Pruning Nested XQuery Queries

Bilel Gueni
Talel Abdessalem, Bogdan Cautis, Emmanuel Waller
TELECOM ParisTech - Université de Paris-Sud
FRANCE
Introduction

- XQuery optimization is crucial and complex
  - Complex: operational semantics
  - Crucial: evaluation efficiency
    - Handling many document accesses is expensive
    - Construction of XML elements is expensive

- Challenge
  - Minimizing document access and intermediary element construction

- Various approaches
  - Projecting XML documents
  - Type based projections
  - Reformulating XQuery queries
  - ...

- Our contribution
  - Rewriting rules for nested XQuery queries with composition
Example of query composition

```
for $j in doc("auction.xml")/site/people/person
return <person>
  <webpage>${j/homepage}</webpage>
  <profile>${j/profile/*}</profile>
</person>
```
Example of query composition

let $i := (for $j in doc("auction.xml")/site/people/person
      return <person>
          <webpage>{$j/homepage}</webpage>
          <profile>{$j/profile/*}</profile>
      </person>)

return $i/webpage
Example of query composition

let $i := (for $j in doc(“auction.xml”)/site/people/person
  return <person>
    <webpage>{$j/homepage}</webpage>
    <profile>{$j/profile/*}</profile>
  </person>
return $i/webpage
Example of query composition

```
let $i := (for $j in doc("auction.xml")/site/people/person
    return <person>
        <webpage>{$j/homepage}</webpage>
    </person>)

return $i/webpage
```
Example of query composition

```
let $i := (for $j in doc("auction.xml")/site/people/person
return <person>
  <webpage>{$j/homepage}</webpage>
A FLWR or any sub-expression
</person>)

return $i/webpage
```

```
+-------------------+      +-------------------+      +-------------------+
| person            |      | person            |      | person            |
|                   |      |                   |      |                   |
| webpage           |      | A                 |      | Z                 |
|                   |      |                   |      |                   |
| homepage          |      |                   |      |                   |
|                   |      |                   |      |                   |
|                   |      |                   |      |                   |
```

ACM CIKM 2008, Napa Valley, CA.
The main idea

- In the case of query composition, some intermediate results/computations may not be necessary
  - Sub-expressions generating them are useless

- Our algorithm:
  - Analyzes the input query in order to detect these sub-expressions
  - Projects out (prune) these sub-expressions
  - Maintain equivalence with the original query
Preliminaries

- Intermediate results/computations
  - Associated to variables

- For each variable $i$
  - its definition denotes the sub-expression producing the results “stored” in $i$
  - its scope denotes the sub-expressions where $i$ can be used
Preliminaries

- Intermediate results/computations
  - Associated to variables

- For each variable $i$
  - Its **definition** denotes the sub-expression producing the results “stored” in $i$
  - Its **scope** denotes the sub-expressions where $i$ can be used

let $i := \textit{definition of } i$ for $i$ in \textit{definition of } i

where \textit{part of the scope of } i

return \textit{part of the scope of } i

where \textit{part of the scope of } i

return \textit{part of the scope of } i
Preliminaries

- Intermediate results/computations
  - Associated to variables

- For each variable $i$
  - its **definition** denotes the sub-expression producing the results “stored” in $i$
  - its **scope** denotes the sub-expressions where $i$ can be used

```
let $i := \text{definition of } i$               for $i$ in \text{definition of } i$
where \text{part of the scope of } i         where \text{part of the scope of } i
return \text{part of the scope of } i          return \text{part of the scope of } i

based on the references to $i$ in the scope, simplify its definition
```
Algorithm overview

A bottom-up static analysis of input query

For each variable $i$, three main steps:

1. Apply recursively the pruning process on its definition and scope

2. Extract from the scope the paths referencing $i$
   Explicit references: $i/A/B/…$
   Implicit references: $k/A/B/…$, while $k$ is linked to $i$
   (ex. for $k$ in $i/…$, or let $k:=i/…$)

3. Using the paths from Step 2, detect and project out the unneeded parts from the definition of $i$
Path extraction

- Returned vs used paths [Marian and Simeon, VLDB’03]
  - Paths determine XML result nodes
  - In some cases (used paths), the subtrees rooted at these result nodes are not necessary for the query evaluation
    For example, when the result nodes are only used for iteration
  - `extractPaths($i, scope of $i) => P, P#`
    - Analyze the scope of $i
    - Return the paths referencing $i: P for used paths, and P# for returned paths
Path extraction

- Returned vs used paths [Marian and Simeon, VLDB’03]
  - Paths determine XML result nodes
  - In some cases (used paths), the subtrees rooted at these result nodes are not necessary for the query evaluation
    For example, when the result nodes are only used for iteration
  
- `extractPaths($i, scope of $i) => P, P#`
  - Analyze the scope of $i
  - Return the paths referencing $i: P for used paths, and P# for returned paths
  - Described in the paper by a set of inference rules
    each rule details the path extraction process according to the query structure of the input scope
Pruning $i$’s definition

\text{projectPaths}(P, P#, \text{$i$’s definition}) \Rightarrow \text{definition’}

- Analyze the definition of $i$ and its expected result
- Determine the result nodes that are reached by some paths in $P$ and $P#$
  - Those that are not reached are unneeded
- Determine whether the descendent nodes are necessary or not
  If a node is only reached by used paths, its descendents are not needed
- Project out the sub-expressions that compute the unneeded nodes (and nothing else)
Pruning $i$’s definition

projectPaths(P, P#, $i$’s definition) => definition’

- Analyze the definition of $i$ and its expected result
- Determine the result nodes that are reached by some paths in P and P#
  - Those that are not reached are unneeded
- Determine whether the descendent nodes are necessary or not
  - If a node is only reached by used paths, its descendents are not needed
- Project out the sub-expressions that compute the unneeded nodes (and nothing else)
- Detailed in the paper by a set of rewriting rules
  - each rule describes the path projection process according to the query structure of the input definition
Pruning rule example

FLWR expression:

\[
\begin{align*}
\text{Env} & |- \text{prune}(e_1) \Rightarrow e'_1 \\
\text{Env.add}(\$i) \\
\text{Env} & |- \text{prune}(e_2) \Rightarrow e'_2 \\
\text{Env} & |- \text{extractPaths}(\$i, e'_2) \Rightarrow P, P# \\
\text{Env} & |- \text{projectPaths}(P, P#, e'_1) \Rightarrow e''_1 \\
\text{Env} & |- \text{prune(}\text{let \$i := e_1 \ return e_2 \}) \Rightarrow \text{let \$i := e''_1 \ return e'_2} 
\end{align*}
\]
Pruning rule example

FLWR expression:

\[
\begin{align*}
\text{Env} &\vdash \text{prune}(e_1) \Rightarrow e'_1 \\
\text{Env.add}(&\$i) \\
\text{Env} &\vdash \text{prune}(e_2) \Rightarrow e'_2 \\
\text{Env} &\vdash \text{extractPaths}(i, e'_2) \Rightarrow P, P# \\
\text{Env} &\vdash \text{projectPaths}(P, P#, e'_1) \Rightarrow e''_1
\end{align*}
\]

\[
\text{Env} \vdash \text{prune(let } i := e_1 \text{ return } e_2 ) \Rightarrow \text{let } i := e''_1 \text{ return } e'_2
\]
Pruning rule example

FLWR expression:

Env |- prune(e₁) => e'₁
Env.add($i)
Env |- prune(e₂) => e'₂
Env |- extractPaths($i, e₂') => P, P#
Env |- projectPaths(P, P#, e'₁)) => e''₁
Env |- prune(let $i := e₁ return e₂ ) => let $i := e''₁ return e'₂
Pruning rule example

FLWR expression:

\[
\begin{align*}
\text{Env } &\vdash \text{prune}(e_1) \Rightarrow e_1' \\
\text{Env.add($i$)} \\
\text{Env } &\vdash \text{prune}(e_2) \Rightarrow e_2' \\
\text{Env } &\vdash \text{extractPaths($i, e_2'$))} \Rightarrow P, P# \\
\text{Env } &\vdash \text{projectPaths(P, P#, e_1'))} \Rightarrow e_1'' \\
\text{Env } &\vdash \text{prune(let $i := e_1 return e_2 )} \Rightarrow \text{let $i := e_1'' return e_2'$}
\end{align*}
\]
Pruning rule example

FLWR expression:

\[
\text{Env} |- \text{prune}(e_1) \Rightarrow e'_1 \\
\text{Env}.\text{add}(i) \\
\text{Env} |- \text{prune}(e_2) \Rightarrow e'_2 \\
\text{Env} |- \text{extractPaths}(i, e'_2) \Rightarrow P, P# \\
\text{Env} |- \text{projectPaths}(P, P#, e'_1) \Rightarrow e''_1 \\
\text{Env} |- \text{prune}(\text{let } i := e_1 \text{ return } e_2) \Rightarrow \text{let } i := e''_1 \text{ return } e'_2
\]
Pruning rule example

FLWR expression:

Env |- prune(e₁) => e'₁
    Env.add($i)
    Env |- prune(e₂) => e'₂ = ()
    Env |- extractPaths($i, e₂') => P, P#
    Env |- projectPaths(P, P#, e₁') => e''₁

Env |- prune(let $i := e₁ return e₂) => let $i := e''₁ return e'₂
            = ()
Pruning rule example

FLWR expression:

Env |- prune(e₁) => e'₁

Env.add($i)

Env |- prune(e₂) => ()

Env |- prune(let $i := e₁ return e₂ ) => ()
Equivalence

Theorem:
For q an XQuery expression, if $\text{Env} \vdash \text{prune}(q) \Rightarrow q'$
then $\forall I, q(I) =_{\text{deep}} q'(I)$.

- Proof by induction on the rewrite rules
  - Details can be found in the paper

- Intuition: the computations we remove are not “linked” to the root computations
Experiments

- We measured the impact on evaluation time

- Using a template query, we varied
  - the nature and complexity of pruned sub-expressions
    - FLWR blocks
    - explicit element construction
    - XPath expressions
  - the percentage of useless intermediate results
  - the size of the source document
Experiments

\[
\text{let } \mathbf{i} := \langle \text{personInf} \rangle \\
\text{for } j \text{ in doc("xmark.xml")/site/people/person} \\
\text{return } ( \langle \text{name}\rangle \{test\_expr\}\langle/\text{name}\rangle, \langle \text{age}\rangle \{test\_expr\}\langle/\text{age}\rangle, \\
\langle \text{gender}\rangle \{test\_expr\}\langle/\text{gender}\rangle, \langle \text{email}\rangle \{test\_expr\}\langle/\text{email}\rangle) \rangle \\
\langle/\text{personInf}\rangle \\
\text{return scope of } \mathbf{i}
\]
Conclusion

- A new technique for pruning nested XQuery queries with composition
- A rule based algorithm for a large XQuery fragment
  - FLWR expression, quantifier
  - Sequence, comparison operation, element construction
  - If-the-else expressions
  - ...
- Experiments show significant gains in evaluation time
- Useful in many contexts (data integration, security)
- Possible extensions:
  - backward axis
  - taking into account document schema
Thank you for your attention