg` formats. The implementation is composed of eight different modules that interact to produce the final XDG:

Main This is the main module that coordinates all the other modules.

Parse Module used to transform XQueries to an internal representation.

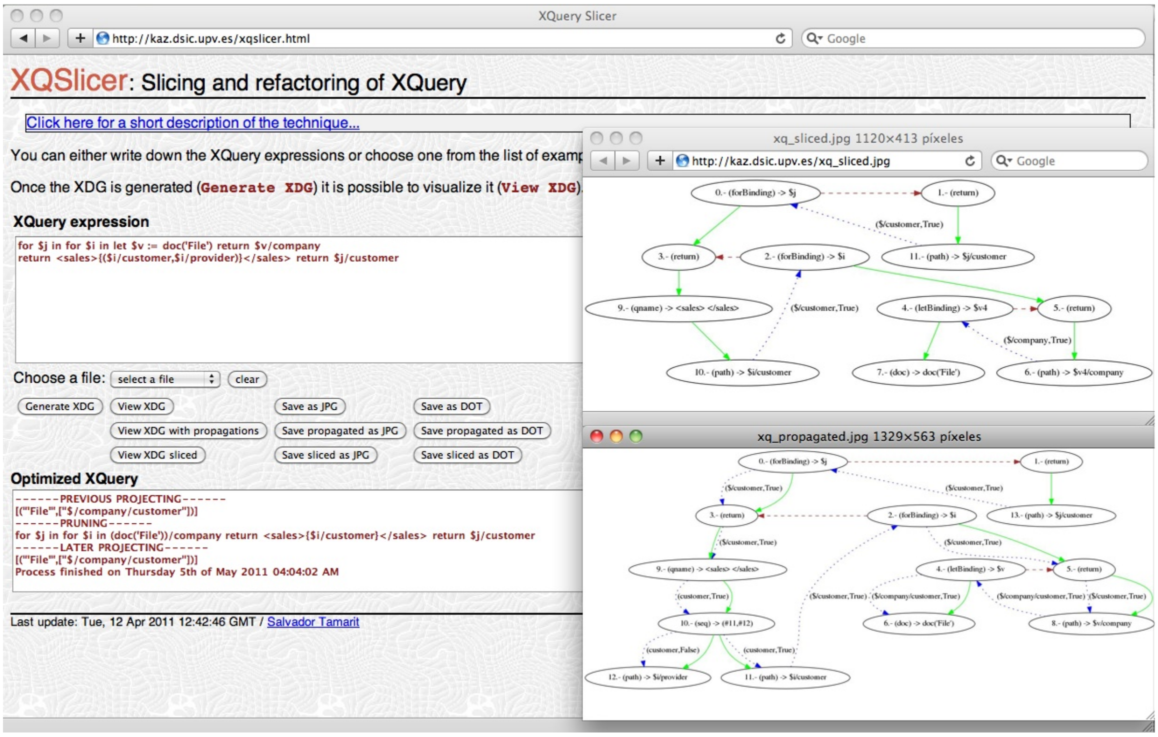
Normalization This module automatically normalizes the expression introduced by the user.

Graph Creation Creates the initial graph (i.e., without propagation) according to Definition 1.

Propagation Performs backward and forward propagation as defined in Algorithms 3 and 4.

Slice Used to obtain the sliced graph after data propagation. It implements Algorithm 5.

Restore Contains all the needed functionality to produce the XQuery associated to a given XDG.



XQSlicer: Slicing and refactoring of XQuery

Click here for a short description of the technique...

You can either write down the XQuery expressions or choose one from the list of examp

Once the XDG is generated (Generate XDG) it is possible to visualize it (View XDG).

XQuery expression

```
for $j in for $i in let $v := doc('File') return $v/company
return <sales>{($i/customer,$i/provider)}</sales> return $j/customer
```

Optimized XQuery

```
------PREVIOUS PROJECTING------
[("'File'",["$/company/customer"])]
------PRUNING------
for $j in for $i in (doc('File'))/company return <sales>{$i/customer}</sales> return $j/customer
------LATER PROJECTING------
[("'File'",["$/company/customer"])]
Process finished on Thursday 5th of May 2011 04:04:02 AM
```

Last update: Tue, 12 Apr 2011 12:42:46 GMT / Salvador Tamarit

Figure 5: Screenshot of the online version of *XQSlicer*

Dot Generation To obtain a Dot file from a given XDG.

Utilities It contains common and auxiliary functions and tools used by the other modules.

There is an online version of *XQSlicer* that can be used to test the tool. This online version is publicly available at `http://kaz.dsic.upv.es/xqslicer.html`. Figure 5 shows a screenshot of the online version of *XQSlicer*. The user can either write down the initial XQuery expression or choose one from the list of available examples. Once the XDG is generated (Generate XDG) it is possible to visualize it (View XDG) and to save it as `jpg` or `dot` formats. The same options are available for the XDG with propagated dependences. And also for the sliced XDG. For instance, the XDGs in Figure 5 has been automatically generated by *XQSlicer* from the XQuery expression of Example 1. After the XDG is generated, the tool shows the final XQuery expression produced after the slicing process and the projecting information.

7. EXPERIMENTAL RESULTS

We tested our approach by means of the *BaseX* processor in a Intel machine of 1.60 GHz and 2 GB of RAM. We tested the following query:

```
for $j in <site>{
      for $i in (doc('File1')/site)
      where $i/people/person/@id = "X"
      return ($i/open_auctions/open_auction,
         for $k in (doc('File1')/site)
         return $k/closed_auctions/closed_auction,
         $i/people/person)
      }</site>
   return
      for $k in doc('File2')/site
where $j/person = $k/people/person
return
<common-auction>{$j/open_auction}</common-auction>
```

We used benchmarks for XML documents with (1) 100, (2) 1000, (3) 10000 and (4) 100000 records. Table 1 shows the execution time in ms.

Table 1: Execution Time

# Records	Pruned	Original
(1) 100	41,05	131,86
(2) 1000	99,92	3028,35
(3) 10000	447,09	264619,42
(4) 100000	3171,67	> 1 hour

This result shows that the proposed technique not only allows us to save memory by avoiding the computation, storage, and transfer of intermediate data structures. It also allows us to scale up with respect to the time needed to process the query. Note that the impact on time is very significative (from ms. to hours in the last case).

8. CONCLUSIONS

This work introduces a program slicing based technique to automatically optimize XQuery expressions. This is the first adaptation of program slicing to XQuery and it has the advantage that the dependence analysis performed allows us to project the source XML documents and to prune the XQuery expression. The technique is based on a data structure, the XDG, that is the adaptation to XQuery of the well-known PDG used in imperative languages. The definition of the XDG is by itself an important contribution of this work because it allows us to perform many other different static analyses and refactoring transformations for XQuery expressions that are expressed with this formalism. The way in which we have defined the XDG for XQuery could be easily adapted to other functional languages. In fact, we think that the slicing algorithms could be also adapted with slight modifications.

One important advantage of our technique is that it produces XQuery expressions that are executable. This means that other analyses and tools could use our transformation as a preprocessing stage simplifying the initial query and producing a more accurate and reduced expression that would predictably speed up the subsequent transformations.

Our proposed slicing technique can be extended in the future with some optimizations. For instance, in [11] they propose a *rewriting-based* optimization technique for XQuery in which they change the order of operations checking boolean conditions before constructing XML elements, and computing statically path expressions when they are applied to XML element constructors. In [15] they study under which conditions *query composition* can be eliminated and show a set of rules to this end. We think that the XDG can be used to improve these transformations.

9. REFERENCES

[1] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyên. Type-based XML projection. In *Proceedings of the 32nd International Conference on Very Large Databases*, pages 271–282. VLDB Endowment, 2006.

[2] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0. *W3C*, 2007.

[3] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery - The Relational Way. In *Proc. of the International Conference on Very Large Databases*, pages 1322–1325, New York, USA, 2005. ACM Press.

[4] S. Bressan, B. Catania, Z. Lacroix, Y. Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data & Knowledge Engineering*, 54(2):211–240, 2005.

[5] C. Brown. *Tool Support for Refactoring Haskell Programs.* PhD thesis, School of Computing, University of Kent, Canterbury, Kent, UK, 2008.

[6] D. Chamberlin, D. Draper, M. Fernández, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler. *XQuery from the Experts.* Addison Wesley, Boston, USA, 2004.

[7] D. Cheda, J. Silva, and G. Vidal. Static slicing of rewrite systems. *Electron. Notes Theor. Comput. Sci.*, 177:123–136, June 2007.

[8] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *Proceedings of the 29th International Conference on Very Large Databases-Volume 29*, pages 1077–1080. VLDB Endowment, 2003.

[9] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[10] X. Franc. Qizx, a fast XML database engine fully supporting XQuery, 2003. `http://www.xmlmind.com/qizx/`.

[11] M. Grinev and M. Pleshachkov. Rewriting-based optimization for xquery transformational queries. In *Database Engineering and Application Symposium, 2005. 9th International*, pages 163–174. IEEE, 2005.

[12] C. Grün. BaseX. The XML Database, 2011. `http://basex.org`.

[13] B. Gueni, T. Abdessalem, B. Cautis, and E. Waller. Pruning nested XQuery queries. In *Proceeding of the 17th ACM Conference on Information and Knowledge Management*, pages 541–550. ACM, 2008.

[14] M. Kay. Ten reasons why Saxon XQuery is fast. *IEEE Data Eng. Bull*, 31(4):65–74, 2008.

[15] C. Koch. On the role of composition in XQuery. In *Proc. WebDB*, 2005.

[16] C. Koch, S. Scherzinger, and M. Schmidt. The GCX system: dynamic buffer minimization in streaming XQuery evaluation. In *Proceedings of the 33rd International Conference on Very Large Databases*, pages 1378–1381. VLDB Endowment, 2007.

[17] A. Marian and J. Simeon. Projecting XML Documents. In *Proc. of International Conference on Very Large Databases*, pages 213–224, Burlington, USA, 2003. Morgan Kaufmann.

[18] N. F. Rodrigues and L. S. Barbosa. Component identification through program slicing. In *In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS*, pages 291–304. Elsevier, 2005.

[19] S. Scherzinger. Bulk data in main memory-based XQuery evaluation. In *Proceedings of the 4th International Workshop on XQuery Implementation, Experience and Perspectives*, pages 1–6. ACM, 2007.

[20] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[21] W3C. XML Query Working Group and XSL Working Group, XQuery 1.0: An XML Query Language. Technical report, www.w3.org, 2007.

[22] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.