1 Background: XML Data and XML Queries

In this section, we define a simple model for XML data that will suffice for our illustrative purposes, and define an XQuery-like query language for this model that consists of: (1) a built-in collection of *XML functions*, denoted XFn, and (2) XQuery's FLWR expressions. A subset of XFn, the so-called *constructor* functions, correspond to XML patterns in XQuery, and (although the details are beyond the scope of this paper) arbitrary XPath queries can be straightforwardly mapped to a combination of XFn and FLWR expressions in our language.

1.1 XML Data

As is common with most papers on XML, we interpret an XML document or a document collection as an ordered forest of rooted node-labeled ordered trees.

Definition 1.1 (XML Forests and Trees) *Let* String *be a set of strings. We define the set* $\mathsf{XMLForest}$ *of XML forests inductively by*

$$\mathsf{XMLForest} \quad = \quad \epsilon \;\mid\; s \;\mid\; [\mathsf{XNode}(s,x)] \;\mid\; x@y,$$

where s *is in* String*,* x *and* y *are in* $\mathsf{XMLForest}$*,* $[\,]$ *denotes an empty forest*[1]*,* $[\mathsf{XNode}(s,x)]$ *denotes a forest containing a single tree with a root labeled* s *and an ordered forest* x*, and* $x@y$ *denotes a concatenation of two ordered forests.*

Queries are built out of *basic functions* of XML forests; these can be combined by function composition and in FLWR expressions.

1.2 Basic Functions

These operations come in three flavors:

Constructors These are functions that create new XML forests; there are four cases:

1. the *empty forest* constructor $\mathsf{0}$,
2. for any $s \in\mathsf{String}$ the *leaf* constructor $'\mathsf{s}'$,
3. for any $x \in\mathsf{XMLForest}$ and $s \in\mathsf{String}$ the *element* constructor $<\!\mathsf{s}\!>x<\!/\mathsf{s}\!>$, and
4. for any $x, y \in\mathsf{XMLForest}$ the *concatenation* constructor xy.

Horizontal Operations These are functions that pick trees from a forest.

1. for $x \in\mathsf{XMLForest}$ the *first* function, $\mathsf{first}(x)$, returns the first tree from x (for nonempty x, empty otherwise).
2. for $x \in\mathsf{XMLForest}$ the *last* function, $\mathsf{last}(x)$, returns the last tree from x (for nonempty x, empty otherwise).

[1] We employ common list notation to represent ordered forests.

$$Q ::= x \mid \mathsf{XFn}(e_1,\ldots,e_k) \mid \mathsf{where}\ \varphi\ \mathsf{return}\ e \mid \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 \mid \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{do}\ e_2$$

$$\begin{array}{rcl}
[\![\, x \,]\!]\, E & = & E(x) \\
[\![\, \mathsf{XFn}(e_1,\ldots,e_k) \,]\!]\, E & = & [\![\, \mathsf{XFn} \,]\!]([\![\, e_1 \,]\!]\, E,\ldots,[\![\, e_k \,]\!]\, E) \\
[\![\, \mathsf{where}\ \varphi\ \mathsf{return}\ e \,]\!]\, E & = & \text{if } [\![\, \varphi \,]\!]\, E \text{ then } [\![\, e \,]\!]\, E \text{ else } [\,] \\
[\![\, \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 \,]\!]\, E & = & [\![\, e_2 \,]\!](E[x := ([\![\, Q_1 \,]\!]\, E)]) \\
[\![\, \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{do}\ e_2 \,]\!]\, E & = & \mathsf{let}\ [v_1,\ldots,v_k] = [\![\, e_1 \,]\!]\, E \\
 & & \quad \mathsf{in}\ [\![\, e_2 \,]\!](E[x := v_1])@\ldots@[\![\, e_2 \,]\!](E[x := v_k])
\end{array}$$

Figure 1: Syntax and Semantics of FLWR-like Expressions.

Other (optional) horizontal operations include select_s (pick all trees with root label s), distinct (pick the trees with unique labels in document order), sort (sorts the trees in increasing order of their root labels, reverse reverses the horizontal order of the trees in the input, etc.

Vertical Operations

1. for $x \in$XMLForest the *children* function, $\mathsf{child}(x)$, returns a forest of all children of x ordered first by the position of the original tree and then by the order of the children in the original tree (i.e., in "document order"),
2. for $x \in$XMLForest the *subtrees* function, $\mathsf{subtrees}(x)$, returns a forest of all subtrees of x ordered first by the position of the original tree and then by the DFS order of their roots in the original tree (i.e., in "document order").

Again, additional functions are $\mathsf{rev} - \mathsf{subtrees}$ (subtrees in reverse DFS order), $\mathsf{bfs} - \mathsf{subtrees}$ (in BFS order), etc.

There are, of course, many more functions definable on XML forests.

1.3 FLWR-like Queries

The functions on XML forests defined in Section 1.2 are used to define basic expressions; that is, terms built out of *variables* and function symbols that correspond to the names of the built-in functions XFn from Section 1.2. The meaning of such an XFn expression is defined with respect to an *environment* E that consists of a set of variable-XML forest pairs. The pairs define the meaning for variables that occur in the XFn expression. The meaning of the whole term is then defined inductively on its structure.

The boolean conditions in the where clause can only be of the form $x == y$, $x == s$, and $\mathsf{empty}(x)$ for x, y variables and $s \in$String that becomes true if the label of the leftmost root of x is equal to the label of the leftmost root of y or to s, or if x represents the empty tree, respectively. *Deep equality on trees* is extra (and you need counting here).

2 Translation to SQL

2.1 Interval Encoding of XML

We now turn our attention to the main goal of this paper—to representing XML data in relational tables and XML queries as domain relational calculus queries over the relational encoding.

Definition 2.1 (An Interval Encoding) *Let* $x \in$XMLForest *be an XML forest. We define a ternary relation* $X \subseteq$ String $\times$ Nat $\times$ Nat *as follows:*

- XNode(s, y) *is a node in* x *if and only if* $(s, l, r) \in X$,
- XNode(s_1, y_1) *is a parent of* XNode(s_2, y_2) *in* x *if and only if there are tuples* $(s_1, l_1, r_1) \in X$, *and* $(s_2, l_2, r_2) \in X$ *such that* $l_1 < l_2$ *and* $r_1 > r_2$, *and*
- XNode(s_1, y_1) *is a left sibling of* XNode(s_2, y_2) *in* x *implies there are two (unique) tuples* $(s_1, l_1, r_1) \in X$ *and* $(s_2, l_2, r_2) \in X$ *such that* $r_1 < l_2$.

We say that R *represents (is a* XML table *for)* x.

2.2 Translation of Builtin Functions

XML Forest Constructors. Let s be a string, $x, y \in$XMLForest be an XML forests, and X and Y XML tables representing x and y, respectively. Then

- $\{\}$ represents the empty forest ϵ,
- $\{(s, 0, 1)\}$ represents the leaf s,
- $\{(s, 0, r_m + 2)\} \cup \{(s, l + 1, r + 1) : X(s, l, r)\}$ represents [XNode(s, x)], and
- $X \cup \{(s, l + r_m + 1, r + r_m + 1) : Y(s, l, r)\}$ represents $x@y$.

where r_m is the maximal value in $\{r : X(s, l, r)\}$.

Translations of Horizontal and Vertical Operations.

Let $x \in$XMLForest be an XML forest and let X represents x. Then

$$\{(s, l, r) : X(s, l, r) \wedge \exists s', l'.r'.X(s', l', r') \wedge r \leq r' \wedge \neg\exists s'', l'', r''.X(s'', l'', r'') \wedge l'' < l'\}$$

represents first(x). Similarly, last, distinct, etc. can be defined by first-order queries.

2.3 Translation of FLWR Expressions

Representation of *environments*:

Definition 2.2 *Let* $e_1, \ldots, e_n$ *be a* sequence *of environments over the variables* $x_1, \ldots, x_l$. *A* relational representation *of this sequence is an instance of the schema* $\rho = (E, X_1, \ldots, X_l)$ *where* E *is an unary relation with an integer attribute and* X_i *XML tables. We say that an instance of* ρ represents $e_1, \ldots, e_k$ *if*

- $E = \{i_1, \ldots, i_n\}$,
- $\{(s, l, r) : X_j(s, l, r) \wedge w_j \cdot i_k \leq l < r \leq w_j(i_n + 1)\}$ *represents the XML forest assigned to the variable* x_j *in* e_k.

where w_j *is the maximal* width *of any forest assigned to* x_j.

Observations:

- the initial environment is $E = \{1\}$ and several variables denoting the *input* documents,
- the w_j values can be determined from the widths of the input documents and the functions applied on these (and thus can be "precomputed" and passed to the query as parameters),
- the implementations of the builtin operations can be extended to handle the environment-based input rather than single XML forest(s).

3 Assignment Goals

Write

- a parser for the XML language above (please make sure it is easy to add additional *builtin* functions),
- a translator of the parse (abstract) tree to a corresponding relational expression (plan for the evaluator from A2).

In addition, you should make a viable attempt on one of the following enhancements:

1. algebraic simplifier of the queries (you may choose if the optimization is done on the XML or the relational side),
2. a cost-based optimizer for the generated plans (here you need to think about appropriate cost model, statistics kept about XML documents, etc.),
3. extension(s) to the query evaluation engine to support the generated queries more efficiently.