Module 4

Implementation of XQuery

Part 1: Overview of Compiler, Runtime System
Now let us talk XQuery

- Compile Time + Optimizations
  - Operator Models
  - Query Rewrite
  - Runtime + Query Execution

- XML Data Representation
  - XML Storage
  - XML Indexes
  - Compression + Binary XML
Code representation

• For SQL, relational algebra
  – e.g., joins, scan, group-by, sort, …
  – logical and physical operators

• For XQuery, many proposals exist:
  – algebra (operators) vs expressions vs automata
  – standard algebra for XQuery (\rightarrow \text{XQuery Formal Sem.})
  – logical vs. physical algebra
  – redundant algebra or not
    • SQL is redundant at the physical not logical level (!)
  – additional structures: dataflow, dependency graphs
Automata representation

[YFilter ’03, Gupta ’03, etc]

$x/chapter//section/title$

- Many variants
  - one path vs. a set of paths
  - NFAs vs DFAs
- Limitations
  - not extensible to full XQuery
  - better suited for push execution, pull is harder
  - lazy evaluation is hard
TLC Algebra
(Jagadish et al. 2004)

• XML Query tree patterns (called *twigs*)
• Annotated with predicates
• Tree matching as basic operation
  – Logical and physical operation

• Tree pattern matching => tuple bindings
  (i.e. relations)
• Tuples combined via classical relational algebra
  – Select, project, join, duplicate-elim., …
XQuery Expressions
XQRL/BEA/Oracle, XL, MXQuery, Zorba / Sausalito

• “Expressions” built during parsing
• (almost) 1-1 mapping between XQuery expressions and internal expressions
  – exception: Match( expr, NodeTest) for path expressions
• Annotated expressions
  – *E.g. unordered* is an annotation
  – Annotations exploited during optimization
• Redundant algebra
  – general FLWR, but also LET and MAP
  – typeswitch, but also instanceof and conditionals
  – many different versions of constructor
    • streaming vs. blocking; recycling of constructed nodes; node ids
• Support for dataflow analysis is fundamental
Expression representation example

- First "normalize" query – make implicit operations explicit

for $line in $doc/Order/OrderLine
  where $line/SellersID eq 1
  return <lineltem>{$line/Item/ID}</lineltem>

for $line in $doc/Order/OrderLine
  where xs:integer(fn:data($line/SellersID)) eq 1
  return <lineltem>{$line/Item/ID}</lineltem>
Translation to expression tree

- Optimization: Transformations on expression tree
- Code gen: Select physical implementation for each expr.
Dataflow Analysis

• **Annotate each operator (attribute grammars)**
  – Type of output (e.g., BookType*)
  – Is output sorted? Does it contain duplicates?
  – Has output node ids? Are node ids needed?

• **Annotations computed in walks through plan**
  – Intrinsic: e.g., preserves sorting
  – Synthetic: e.g., type, sorted
  – Inherited: e.g., node ids are required

• **Optimizations based on annotations**
  – Eliminate redundant sort operators
  – Avoid generation of node ids in streaming apps
Dataflow Analysis: Static Type

Match("book")

elem book of BookType

elem book of BookType
or
elem thesis of BookType

elem bib of BibType

doc of BibType

validate as "bib.xsd"

doc("bib.xml")

item*
Order, Duplicate Annotations

• Program: $doc/a/b

• Implicit operators of Xpath
  – sort in document order
  – eliminate duplicates

• Very expensive operations
  – do not do them if unnecessary
  – do not worry about node-ids if no necessary

• Example also shows need for different implementations, algebraic properties of operators
  – dup-elim before / after sort???
Order, Duplicate Annotations: $doc/a/b$

Order = ?, Duplicates = no
Order, Duplicate Annotations: $doc/a/b$

Order = ?, Duplicates = no
Order = ?, Duplicates = no
Order, Duplicate Annotations: $doc/a/b$

Order = ?, Duplicates = no
Order = ?, Duplicates = no
Order = ?, Duplicates = no

- dup-elim
- sort(id)
- Match(“a“)
- FO:Child
- dup-elim
- sort(id)
- Match(“a“)
- FO:Child
- Var($doc$)
Order, Duplicate Annotations: $doc/a/b$

**Order = yes, Duplicates = no**

**Order = ?, Duplicates = no**

**Order = ?, Duplicates = no**

**Order = ?, Duplicates = no**
Order, Duplicate Annotations: $doc/a/b$

- Order = yes, Duplicates = no
- Order = yes, Duplicates = no
- Order = yes, Duplicates = no
- Order = ?, Duplicates = no
- Order = ?, Duplicates = no
- Order = ?, Duplicates = no

- dup-elim
- sort(id)
- Match("a")
- FO:Child
- Var($doc$)
Order, Duplicate Annotations: $doc/a/b$

**Order = yes, Duplicates = no**

**Order = yes, Duplicates = no**

**Order = yes, Duplicates = no**

**Order = ?, Duplicates = no**

**Order = ?, Duplicates = no**

**Order = ?, Duplicates = no**
Order, Duplicate Annotations: $\text{doc/a/b}$

Order = yes, Duplicates = no
- dup-elim

Order = yes, Duplicates = no
- sort(id)

Order = yes, Duplicates = no
- Match("a")

Order = yes, Duplicates = no
- FO:Child

Order = yes, Duplicates = no
- dup-elim

Order = yes, Duplicates = no
- sort(id)

Order = ?, Duplicates = no
- Match("a")

Order = ?, Duplicates = no
- FO:Child

Order = ?, Duplicates = no
- Var($doc$)
Optimizing: $\text{doc}/a/b$

- Order = yes, Duplicates = no
  - dup-elim
- Order = yes, Duplicates = no
  - sort(id)
- Order = yes, Duplicates = no
  - Match("a")
- Order = yes, Duplicates = no
  - FO:Child
- Order = yes, Duplicates = no
  - dup-elim
- Order = yes, Duplicates = no
  - sort(id)
- Order = ?, Duplicates = no
  - Match("a")
- Order = ?, Duplicates = no
  - FO:Child
- Order = ?, Duplicates = no
  - Var($\text{doc}$)
How about $\text{doc} // \text{a} // \text{b}$

- Does „//“ preserve order?
- Does „//“ generate duplicates?
- How would you implement „//“
  - under which circumstances can you stream it?
  - under which circumstances do you have to materialize?
- Properties of „//“ depend on
  - algorithm used to compute „//“
  - knowledge of the types
Architecture of XQuery Processor

Compilation

Query

Parsing & Verification

Internal query/program representation

Code rewriting

Lower level internal query representation

Code generation

Executable code

Data access pattern (APIs)
Major compilation steps

1. Parsing
2. Normalization
3. Type checking
4. **Optimization**
   1. *Data access patterns agnostic optimization*
   2. Optimization that exploit the existing data access patterns
   3. *(Cost-based optimizations)*
5. Code Generation
XQuery Rewritings

• Algebraic properties of comparisons
• Algebraic properties of Boolean operators
• LET clause folding and unfolding
• Function inlining
• Constant folding
• Common sub-expressions factorization
• Type based rewritings
• Navigation based rewritings
Algebraic properties of comparisons

• General comparisons not reflexive, transitive
  – \((1,3) = (1,2)\) (but also \(!=\), \(<\), \(>\), \(\leq\), \(\geq\) !!!!!!)
  – Reasons
    • implicit existential quantification, dynamic casts

• Negation rule does not hold
  – \(\text{fn:not}($x = $y)\) is not equivalent to \($x \neq $y\)

• Value comparisons are \textit{almost} transitive
  – Exception:
    • \texttt{xs:decimals} due to the loss of precision

\textbf{Impact on grouping, hashing, indexing, caching !!!}
Properties of Boolean operators

- *And, Or* are commutative
- Short-circuiting is allowed
- Boolean operators are non-deterministic
  - surprise for programmers (lost satellites):
    
    \[
    \text{If } ((x \text{ castable as xs:integer}) \text{ and } 
    ((x \text{ cast as xs:integer}) \text{ eq } 2)) \ldots
    \]

  - Is SQL deterministic? How can that happen in SQL?
- 2 value logic (unlike SQL!)
  - () is converted into fn:false() before use
- Conventional distributivity rules hold
LET clause folding

• Traditional rewriting

```plaintext
let $x := 3                        3+2
return $x +2
```

• Not so easy!

```plaintext
let $x := <a/>                (<a/>, <a/> )
return ($x, $x )
```

`NO. Side effects. (Node identity)`

```plaintext
declare namespace ns="uri1"
let $x := <ns:a/>         ($x, $x )
return <b xmlns:ns="uri2">{$x}</b>
```

`NO. Context sensitive namespace processing.`

```plaintext
declare namespace ns ="uri1"
<b xmlns:ns="uri2">{<ns:a/>}</b>
```
LET clause folding (cont.)

• Impact of unordered{..} /* context sensitive*/

\[ \text{let } x := (y/a/b)[1] \]
\[ \text{return unordered } \{ x/c \} \]

the c’s of a specific b parent
(in no particular order)

\[ \text{not equivalent to} \]
\[ \text{unordered } \{(y/a/b)[1]/c \} \]

the c’s of “some” b
(in no particular order)
LET Clause Folding

- **Sufficient conditions for correct rewriting of … into …**

  (: before LET :)  (: before LET :)
  let $x := \text{expr1}$  (: after LET :)
  (: after LET :)  return \text{expr2’}
  return \text{expr2}

  where \text{expr2’} is \text{expr2}
  with substitution \{x => \text{expr1}\}

- **Expr1 does not generate new nodes**
- **OR $x$ is used**
  a) only once and  
b) not part of a loop and  
c) not input to a recursive function
- **Dataflow analysis required**
Let Clause Unfolding

• Traditional rewriting

\[
\text{let } y := (1 \text{ to } 10) \\
\text{return } (y+2) + x
\]

\[
\text{for } x \text{ in } (1 \text{ to } 10) \\
\text{return } y + x
\]

• Not so easy!

– Same problems as before: side-effects, NS handling, unordered
– Additional problem: \textit{error handling}

\[
\text{let } y := (\text{input div 0})
\]

\[
\text{for } x \text{ in } (1 \text{ to } 10) \\
\text{return if } (x \text{ lt 1}) \\
\text{then } (\text{input div 0}) \\
\text{else } x
\]

\[
\text{let } y := (\text{input div 0})
\]

\[
\text{for } x \text{ in } (1 \text{ to } 10) \\
\text{return if } (x \text{ lt 1}) \\
\text{then } y \\
\text{else } x
\]

Guaranteed only if runtime implements consistently lazy evaluation. Otherwise dataflow analysis \textit{and} error analysis required.
Function inlining

• Traditional FP rewriting technique
  
  define function \( f(x \text{ as } xs:\text{integer}) \text{ as } xs:\text{integer} \)
  \( \{x+1\} \)
  \( f(2) \)

• Not always!
  - Same problems as for LET (NS handling, side-effects, unordered
  - Additional problems: *implicit operations (atomization, casts)*
    
    define function \( f(x \text{ as } xs:\text{double}) \text{ as } xs:\text{boolean} \)
    \( \{x \text{ instance of } xs:\text{double}\} \)
    \( f(2) \)

    (2 instance of xs:double) NO

• Make sure this rewriting is done *after* normalization
Constant folding

• Place constant values where the result can already be determined at compile time

```
for $x$ in (1 to 10) where $x$ eq 3
  return $x+1$
```

```
for $x$ in (1 to 10) where $x$ eq 3
  return (3+1)
```
Constant folding - counterexamples

for $x$ in $\text{input}/a$
where $x \text{ eq } 3$
return $\langle b \rangle x \langle /b \rangle$

for $x$ in $\text{input}/a$
where $x \text{ eq } 3$
return $\langle b \rangle 3 \langle /b \rangle$

for $x$ in (1.0, 2.0, 3.0)
where $x \text{ eq } 1$
return ($x \text{ instance of } \text{xs:integer}$)

for $x$ in (1.0, 2.0, 3.0)
where $x \text{ eq } 1$
return (1 instance of $\text{xs:integer}$)
Common Sub-expressions

• Preliminary questions
  – *Same* expression ?
  – *Same* context ?
  – *Error* “equivalence” ?
  – Create the same *new nodes*?

```plaintext
for $x$ in $input/a/b$
where $x/c$ lt 3
return if ($x/c$ lt 2)
  then if ($x/c$ eq 1)
      then (1 idiv 0)
      else $x/c$+1
  else if($x/c$ eq 0)
      then (1 idiv 0)
      else $x/c$+2
else
  then if ($x/c$ eq 0)
      then (1 idiv 0)
      else $x/c$+2
else
  then $y$ := (1 idiv 0)
  for $x$ in $input/a/b$
  where $x/c$ lt 3
  return if($x/c$ lt 2)
    then if ($x/c$ eq 1)
        then $y$
        else $x/c$+1
    else if($x/c$ eq 0)
        then $y$
        else $x/c$+2
else
  then $y$
  else $x/c$+2
```
FLWR unnesting

- **Traditional database technique**
  
  ```
  for $x$ in (for $y$ in $input/a/b$
    where $y/c$ eq 3
    return $y/d$)
  where $x/e$ eq 4
  return $x$
  
  for $y$ in $input/a/b,$
  $x$ in $y/d$
  where ($x/e$ eq 4) and ($y/c$ eq 3)
  return $x$
  ```

- **Problem simpler than in OQL/ODMG**
  - No nested collections in XML

- **Order-by more complicated**
Another traditional database technique

for $x$ in $input/a/b$
where $x/c$ eq 3
return (for $y$ in $x/d$
  ($y/c$ eq 3)
  where $x/e$ eq 4
  return $y$)

for $x$ in $input/a/b$, $y$ in $x/d$
where ($x/e$ eq 4) and
return $y$
Type-based rewritings

- Increase the advantages of lazy evaluation
  - $\text{input}/a/b/c \rightarrow (((\text{input}/a)[1]/b[1])/c)[1]

- Eliminate the need for expensive operations (e.g., sort)
  - $\text{input}//a/b \rightarrow \text{input}/c/d/a/b$

- Static dispatch for overloaded built-in functions
  - e.g. min, max, avg, arithmetics, comparisons
  - Maximizes the use of indexes

- Elimination of no-operations
  - e.g. casts, atomization, effective boolean value

- Choice of various run-time implementations for certain logical operations
Dealing with backwards navigation

• Replace backwards navigation with forward axis

for $x$ in $\text{input/a/b}$
return <c>{$x/.., $x/d}</c>

for $y$ in $\text{input/a,}$
$x$ in $y/b$
return <c>{$y, $x/d}</c>

for $x$ in $\text{input/a/b}$
return <c>{$x//e/..}</c>

• Enables streaming
More compiler support for efficient execution

- Streaming vs. data materialization
- Node identifiers handling
- Document order handling
- Scheduling for parallel execution
Detour/Background: Query Evaluation

- **Hard to discuss special algorithms**
  - Strongly depend on algebra
  - Strongly depends on the data storage, APIs and indexing

- **Main issues:**
  1. Streaming or materializing evaluations
  2. Lazy evaluation or not
Lazy Evaluation

• **Compute expressions on demand**
  – compute results only if they are needed
  – requires a **pull-based** interface (e.g. iterators)

• **Example:**
  
  ```
  declare function endlessOnes() as integer*
  { (1, endlessOnes()) };
  some $x$ in endlessOnes() satisfies $x$ eq 1
  ```

• **The result of this program should be:** true
Lazy Evaluation

• Lazy Evaluation also good for SQL
  – e.g., nested queries

• Particularly important for XQuery
  – existential, universal quantification (often implicit)
  – top N, positional predicates
  – recursive functions (non terminating functions)
  – if then else expressions
  – match
  – correctness of rewritings, …
Stream-based Processing

- Pipe input data through query operators
  - produce results before input is fully read
  - produce results incrementally
  - minimize the amount of memory required for the processing

- Stream-based processing
  - online query processing, continuous queries
  - particularly important for XML message routing

- Traditional in the database/SQL community
Stream based processing issues

• Streaming burning questions:
  – push or pull?
  – Granularity of streaming? Byte, event, item?
  – Streaming with flexible granularity?

• Pure streaming?
  – Processing XQuery needs some data materialization
  – Compiler support to detect and minimize data materialization

• Notes:
  – Streaming + Lazy Evaluation possible
  – Partial Streaming possible/necessary
When should we materialize?

• Pipeline breakders operators (e.g. sort)
• Other conditions:
  – Whenever a variable is used multiple times
  – Whenever a variable is used as part of a loop
  – Whenever the content of a variable is given as input to a recursive function
  – In case of backwards navigation
• Those are the ONLY cases
• materialization can be partial and lazy
• Compiler can detect via dataflow analysis
How to minimize the use of node IDs?

• Node identifiers are required by the XQuery Data model but onerous (time, space)

• Solution:
  1. Decouple the node construction operation from the node id generation operation
  2. Generate node ids *only if really needed*

    • Only if the query contains (after optimization) operators that need node identifiers (e.g. sort by doc order, is, parent, <<) OR node identifiers are required for the result (e.g., XQuery Update Facility)

• Compiler support: dataflow analysis
How can we deal with Xpath?

- Sorting by document order and duplicate elimination required by the XQuery semantics but very expensive
- Semantic conditions
  - $\text{document} / a / b / c$
    - Guaranteed to return results in doc order and not to have duplicates
  - $\text{document} / a // b$
    - Guaranteed to return results in doc order and not to contain duplicates
  - $\text{document} // a / b$
    - NOT guaranteed to return results in doc order but guaranteed not to contain duplicates
  - $\text{document} // a // b$                 $\text{document} / a / .. / b$
    - Nothing can be said in general
Parallel execution

\[
\text{ns1:WS1($input$)} + \text{ns2:WS2($input$)}
\]

\[
\text{for } x \text{ in } (1 \text{ to } 10) \\
\text{return ns:WS($i$)}
\]

- Certain expressions can be executed in parallel
  - Scheduling based on data dependency
- Parallelism \textit{within} a single expression
  - Horizontal and vertical partitioning
- \textbf{errors and parallellism is tricky}
  - in particular for side-effecting expressions
XQuery expression analysis (1)

• How many times expression uses a variable?
  – potential for common subexpression factorization
• Does expression use variable in loop?
  – limits unfolding
• Is an expression a map on a certain variable?
  – great for parallelization
• Does expression return results in doc order?
  – eliminate unnecessary sorts
• Does expression return distinct nodes?
  – eliminate unnecessary duplicate-elims
XQuery expression analysis (2)

- Is an expression a “function”?
- Can the result of an expression contain newly created nodes?
- Is the evaluation of an expression context-sensitive?
- Can an expression raise user errors?
- Is a sub expression of an expression guaranteed to be executed?
- Etc.
Compiling XQuery vs. XSLT

• Empiric assertion: it depends on the entropy level in the data (see M. Champion xml-dev):
  – XSLT easier to use if the shape of the data is totally unknown (entropy *high*)
  – XQuery easier to use if the shape of the data is known (entropy *low*)

• Dataflow analysis possible in XQuery, much harder in XSLT
  – Static typing, error detection, lots of optimizations

• Conclusion: less entropy means more potential for optimization, unsurprisingly.