

A Quilt, not a Camel

Don Chamberlin

Jonathan Robie

Daniela Florescu

May 19, 2000

The Web Changes Everything

- All kinds of information can be made available everywhere, all the time
- XML is the leading candidate for a universal language for information interchange
- To realize its potential, XML needs a query language of comparable flexibility
- Several XML query languages have been proposed and/or implemented
 - XPath, XQL, XML-QL, Lorel, YATL
 - Most are oriented toward a particular domain such as semi-structured documents or databases

Goals of the Quilt Proposal

- Leverage the most effective features of several existing and proposed query languages
- Design a small, clean, implementable language
- Cover the functionality required by all the XML Query use cases in a single language
- Write queries that fit on a slide
- Design a quilt, not a camel
- "Quilt" refers both to the origin of the language and to its intended use in knitting together heterogeneous data sources

Antecedents: XPath and XQL

- Closely-related languages for navigating in a hierarchy
- A *path expression* is a series of *steps*
- Each step moves along an *axis* (children, ancestors, attributes, etc.) and may apply a *predicate*
- XPath has an *abbreviated syntax*, adapted from XQL:
 - `/book[title = "War and Peace"]`
 - `/chapter[title = "War"]`
 - `//figure[contains(caption, "Guns")]`
- XQL has some additional operators: BEFORE, AFTER, ...

Antecedent: XML-QL

- Proposed by Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, Dan Suciu
- WHERE-clause binds variables according to a pattern, CONSTRUCT-clause generates output document

WHERE

```
<part pno = $pno> $pname </> in "parts.xml",  
<supp sno = $sno> $sname </> in "supp.xml",  
<sp pno = $pno sno = $sno> </> in "sp.xml"
```

CONSTRUCT

```
<purchase>  
  <partname> $pname </>  
  <suppname> $sname </>  
</purchase>
```

Antecedents: SQL and OQL

- SQL and OQL are database query languages
- SQL derives a table from other tables by a stylized series of clauses: SELECT - FROM - WHERE
- OQL is a functional language
 - A query is an expression
 - Expressions can take several forms
 - Expressions can be nested and combined
 - SELECT-FROM-WHERE is one form of OQL expression

A First Look at Quilt

- "Find the description and average price of each red part that has at least 10 orders"

```
FOR $p IN document("parts.xml")
    //part[color = "Red"]
LET $o := document("orders.xml")
    //order[partno = $p/partno]
WHERE count($o) >= 10
RETURN
    <important_red_part>
        $p/description,
        <avg_price> avg($o/price) </avg_price>
    </important_red_part>
```

Quilt Expressions

- Like OQL, Quilt is a functional language (a query is an expression, and expressions can be composed.)
- Some types of Quilt expressions:
 - A *path expression* (using abbreviated XPath syntax):
`document("bids.xml")//bid[itemno="47"]/bid_amount`
 - An expression using operators and functions:
`($x + $y) * foo($z)`
 - An *element constructor*:
`<bid>
 <userid> $u </userid> ,
 <bid_amount> $a </userid>
</bid>`
 - A *"FLWR" expression*

A FLWR Expression

- A FLWR expression binds some variables, applies a predicate, and constructs a new result.

FOR ... LET ... WHERE ... RETURN

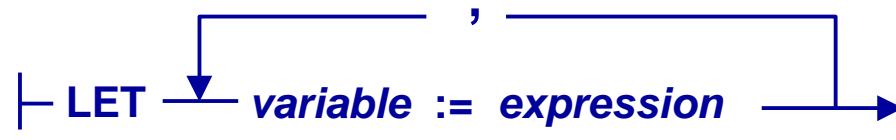


FOR Clause



- FOR is used for iterating over one or more collections
- Each *expression* evaluates to a collection of nodes
- The FOR clause produces many binding-tuples from the Cartesian product of these collections
- In each tuple, the value of each variable is one node and its descendants.
- The order of the tuples preserves document order unless some *expression* contains a non-order-preserving function such as `distinct()`.

LET Clause



- LET is used for binding variables (without iteration)
- A LET clause produces one binding for each variable (therefore it does not affect the number of binding-tuples)
- The variable is bound to the value of *expression*, which may contain many nodes.
- Document order is preserved among the nodes in each bound collection, unless *expression* contains a non-order-preserving function such as `distinct()`.

WHERE Clause

|—— WHERE *boolean-expression* ——>|

- Applies a predicate to the tuples of bound variables
- Retains only tuples that satisfy the predicate
- Preserves order of tuples, if any
- May contain AND, OR, NOT
- Applies scalar conditions to scalar variables:

`$color = "Red"`

- Applies set conditions to variables bound to sets:

`avg($emp/salary) > 10000`

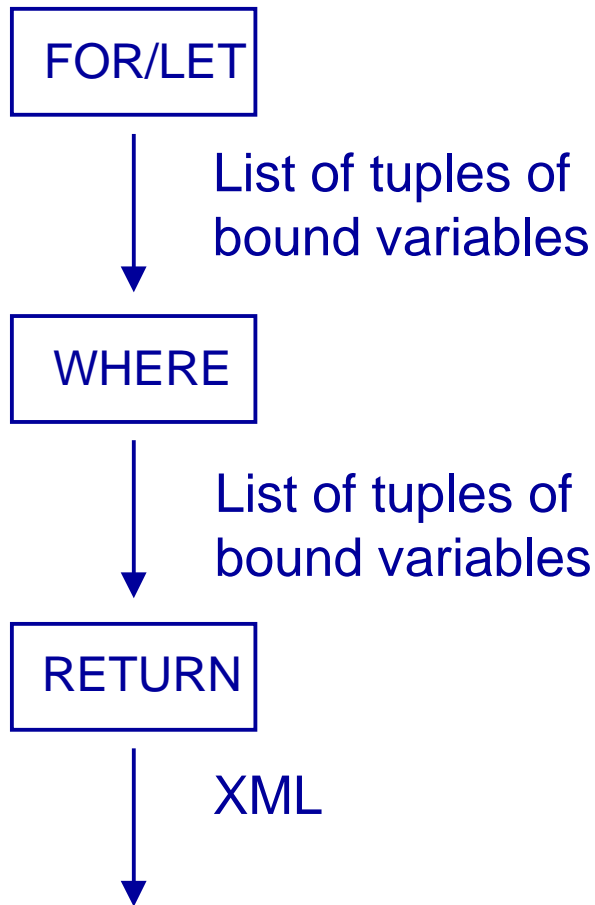
RETURN Clause

|—— RETURN *expression* ——>|

- Constructs the result of the FLWR expression
- Executed once for each tuple of bound variables
- Preserves order of tuples, if any, ...
- OR, can impose a new order using a SORTBY clause
- Often uses an element constructor:

```
<item>  
  $item/itemno,  
  <avg_bid> avg($b/bid_amount) </avg_bid>  
</item> SORTBY itemno
```

Summary of FLWR Data Flow



(\$x = value, \$y = value, \$z = value),
(\$x = value, \$y = value, \$z = value),
(\$x = value, \$y = value, \$z = value)

XML = ordered forest of nodes

An Example Document: bib.xml

- bib.xml has the following structure:

```
<bib>
  <book>
    <title>      ... </title>
    <author>    ... </author>
    . . .
    <publisher> ... </publisher>
    <year>      ... </year>
    <price>     ... </price>
  </book>
  . . .
</bib>
```

Simple XQuery queries

- "Find all the books published in 1998 by Penguin"

```
FOR $b IN document("bib.xml")//book
WHERE $b/year = "1998"
AND $b/publisher = "Penguin"
RETURN $b SORTBY(author, title)
```

- "Find titles of books that have no authors"

```
<orphan_books>
  FOR $b IN document("bib.xml")//book
  WHERE empty($b/author)
  RETURN $b/title SORTBY(.)
</orphan_books>
```

A nested query

- "Invert the hierarchy from publishers inside books to books inside publishers"

```
FOR $p IN distinct(//publisher)
```

```
RETURN
```

```
  <publisher>
```

```
    <name> $p/text() </name> ,
```

```
    FOR $b IN //book[publisher = $p]
```

```
    RETURN
```

```
      <book>
```

```
        $b/title ,
```

```
        $b/price
```

```
      </book> SORTBY(price DESCENDING)
```

```
    </publisher> SORTBY(name)
```

Operators based on global ordering

$$\mathit{expr1} \left\{ \begin{array}{c} \text{BEFORE} \\ \text{AFTER} \end{array} \right\} \mathit{expr2}$$

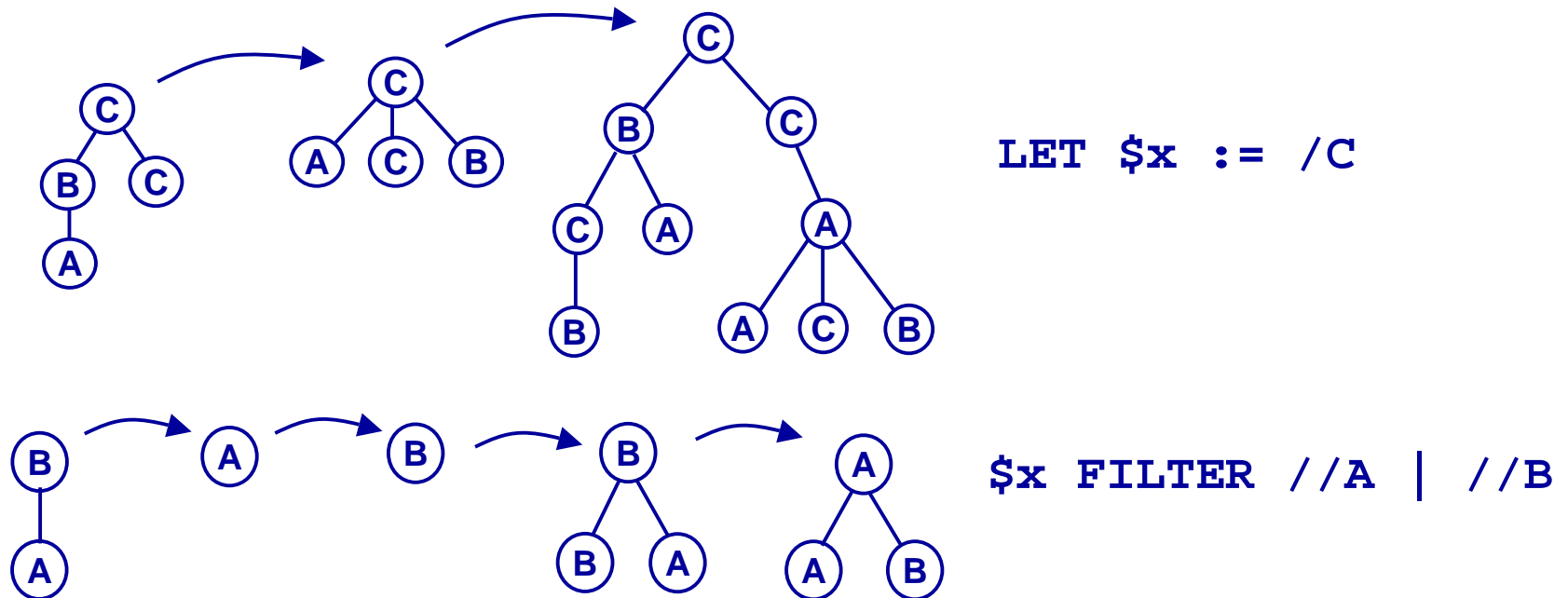
- Returns nodes in *expr1* that are before (after) some node in *expr2*
- "Find procedures where no anesthesia occurs before the first incision."

```
FOR $proc IN //section[title = "Procedure"]
WHERE empty( $proc//anesthesia
              BEFORE ($proc//incision)[1] )
RETURN $proc
```

The FILTER Operator

expression FILTER *path-expression*

- Returns the result of the first expression, "filtered" by the second expression
- Result is an "ordered forest" that preserves sequence and hierarchy.



Projection (Filtering a document)

- "Generate a table of contents containing nested sections and their titles"

```
<toc>
  document ( "cookbook.xml" )
    FILTER //section
      | //section/title
      | //section/title/text()
</toc>
```

Conditional Expressions

```
IF expr1 THEN expr2 ELSE expr3
```

- "Make a list of holdings, ordered by title. For journals, include the editor; otherwise include the author."

```
FOR $h IN //holding
RETURN
  <holding>
    $h/title,
    IF $h/@type = "Journal"
      THEN $h/editor
      ELSE $h/author
  </holding> SORTBY(title)
```

Functions

- A query can define its own local functions
- If f is a scalar function, $f(S)$ is defined as $\{ f(s): s \text{ in } S \}$
- Functions can be recursive
- "Compute the maximum depth of nested parts in the document named partlist.xml"

```
FUNCTION depth($e)
  { IF empty($e/*) THEN 1
    ELSE max(depth($e/*)) + 1
  }
depth(document("partlist.xml") FILTER //part)
```

Quantified Expressions

$\left\{ \begin{array}{c} \text{SOME} \\ \text{EVERY} \end{array} \right\} \text{ var IN expr SATISFIES predicate}$

- Quantified expressions are a form of predicate (return Boolean)
- "Find titles of books in which both sailing and windsurfing are mentioned in the same paragraph"

```
FOR $b IN //book
WHERE SOME $p IN $b//para SATISFIES
  contains($p, "Sailing")
  AND contains($p, "Windsurfing")
RETURN $b/title
```

Variable Bindings

LET *variable* := *expression* **EVAL** *expression*

- "For each book that is more expensive than average, list the title and the amount by which the book's price exceeds the average price"

```
LET $a := avg(//book//price)
EVAL
  FOR $b IN //book
  WHERE $b/price > $a
  RETURN
    <expensive_book>
      $b/title,
      <price_difference>
        $b/price - $a
      </price_difference>
    </expensive_book>
```

XML Views of Relational Data

- A relational database system could export data in XML format using these steps:
 - Use an SQL query to define the data to be exported (in tabular form)
 - Use a simple default mapping from each exported table to an XML tree
 - Use a Quilt query to compose the trees into an XML view with any desired structure
 - Quilt queries against the XML view can be composed with the Quilt query that defines the view

Querying Relational Data

- Tables can be represented by simple XML trees
 - Table = root
 - Each row becomes a nested element
 - Each data value becomes a further nested element

SUPPLIERS

SNO	SNAME
------------	--------------

```
<suppliers>
```

```
  <s_tuple>
```

```
    <sno>
```

```
    <sname>
```

PARTS

PNO	DESCRIP
------------	----------------

```
<parts>
```

```
  <p_tuple>
```

```
    <pno>
```

```
    <descrip>
```

CATALOG

SNO	PNO	PRICE
------------	------------	--------------

```
<catalog>
```

```
  <c_tuple>
```

```
    <sno>
```

```
    <pno>
```

```
    <price>
```

SQL vs. Quilt

"Find part numbers of gears, in numeric order"

- SQL:

```
SELECT pno
FROM parts AS p
WHERE descrip LIKE 'Gear'
ORDER BY pno;
```

- Quilt:

```
FOR $p IN document("parts.xml")//p_tuple
WHERE contains($p/descrip, "Gear")
RETURN $p/pno SORTBY(.)
```

GROUP BY and HAVING

"Find part no's and avg. prices for parts with at least 3 suppliers"

- SQL:

```
SELECT pno, avg(price) AS avg_price
FROM catalog AS c
GROUP BY pno HAVING count(*) >= 3
ORDER BY pno;
```

- Quilt:

```
FOR $p IN distinct(document("parts.xml")//pno)
LET $c := document("catalog.xml")
                //c_tuple[pno = $p]
WHERE count($c) >= 3
RETURN
    <well_supplied_part>
        $p,
        <avg_price> avg($c/price) </avgprice>
    </well_supplied_part> SORTBY(pno)
```

Inner Join

"Return a 'flat' list of supplier names and their part descriptions"

- Quilt:

```
FOR $c IN document("catalog.xml")//c_tuple,  
    $p IN document("parts.xml")  
        //p_tuple[pno = $c.pno],  
    $s IN document("suppliers.xml")  
        //s_tuple[sno = $c.sno]  
RETURN  
    <offering>  
        $s/sname,  
        $p/descrip  
    </offering> SORTBY(sname, descrip)
```

Outer Join

"List names of all suppliers in alphabetic order; within each supplier, list the descriptions of parts it supplies (if any)"

- Quilt:

```
FOR $s IN document("suppliers.xml")//s_tuple
RETURN
  <supplier>
    $s/sname,
    FOR $c IN document("catalog.xml")
      //c_tuple[sno = $s/sno],
      $p IN document("parts.xml")
        //p_tuple[pno = $c/pno]
    RETURN $p/descrip SORTBY(.)
  </supplier> SORTBY(sname)
```

Quilt grammar (1)

- Queries and Functions:

```
query ::= function_defn* expr
function_defn ::=
    'FUNCTION' function_name
        '(' variable_list ')' '{' expr '}'
```

- Example of a function definition:

```
FUNCTION spouse_age($x)
{ $x/spouse/age }
```

- Functions:

- Core function library: contains, empty, distinct, avg, ...
- domain-dependent library: eg. area of a polygon
- local functions: eg. spouse_age(\$x)

Quilt grammar (2)

- Expressions:

```
expr ::= variable
      | constant
      | expr infix_operator expr
      | prefix_operator expr
      | function_name '(' expr_list? ')
      | '(' expr ')'
      | expr '[' expr ']'
      | 'IF' expr 'THEN' expr 'ELSE' expr
      | 'LET' variable ':=' expr 'EVAL' expr
```

- Infix operators:

```
+ - * div mod = < <= > >= != | AND OR NOT
UNION INTERSECT EXCEPT BEFORE AFTER
```

- Prefix operators: + - NOT

Quilt grammar (3)

- Expressions, continued:

```
expr ::= path_expression
      | element_constructor
      | FLWR_expression
```

```
element_constructor ::=
    start_tag expr_list? end_tag
```

```
start_tag ::= '<' tag_name attributes? '>'
```

```
attributes ::= ( attr_name '=' expr )+
            | 'ATTRIBUTES' expr
```

```
<end_tag> ::= '</' tag_name '>'
```

```
<tag_name> ::= QName | variable
```

```
<attr_name> ::= QName | variable
```

Quilt grammar (4)

- FLWR_Expressions:

```
FLWR_expression ::= for_clause  
                  ( for_clause | let_clause )*  
                  where_clause? return_clause  
for_clause ::=  
    'FOR' variable 'IN' expr  
    ( ',' variable 'IN' expr )*  
let_clause ::=  
    'LET' variable ':=' expr  
    ( ',' variable ':=' expr )*  
where_clause ::= 'WHERE' expr  
return_clause ::= 'RETURN' expr
```

Quilt grammar (5)

- Second-order expressions:

```
expr ::= expr 'FILTER' path_expression
      | quantifier variable 'IN' expr
        'SATISFIES' expr
      | expr 'SORTBY'
        '(' expr order? , ... ')'
```

```
quantifier ::= 'SOME' | 'EVERY'
```

```
order ::= 'ASCENDING' | 'DESCENDING'
```

Comments on the Grammar

- In general, correctness of a query is enforced by:
 - Syntactic rules (e.g. grammar)
 - Semantic rules (e.g. variable and function scope)
 - Type checking rules (e.g. the expression in the WHERE clause must be of type Boolean)
- The Quilt grammar is quite permissive
 - It deals with only the first of the above items
- The Quilt grammar is just a beginning. Still to come:
 - Core function library
 - Type checking rules
 - Formal semantic specification
 - Grammar may change as language evolves

Summary

- XML is very versatile markup language
- Quilt is a query language designed to be as versatile as XML
- Quilt draws features from several other languages
- Quilt can pull together data from heterogeneous sources
- Quilt can help XML to realize its potential as a universal language for data interchange
- For more details, see www.almaden.ibm.com/cs/people/chamberlin/quilt.html

QUILT