XML Query Processing and Optimization

Ning Zhang

School of Computer Science
University of Waterloo
nzhang@uwaterloo.ca

Abstract. In this paper, I summarize my research on optimizing XML queries. This work has two components: the first component is the definition of a logical algebra and logical optimization techniques. This algebra can be translated into different physical algebras such as native or extended-relational algebras. The second component is the design of physical storage structures and physical optimization techniques.

1 Introduction

XML has grown from a markup language for special-purpose documents to a standard for the interchange of heterogeneous data over the Web, a common language for distributed computation, and a universal data format to provide users with different views of data by transforming between different formats. All of these increase the volume of data encoded in XML, consequently increasing the need for database management support for XML documents. An essential concern is how to store and query potentially huge amounts of XML data efficiently.

There are generally two approaches to providing database support for XML data management: extended-relational approach and native XML approach. In the extended-relational approach, XML documents are transformed to relational data, and XQuery expressions are translated to SQL expressions. The space for XML query optimization is on the XQuery-to-SQL translation and the SQL expression itself. On the other hand, in the native XML approach, XML documents are managed without any relational database support. XML data are simply modeled as labeled, ordered, rooted trees. Any query on XML data is then translated into a series of operations on XML trees. The optimization is therefore on the translation and the operations themselves.

Both of these approaches have their own advantages and disadvantages. For example, the extended-relational approach is heavily dependent on the physical level representation (e.g., interval encoding [1]) of XML data. Existing translation schemes are either generally inefficient in processing tree structured data, or require significant amount of work on update. On the other hand, the native XML approach benefits from the fact that one can design new storage structures and define new operators to satisfy both query and update requirements. The downside of the native approach is its lack of maturity, especially in the optimization techniques.

In this paper, I propose a novel approach to compromise between the above two approaches by defining a logical algebra that captures the semantics of XQuery. The definition of the logical algebra should be general enough so that it can be implemented by the extended-relational or the native approach. A set of rewrite rules and a cost model should also be defined so that equivalent execution plans could be deduced and their costs could be estimated. Another major part of my thesis will be the development of physical optimization techniques that deal with storage structures and access methods. The aim of my Ph.D. research is, thus, to develop XML query processing and optimization techniques, in particular those techniques that are not only provably efficient in theory, but also empirically verifiably so in practice.

2 Significant Research Problems

In my Ph.D. research, I am concentrating on two research questions:

1. How to define a logical algebra and logical optimization rules that can be used by an extended-relational or native optimizer.
2. How to design physical storage structures and access methods in support of this logical algebra for efficient query processing.

XQuery is a functional programming language, thus it can be evaluated as any other functional programming language and enjoys their optimization techniques. However, since XQuery is designed as a database query language, one should bear in mind that the input data could be massive. To efficiently evaluate XQuery expressions against large amounts of data, we follow the traditional database query model:

- Define a logical algebra and rewrite rules for the query language.
- Design new physical storage structures if needed.
- For each logical operator, many physical operators (access methods) that implement the same functionalities could also be defined. These physical operators have different performances depending on the physical data distributions and the presence of auxiliary data (e.g., indexes).
- A cost model is also needed as a basis of choosing the optimal physical query plan.

In this paper, I shall present my preliminary study of the first three issues in the above list. The cost model is left as my future work.

3 Logical Algebra

3.1 Problem Definition

A logical algebra can be thought of as a specification that captures the semantics of XQuery expressions so that the primary concern of the physical operators is

how to satisfy the specification. The algebra must be *sound* (the translation of XQuery expressions into algebraic expressions must be correct) and *complete* (*any* XQuery expression can be translated into an algebraic expression). There are several problems that need to be considered when defining the algebra:

Sorts: The data model defined by W3C is a sequence of tree nodes. This implies that at least two sorts, List and TreeNode, are needed as indicated by the W3C's XQuery Formal Semantics [2]. However, we shall see that more sorts, such as NestedList and Tree, are necessary or convenient.

Operators: As usual, the algebra includes a set of operators, defined by their signatures and semantics. These should be *closed* under the sorts we define. I shall investigate well-known operations on these sorts, as well as defining new operations, and rewrite rules based on them.

Completeness vs. safety: The completeness property requires that any XQuery expression can be translated into an algebraic expression that returns the same result. However, since XQuery is Turing-complete [3], which introduces possibly non-terminating expressions, a complete algebra will be unsafe. To avoid this problem, I identify a subclass of XQuery that does not include recursive functions, and define a complete algebra for this subclass only. To expand to a larger subclass, one can define additional (maybe higher-order) operators based on this algebra.

Other issues such as type checking and error/exception handling are outside the scope of my thesis. The definition of rewrite rules is leaf as future work.

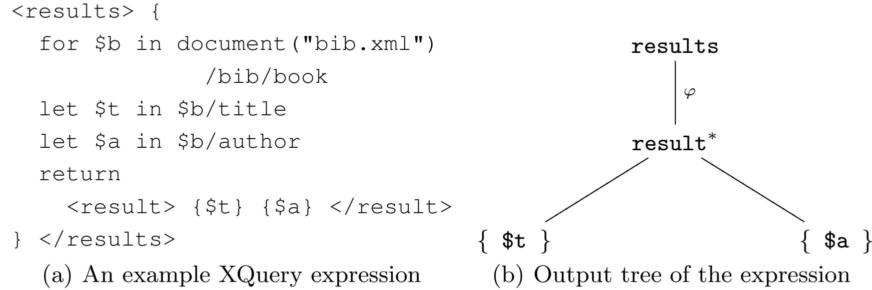
3.2 Preliminary Results

Sorts The input and output of an XQuery expression defined by XQuery data model is a *flat* sequence of tree nodes. Since there are no nested sequences, it seems that it is sufficient to define operators over two sorts: List (representing flat list) and TreeNode, together with other primitive sorts such as Integer, Boolean and String. However, flat list operations are not efficient in manipulating trees, which can be thought of as nested lists. For example, Fig. 1(a) is an XQuery expression taken from XQuery Use Cases. The output of the expression conforms to the tree schema shown in Fig. 1(b), where the root is labeled as `results`, which has zero or more children labeled `result`, under which there are two children (subtrees) whose values depend on the value of `$t` and `$a`. The leaf nodes labeled with { } are placeholders that can be replaced by the value (subtree) of the enclosed expression.

The edge label φ is an expression that evaluates the values for the variable(s) to replace the placeholder(s). It can be translated more formally into a list comprehension:

$$\varphi = \left[(\texttt{\$t}, \texttt{\$a}) \mid \texttt{\$b} \leftarrow \texttt{doc()/bib/book};\ \texttt{\$t} \leftarrow \texttt{\$b/title};\ \texttt{\$a} \leftarrow \texttt{\$b/author}\right]$$

The output of the comprehension is a list of 2-tuples (i.e., nested list), instead of a flat list of tree nodes. Relying on the W3C's data model over flat lists, the



(a) An example XQuery expression (b) Output tree of the expression

Fig. 1. An XQuery expression and its output schema

list comprehension can be evaluated either by iterating over each element in `$b`, during which `$t` and `$a` are evaluated accordingly, or by first creating a long list for each `$t` and `$a` in a batch, then joining them based on their structural relationship. Both of these strategies are not optimal since the first (pipelining) approach suffers from exponential runtime (to the size of the query) in the worst case [4], while the join-based approach needs an additional structural join [5].

On the other hand, generalizing the input and output as nested lists enables a single operator to implement the above list comprehension as a whole. A physical tree pattern matching operator (introduced later), for example, could implement the list comprehension with a single scan of the input data without the need for structural joins. This efficiency benefit necessitates a new sort NestedList that allows arbitrary level of nestings. It is straightforward to convert a nested list to a tree. However, there are no labels on the converted tree nodes. Therefore, it is necessary to define another sort Tree that represents labeled trees. In our algebra, an input XML document is of sort Tree; while some of the intermediate results could be of sort NestedList.

In addition, we also need sorts such as PatternGraph and SchemaTree, which are used to translate path expressions and constructor expressions, respectively. The sort PatternGraph captures the constraints specified by one or more path expressions. For example, the path expression `/a[b][c]` can be modeled as a pattern graph that has four vertices labeled with `root`, `a`, `b` and `c`, and three directed edges (`root`,`a`), (`a`,`b`), and (`a`,`c`). The three edges are all labeled with "/", which means that the "from" and "to" vertices are in parent-child relationship. The vertex `a` is marked as returning vertex, which means that the tree nodes matched with this vertex should be returned.

Definition 1 (PatternGraph). *A PatternGraph is a labeled, directed graph, which is denoted by a 5-tuple $\mathcal{P} = \langle \Sigma, \mathcal{V}, \mathcal{A}, \mathcal{R}, \mathcal{O} \rangle$, where Σ is a finite alphabet representing element (attribute, etc.) names, $\mathcal{V}$ and $\mathcal{A}$ are the sets of vertices and arcs (directed edges) in the graph, respectively, $\mathcal{R}$ is the set of binary relations between vertices, and $\mathcal{O} \subseteq \mathcal{V}$ is a set of output vertices indicating that all matching vertices should be output as results.*

For each vertex $v \in \mathcal{V}$:

- *v is labeled with $*$ or a set of characters in Σ, where $* \notin \Sigma$.*
- *v is associated with a list (may be empty) of $\langle \prec, l \rangle$ tuples, where $\prec$ is a comparison operator and l is a numerical or string literal.*

Each arc $(s,t) \in \mathcal{A}$ is labeled with a relation $r \in \mathcal{R}$, which indicates that (s,t) is in relation r.

The sort SchemaTree (cf. Fig. 1(b)) represents a labeled tree that is extracted from XQuery constructor expressions. This labeled tree structure specifies the schema of an out XML document. Each node in the schema tree is labeled with an element name, with an optional wildcard symbol * or ?. A node could be marked as a placeholder, in which case its node label should be an algebraic expression. All arcs in the tree represent parent-child relationship, so there is no label on the arcs.

Definition 2 (SchemaTree). *An SchemaTree is a labeled tree structure, which is denoted by a 4-tuple $\mathcal{O} = \langle \Sigma, \mathcal{N}, \mathcal{A}, \mathcal{E} \rangle$, where Σ is a set of finite alphabet representing element/attribute names, $\mathcal{N}$ is a set of tree nodes, $\mathcal{A}$ is a set of tree arcs, and $\mathcal{E}$ is a set of (XQuery/algebraic) expressions.*

Each leaf-node in $\mathcal{N}$ is labeled with a character in Σ (in which case it is an empty element/attribute) or an expression in $\mathcal{E}$. For each non-leaf node, it is labeled with a character in Σ (in which case it is called a constructor-node*) or a boolean-valued expression in $\mathcal{E}$ (in which case it is called an* if-node*).*

Each arc in $\mathcal{A}$ may (or may not) be labeled with an expression in $\mathcal{E}$.

In addition to path expression and constructor expression, FLWOR expression is another type of major expression. The uniqueness of FLWOR expression is that it is the only kind of expression that can introduce new variables. Since variables can be referenced by other expressions in some scope, it is necessary to define a scope-wise sort Env to keep track of all the defined variables and their bounded values constitute. The sort Env can be thought of as a NestedList (or unlabeled tree) with additional variable bindings to each level in the nesting.

Definition 3 (Env). *An* environment *(Env) is a layered, balanced tree structure, which is denoted by a 3-tuple $\mathcal{E} = \langle \mathcal{N}, \mathcal{A}, \mathcal{V} \rangle$, where $\mathcal{N}$ is a set of nodes, $\mathcal{A}$ is a set of arcs, and $\mathcal{V}$ is a set of variables. All tree nodes at the same level from the root forms a* layer*. Each layer is either associated with a variable in $\mathcal{N}$, or a boolean formula. All nodes in the same layer have the same cardinality relationship, i.e., the parent-child relationship between layers x and $x+1$ is either one-to-one or one-to-many, but not mixed. A path from the root to a leaf is called a* total variable bindings*, which include the bindings for all variables and boolean formula introduced in the where-clause.*

Example 1. The following FLWOR expression:

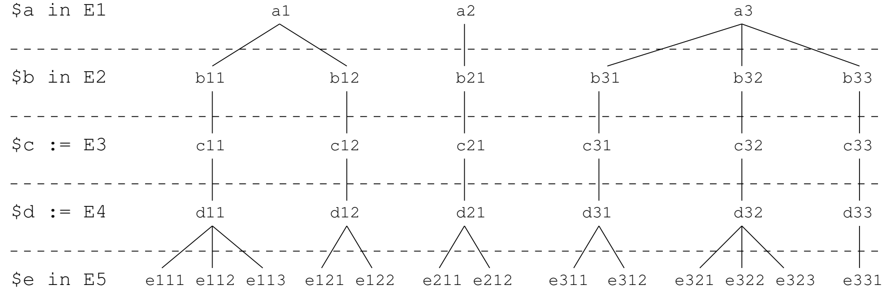


Fig. 2. An example environment by the schema `($a,($b,$c,$d,($e)))`

```
for $a in E1, $b in E2
  let $c := E3, $d := E4
    for $e in E5
      return E6
```

generates a nested list `($a,($b,$c,$d,($e)))`. Variables in this nested list from left to right are nested in different levels (as indicated in Fig. 2). Whether there is a opening parenthesis "(" between two variables determines how the tree is constructed. If there is a "(", every list item in the value bounded to the second variable is a child of the first variable binding. This corresponds to a for-clause style of binding values. If there is no "(", the list value of the second variable as a whole is a child of the first variable binding. A possible environment instantiated from this nested list is shown in Fig. 2.

The environment forms a forest, in which the number of trees is the number of values instantiating `$a`. The list of nodes at each level are generated by the expression at the left-hand-side, given the input nodes at its top levels: the first-level nodes are generated by the expression `E1`, and the second-level nodes are generated by the expression `E2` given the input of the nodes at the first level, and so on. This environment actually specifies 13 possible value assignments (each corresponding to a path from a root the a leaf) to the five variables. Therefore, to obtain the final result, the expression `E6` is evaluated once for each of these paths, and their results are concatenated together.

Operators The XQuery data model has already defined functions and operations for each sort (data type). We can use these as the operators over the predefined sorts in our algebra. In addition, we need to define operators for the newly introduced sorts. In relational algebra, some operators are schema-related (π and ρ, etc.), and some operators are value-related (σ, $\bowtie$, sorting, etc.). In our XML algebra, we also define operators in these two categories, but some operators may examine both schema and value information (cf. Table 1).

The operators listed in the structure-based and value-based have extended but very similar semantics to the relational operators. For example, σ^s selects

Table 1. Operators

Category	Operator	Signature	Short Description
structure-based	σ^s	List → List	selection based on tag names
	$\bowtie^s$	List × List → List	structural join
	π^s	List → NestedList	tree navigation along axis s
value-based	σ^v	List → List	selection based on values
	$\bowtie^v$	List × List → List	value-based join
hybrid	τ	List × PatternGraph → NestedList	tree pattern matching
	γ	NestedList × SchemaTree → Tree	tree construction operator

tree nodes whose tag names are "s", and $\bowtie^s$ join two lists of nodes if their structural relationships are "s".

The two hybrid operators involve both types of information and thus have more complex semantics. The tree pattern matching (TPM) operator (τ) takes an XML tree and a pattern graph as input, and produces a nested list as output. The semantics of τ operator is to find all nodes in the input tree that satisfy all the constraints specified by the pattern graph. The resulting nodes are output as a nested list according to their structural relationships in the input tree (i.e., two nodes are immediately nested in the output nested list iff they are in immediate ancestor-descendant relationship in the input tree). This operator is particularly interesting and I shall introduce this physical optimization technique in Section 4.

The construction operator (γ) takes a NestedList and SchemaTree as input, and produce a labeled Tree (which represent an XML document). Among the parameters, the NestedList is the intermediate results (tree without node labels) obtained by other operators, and the SchemaTree is the schema information that indicates how the intermediate results should be labeled.

These two operators, τ and γ, reside on the bottom and top of the execution plan, respectively. The τ operators process the input XML documents and produce intermediate results (nested lists). The γ operator takes the intermediate results together with the output schema, and produces the resulting XML document. The other operators reside in the middle in the execution plan, and serve as transformation tools on the intermediate results.

4 Physical Storage and Operators

4.1 Problem Definition

Database query execution is I/O-expensive, and physical optimization concentrates on reducing I/O costs based on the physical storage scheme and data distribution. Unlike relational databases, XML data are ordered and hierarchically organized. More importantly, the XML schema is more flexible and allows mixed content. Existing extended-relational approaches shred XML documents into small pieces (elements) and store them without considering their structural relationships. To answer queries related to structural constraints, it is necessary

to perform a structural join on each structural constraint, which could pose optimization difficulties. To efficiently answer such queries, we need to study the following problems:

Path expressions: Path expression is arguably the most natural way to query tree-structure data. It deserves attention because, first, it is one of the most heavily used expressions in XQuery, and its performance will greatly affect the overall performance. Second, it is significantly different from relational query languages, so it may require new optimization techniques.

Physical storage structures: As discussed above, relational storage structure is invariant to ordering. Therefore, a new storage scheme taking into consideration the ordering of a an tree structure may be advantageous for relational storage structure.

4.2 Preliminary Results

To answer path queries, we have designed a succinct XML physical storage scheme [6], in which schema information (tree structure consisting of tags) and data information (element contents attached to the leaves of the subject tree) are stored separately. The reason for the separation is that, first, the physical management issues are much easier when we consider only one type of data — the tree structure without variable-length element contents is more regular and can be managed more efficiently, and content-based indexes (such as B^+ trees and suffix trees) can be created only on the content information without worrying about its structure. Second, queries can be decomposed into operations that operate on these two types of information individually, and the final result can be composed by the partial results from the two classes of operators.

Another requirement for the storage structure design is to efficiently answer tree-structure related queries. The rationale is to *cluster* XML elements at the physical level based on one of the "local" structural relationships (say parent-child). The idea is to linearize the tree nodes in pre-order but keep balanced parentheses to denote the beginning and ending of a subtree. This clustering method makes update easier since each update only affects a local sub-string. Furthermore, pre-order of the tree nodes coincides with the streaming XML element arrival order. So the path query evaluation algorithm (outlined below) can also be used in the streaming context.

We have also identified a subset of the path expression, which we call next-of-kin (NoK) expressions, consisting of only those local structural relationships. The evaluation of NoK expressions can be performed more efficiently using a navigational technique based on our physical storage structures without the need for structural joins. Given a general path expression, we first partition it into interconnected NoK expressions, to which we apply the more efficient navigational pattern matching algorithm. Then, we join the results of the NoK pattern matching based on their structural relationships, just as in the join-based approach. Experimental results show that our approach outperforms existing join-based approaches and a state-of-the-art commercial native XML management system [6].

5 Related Work

The definition of our algebra is inspired by previous algebras (in particular, YAT algebra [7] and TAX algebra [8, 9]). In YAT algebra, operators can be separated into three categories: (1) Bind operators take a collection of XML documents and produce a set of structures called *Tab*, which are ¬1NF relations, as output. (2) Then any standard relational operators (such as Selection, Projection, Join, and Union) can be applied to the *Tab* structure. (3) Finally, Tree operators are applied to *Tab* structures to generate new nested XML documents. In our algebra, we also define some "border" operators that insulate other more "relational-style" operators from the tree data model. Therefore, we can reuse the optimization techniques developed for relational databases. In addition, as discussed in Section 3.2, more sorts and operators are needed to support efficient query processing.

In TAX algebra [8], the input, as well as the output, of any operator is a collection of labeled trees. A great difficulty for this approach is that trees are more heterogeneous than tuples (relations). When defining operators, specifying the objects (tree nodes) to manipulate is therefore much more difficult than that for tuples. By comparison, relational algebra specifies attributes by their names or their positions in the tuple; while TAX algebra specifies tree nodes by a tree pattern (or generalized tree pattern in [9]), to specify certain nodes by giving the constraints that these nodes should satisfy. Therefore, TAX algebra generalizes relational algebra in terms of how to specify objects to manipulate, but also introduces complexity at the same time. We shall see that in some cases, a pattern tree is an overkill for some intermediate results, which can be represented by lists or nested lists and thus can be manipulated more efficiently by operations on these specialized data structures.

Previous research on the evaluation and optimization of path expressions fall into two classes. *Navigational* approaches traverse the tree structure and test whether a tree node satisfies the constraints specified by the path expression [10]. *Join-based* approaches first select a list of XML tree nodes that satisfy the node-associated constraints for each pattern tree node, and then pairwise join the lists based on their structural relationships [11–14]. Compared to the navigational techniques, join-based approaches are more scalable and enjoy optimization techniques from the relational database technology. However, there are inevitable difficulties because join-based approach usually results in many structural joins. In Section 4, we follow a novel approach that combines the advantages of both navigational and join-based approaches.

6 Open Problems and Planned Work

In this paper, I have presented my preliminary work on the definition of algebra and design of physical storage and optimization techniques. What remains is to define a complete set of operators, prove the soundness and completeness properties in a large fragment of XQuery (basically XQuery without recursive

functions), and develop logical optimization techniques. Another primary goal of my research is to demonstrate how extensively this algebra accommodates optimization techniques, and how effective these techniques are in practice. I will justify the efficiency of the algebra by defining rewrite rules and investigating different evaluating strategies such as lazy evaluation (or output-oriented) strategy. We have shown (as in Fig. 1(b)) that the output template (SchemaTree) can be extracted from an XQuery expression. The remaining work is to show how to further generate an execution plan by backward (from output to input) analysis.

References

1. DeHann, D., Toman, D., Consens, M.P., Özsu, M.T.: A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (2003) 623–634
2. Fankhauser, P., Fernandez, M., Malhotra, A., Rys, M., Simeon, J., Wadler, P.: XQuery 1.0 Formal Semantics. Available at http://www.w3.org/TR/query-semantics/ (2004)
3. Kepser, S.: A Proof of the Turing-completeness of XSLT and XQuery. Technical report, Unviersity of Tübingen (2002)
4. Gottlob, G., Koch, C., Pichler, R.: Efficient Algorithms for Processing XPath Queries. In: Proc. 28th Int. Conf. on Very Large Data Bases. (2002) 95–106
5. Wu, Y., Patel, J.M., Jagadish, H.V.: Structural Join Order Selection for XML Query Optimization. In: Proc. 19th Int. Conf. on Data Engineering. (2003)
6. Zhang, N., Kacholia, V., Özsu, M.T.: A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In: Proc. 20th Int. Conf. on Data Engineering. (2004)
7. Christophides, V., Cluet, S., Simeon, J.: On Wrapping Query Languages and Efficient XML Integration. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (2000) 141–152
8. Jagadish, H., Lakshmanan, L.V.S., Srivastava, D., Thompson, K.: TAX: A Tree Algebra for XML. In: Proc. 8th Int. Workshop on Database Programming Languages. (2001)
9. Chen, Z., Jagadish, H.V., Lakshmanan, L.V.S., Paparizos, S.: From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In: Proc. 29th Int. Conf. on Very Large Data Bases. (2003)
10. Simeon, J., Fernandez, M.: Galax (2004) Available at http://www-db-out.bell-labs.com/galax/.
11. Zhang, C., Naughton, J., Dewitt, D., Luo, Q., Lohman, G.: On Supporting Containment Queries in Relational Database Management Systems. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (2001) 425–436
12. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In: Proc. 18th Int. Conf. on Data Engineering. (2002) 141–152
13. Bruno, N., Koudas, N., Srivastava, D.: Holistic Twig Joins: Optimal XML Pattern Matching. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (2002) 310–322
14. Shasha, D., Wang, J.T.L., Giugno, R.: Algorthmics and Applications of Tree and Graph Searching. In: Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems. (2002) 39–53