Structural Summaries for Efficient XML Query Processing

Sofía Brenes
Advisor: Yuqing Melanie Wu
Computer Science Department,
Indiana University, Bloomington
sbrenesb@cs.indiana.edu

ABSTRACT

The importance of performing efficient XML query processing increases along with its usage and pervasiveness. Studying the properties of important fragments of XML query languages and designing accurate structural summaries (including indexes and statistical summaries) are all critical ingredients in solving this problem. However, up to this point there has been a gap between the theoretical and engineering efforts taken in the context of XML. We draw from research methodologies used in relational query languages and database design and apply it to the study of XPath and the design of structural summaries for XML. In particular, we study the roles various fragments of XPath algebra play in distinguishing data components in an XML document, and leverage the results in designing novel structural indexes and statistical summaries for more efficient XML query processing and more accurate result size estimation.

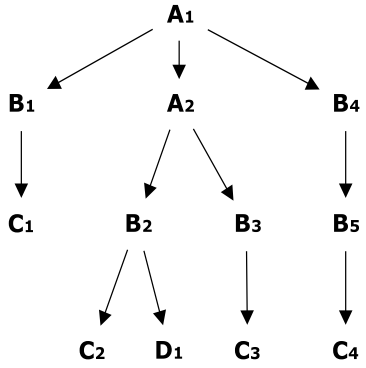

Figure 1: A sample XML document (Subscripts identify instances of elements with the same tag)

1. INTRODUCTION

The available types of data and the content of the data itself have evolved quickly to surpass the capabilities of traditional relational databases. Semi-structured data, particularly XML, has become a standard for information exchange and representation. As part of its fundamental properties, semi-structured data has been identified in [1] as having an irregular, implicit, and partial structure. As shown in Figure 1, element A may have B sub-element(s) and nested A sub-element(s); and element B may or may not have a D sub-element. An XML document schema can change at any time, or even be omitted. This poses a benefit in terms of flexibility, but also a challenge for properly modeling, storing, and querying XML data.

The two most popular XML query languages are XPath [28] and XQuery [29]. XPath navigates the nodes of an XML document using path expressions which may or may not contain branching conditions. XQuery is an extension of XPath that uses a FLWR (For, Let, Where, Return) expression to specify desired results. It allows for more powerful queries to be performed and for results to be constructed independently of the source XML document. Given that XPath is at the core of XQuery and other XML query languages such as XSLT [30], efficiently evaluating XPath queries is essential to the efficient processing of XML queries in general.

Since the debut of XML, extensive research has been done in the theoretical study of XML query languages and the engineering design of XML data engines. However, there is a gap between the two areas that prevents the engineering design from taking full advantage of the results in the theoretical study, an approach that has been successfully taken in the context of relational databases.

In this paper, we focus on the study of XPath algebra and some of its important fragments, in terms of their capability in distinguishing data components in an XML document. We show how to leverage the results in the design of structural indexes that lead to efficient processing of XPath queries in general. We then discuss future research topics that take advantage of this study in the design of statistical summaries and result size estimation algorithms.

The rest of this paper is organized as follows: Section 2 provides an overview of related work, in Section 3 we define the problem we address in our research, Section 4 presents the preliminary results we have obtained in the formal study of the coupling of language and document induced partitions of XML. Section 5 discusses the implications of our work and how this will guide our research in the design of a new family of structural indexes for XML. Finally, Sections 6 and 8 present the conclusions of our work and the feedback received from the presentation in the conference workshop.

EDBT 2008 Ph.D. Workshop '08 March 25, 2008, Nantes, France

2. RELATED WORK

2.1 Theoretical Study of XML Query Languages

The theory underlying the XQuery and XPath query languages shares its foundations with the theory of query languages for relational, object-oriented, and semi-structured databases, with finite model theory at the core of much of this work. Several natural semantic issues have been investigated in recent years for various fragments of XPath [13, 19], including expressibility, closure properties, complexity of evaluation [4, 11, 20], and decision problems such as satisfiability, containment, and equivalence [3, 21]. These properties can be used to formulate useful query rewrite rules for query optimization. Other types of rewrite rules [23] focus on eliminating XPath ancestor axes to enable more efficient query processing.

2.2 XML Indexing Techniques

As in any large data repository, the existence and proper use of indexes greatly improves the performance of certain queries. When considering XML data, the structural information that is inherently encoded in a document provides a new dimension for performing queries and creating indexes. Over twenty different types of indexes, as summarized in [12], have been proposed in the past few years and have led to significant improvements in the performance of XML query evaluation. Based on their types, they can be categorized into a few big families.

2.2.1 Value Indexes

Indexes similar to those used in relational databases, namely value indexes on element tags, attribute names, and text values, were first used together with structural join algorithms [2, 5] in XML query evaluation. This approach turns out to be simple and efficient, but fails to capture the structural containment relationships native to XML data.

2.2.2 Structural Indexes

A family of structural indexes were introduced to directly capture XML structural information. DataGuides [10] provide a concise and accurate dynamic structural summary of a semi-structured database by describing every unique path exactly once. Each path has an associated "target set" which contains all objects reachable by traversing it. Milo *et al* [22] improved the DataGuides through the T-Index in which classes of paths are associated with an index through an arbitrary path template specification. For example, a template can specify all nodes that are reachable from the root (the 1-Index), or node pairs connected by a particular path (the 2-Index).

Unfortunately, these index structures are usually too large to be effectively used in practice. To solve this problem, Kaushik *et al* [18] proposed the $\mathcal{A}[k]$-index, which uses the notion of upward node bi-similarity limited to paths of length k. Two nodes are considered to be k-bisimilar if they share the same label and their incoming paths of length $\leq k$ are the same. The size of an $\mathcal{A}[k]$-index and the degree to which it helps in query evaluation varies upon the selection of the parameter k. However, queries with branching predicates cannot be answered directly using an $\mathcal{A}[k]$-index. Validation is required, resulting in expensive disk access.

The Forward and Backward-Index (F&B-Index) [16] groups nodes by considering both incoming and outgoing paths and is capable of handling queries with branches. However, as pointed out by [32], the F&B-Index is too large to fit in memory and does not provide fully optimized methods for answering queries.

2.2.3 Workload-aware Indexes for XML

In an $\mathcal{A}[k]$-index, the k parameter determines both the size and the usefulness of the index. Chen *et al* [6] argued that not all structure has equivalent significance. Their $D(k)$-index is an adaptive structural summary that allows different similarity values for its indexed paths, adjusting them according to query workload.

In [14], He *et al* improved the update algorithms in the $D(k)$-index to create the $M(k)$-index. In their approach, the update algorithm never over-refines unnecessary paths. They also proposed the $M^*(k)$-index which is a collection of $M(k)$-indexes with different resolutions connected to one another.

2.2.4 Other XML Indexes

Other directions in XML indexing techniques include indexing frequent query sub-patterns [7], or indexing XML trees and queries as sequences. ViST [31] represents both XML documents and queries as structure-encoded sequences. [26] and [31] use $Prüfer$ sequences and string indexes respectively. In [24], XML documents are indexed using forms of prefix-tree encodings.

An index selection tool [27] uses algorithms that consider query workload information and cost-benefit formulae to propose both the paths to be included in an index, and also the best index to use for a given query.

2.3 Query Processing and Optimization

A key issue in XML query processing is finding all occurrences of the structural relationships expressed in XML queries. This is an issue that arises in both relational database systems that support XML, and in native XML query engines. The problem has been addressed in the latter with two different approaches: structural indexes (as discussed in Section 2.2) and structural join algorithms [2, 5].

Optimization techniques have been proposed to choose efficient evaluation plans based on the availability of indexes and statistical information [8, 33]. These optimization techniques rely on well-defined cost models and accurate estimates of cardinalities of intermediate results to function effectively. Statistical summaries, such as histograms [34] and XSketch [25], and their corresponding usage and maintenance algorithms, have been proposed for providing such estimates.

3. PROBLEM DEFINITION

Theoretical results have successfully guided the design and implementation of relational database management systems (RDBMS), as witnessed by the systematical theoretical study of relational algebra, relational calculus, and the relational query language.

The coupling between theory and engineering is exemplified in the design of indexes used in a RDBMS. In this case, range queries select tuples from a relational table satisfying the condition that its values on a certain attribute set fall within a given range. The tuples are partitioned into groups, each group featuring a unique, distinct value on the

given attribute set. The result of any range query is composed of the tuples in such a group, or the union of a set of such groups. A B^+-tree index partitions tuples by the values of a set of attributes (the index key) and organizes the groups according to the distance between their index key values. The B^+-tree index provides a good coupling with the theory behind range queries and as such, can answer them efficiently. A histogram, which keeps track of the number of tuples that satisfy a certain value condition, is bound to provide an accurate selectivity estimate of range queries.

Since the debut of XML in the late 1990's there have been numerous efforts in both the theoretical and engineering research for this data model. However, the type of tight coupling between the two, as seen in the relational database context and shown in the above example, has yet to emerge. This leads to a series of open questions in this research area, among which we have identified:

1. For each family of XML indexes, which classes of XML queries are these indexes ideally suited for?
2. For each class of XML queries, which XML index structure works ideally for answering these queries?
3. How can the partition classes represented by each type of index be labeled, and how to efficiently locate the partition classes that contribute to the query evaluation for a given set of queries?
4. How to choose a query evaluation plan that takes full advantage of available indexes?
5. How to use query workload information to further optimize the structural indexes?
6. How to capture structural information of XML data for more accurate result size estimation for each class of queries?

In this proposed research, we adopt a methodology that is drawn from the classical study of relational queries, the design of relational indexes, and query evaluation techniques and extend it to the context of XML. Rather than attempting to tackle all the open problems listed above, we will focus on certain fragments of XPath. In particular, we will:

1. Analyze the structural features of XML documents and study important fragments of XPath algebra and their properties.
2. Design novel structural indexes and corresponding index access and maintenance algorithms for efficient query evaluation.
3. Design workload-aware structural indexes and algorithms for constructing, accessing and updating such indexes.
4. Design summary data structures for accurate result size estimation.

4. PRELIMINARY RESULTS

At this stage, we have proposed the notion of *label-paths* and *label-path based partitions* of nodes and pairs in an XML document. We have studied a family of algebras for sublanguages of XPath and discovered the coupling between a partition of XML data induced from the algebra and a partition induced from purely analyzing the structure of the data.

We are using these results in the design and implementation of the $\mathcal{N}[k]$-Trie and $\mathcal{P}[k]$-Trie structural indexes for efficient query evaluation. In this section, we will present these preliminary results and then discuss our ideas on further utilizing the results in the design of workload-aware structural indexes and the design of statistical structural summaries and the corresponding result size estimation algorithms in Section 5.

4.1 Label-Path Based Partitions

An XML document X is a node-labeled tree. Formally, we define it as a 4-tuple (V, Ed, r, λ), with V the finite set of nodes, $Ed \subseteq V$ x V the set of edges, $r \in V$ the root, and $\lambda\colon V \to \mathcal{L}$ a node-labeling function into the set of labels $\mathcal{L}$.

For a given pair of nodes m and n in an XML document X where m is an ancestor of n, we define its associated *label-path* to be the unique path between m and n, denoted $LP(m, n)$. We use $DownPairs(X)$ to represent the set of all such pairs in X, and use $DownPairs(X, k)$ to represent the set of node pairs such that (1) m is an ancestor of n in X, and (2) $length(m, n)$[1] $\leq k$.

Given a node n in X, and a number $k \in \mathbb{N}$, we define the *k-label-path* of n, denoted $LP(n, k)$, to be the label-path of the unique downward path of length l into n where $l = \min\{height(n), k\}$.[2]

EXAMPLE 4.1. *Take the XML document shown in Figure 1. We have that* $(A_2, C_2) \in DownPairs(X, 2)$ *but* $(A_2, C_2) \notin DownPairs(X, 1)$. *The label-path between these nodes is* $LP(A_2, C_2) = (A, B, C)$. *Similarly,* $LP(C_1, 1) = (B, C)$ *and* $LP(C_1, 5) = (A, B, C)$.

We use the notion of label-paths to define $\mathcal{N}[k]$-equivalence.

DEFINITION 4.1. *Let* $X = (V, Ed, r, \lambda)$ *be an XML document, and let* $k \in \mathbb{N}$. *We say that nodes* n_1 *and* n_2 *in* V *are* $\mathcal{N}[k]$-equivalent *(denoted* $n_1 \equiv_{\mathcal{N}[k]} n_2$*) if they have the same k-label-path, i.e.,* $LP(n_1, k) = LP(n_2, k)$.

The $\mathcal{N}[k]$-*partition* of X is then defined as the partition induced by this equivalence relation. It immediately follows that each partition class C in the $\mathcal{N}[k]$-*partition* can be associated with a unique label-path, the label-path of the nodes in C, denoted $LP(C)$. On the other hand, a k-label-path p in an XML document X uniquely identifies an $\mathcal{N}[k]$-*partition* class, which we denote as $\mathcal{N}[k][p]$. In [9] we proved that the $\mathcal{N}[k]$-partition and the $\mathcal{A}[k]$-partition are the same.

DEFINITION 4.2. *Let* $X = (V, Ed, r, \lambda)$ *be an XML document, and let* $k \in \mathbb{N}$. *We define the k-pair equivalence relation on the set* $DownPairs(X, k)$ *as follows: two pairs* (m_1, n_1) *and* (m_2, n_2) *in* $DownPairs(X, k)$ *are* $\mathcal{P}[k]$-*equivalent (denoted* $(m_1, n_1) \equiv_{\mathcal{P}[k]} (m_2, n_2)$*) if they have the same label-path, i.e.,* $LP(m_1, n_1) = LP(m_2, n_2)$.

The $\mathcal{P}[k]$-*partition* of X is then defined as the partition on $DownPairs(X, k)$ induced by this equivalence relation. Similar to the $\mathcal{N}[k]$-*partition*, label-paths can be associated with each partition class in a $\mathcal{P}[k]$-*partition*, which we denote as $\mathcal{P}[k][p]$.

[1] $length(m, n)$ denotes the length of the unique path between m and n in X.

[2] $height(n)$ denotes the height of node n in X.

$\mathcal{N}[1]$-partition	$\mathcal{N}[2]$-partition
$\mathcal{N}[1][(A)] = \{A_1\}$ $\mathcal{N}[1][(A,A)] = \{A_2\}$	$\mathcal{N}[2][(A)] = \{A_1\}$ $\mathcal{N}[2][(A,A)] = \{A_2\}$
$\mathcal{N}[1][(A,B)] = \{B_1, B_2, B_3, B_4\}$ $\mathcal{N}[1][(B,B)] = \{B_5\}$	$\mathcal{N}[2][(A,B)] = \{B_1, B_4\}$ $\mathcal{N}[2][(A,A,B)] = \{B_2, B_3\}$ $\mathcal{N}[2][(A,B,B)] = \{B_5\}$
$\mathcal{N}[1][(B,C)] = \{C_1, C_2, C_3, C_4\}$	$\mathcal{N}[2][(A,B,C)] = \{C_1, C_2, C_3\}$ $\mathcal{N}[2][(B,B,C)] = \{C_4\}$
$\mathcal{N}[1][(B,D)] = \{D_1\}$	$\mathcal{N}[2][(A,B,D)] = \{D_1\}$

Table 1: The $\mathcal{N}[1]$ and $\mathcal{N}[2]$-partitions and label-paths of the sample XML document

$\mathcal{P}[1]$-partition	$\mathcal{P}[2]$-partition
$\mathcal{P}[1][(A)] = \{(A_1,A_1),(A_2,A_2)\}$ $\mathcal{P}[1][(B)] = \{(B_1,B_1),(B_2,B_2),(B_3,B_3),(B_4,B_4),(B_5,B_5)\}$ $\mathcal{P}[1][(C)] = \{(C_1,C_1),(C_2,C_2),(C_3,C_3),(C_4,C_4)\}$ $\mathcal{P}[1][(D)] = \{(D_1,D_1)\}$	$\mathcal{P}[2][(A)] = \{(A_1,A_1),(A_2,A_2)\}$ $\mathcal{P}[2][(B)] = \{(B_1,B_1),(B_2,B_2),(B_3,B_3),(B_4,B_4),(B_5,B_5)\}$ $\mathcal{P}[2][(C)] = \{(C_1,C_1),(C_2,C_2),(C_3,C_3),(C_4,C_4)\}$ $\mathcal{P}[2][(D)] = \{(D_1,D_1)\}$
$\mathcal{P}[1][(A,A)] = \{(A_1,A_2)\}$ $\mathcal{P}[1][(A,B)] = \{(A_1,B_1),(A_2,B_2),(A_2,B_3),(A_1,B_4)\}$ $\mathcal{P}[1][(B,B)] = \{(B_4,B_5)\}$ $\mathcal{P}[1][(B,C)] = \{(B_1,C_1),(B_2,C_2),(B_3,C_3),(B_5,C_4)\}$ $\mathcal{P}[1][(B,D)] = \{(B_2,D_1)\}$	$\mathcal{P}[2][(A,A)] = \{(A_1,A_2)\}$ $\mathcal{P}[2][(A,B)] = \{(A_1,B_1),(A_2,B_2),(A_2,B_3),(A_1,B_4)\}$ $\mathcal{P}[2][(B,B)] = \{(B_4,B_5)\}$ $\mathcal{P}[2][(B,C)] = \{(B_1,C_1),(B_2,C_2),(B_3,C_3),(B_5,C_4)\}$ $\mathcal{P}[2][(B,D)] = \{(B_2,D_1)\}$
	$\mathcal{P}[2][(A,A,B)] = \{(A_1,B_2),(A_1,B_3)\}$ $\mathcal{P}[2][(A,B,B)] = \{(A_1,B_5)\}$ $\mathcal{P}[2][(A,B,C)] = \{(A_1,C_1),(A_2,C_2),(A_2,C_3)\}$ $\mathcal{P}[2][(A,B,D)] = \{(A_2,D_1)\}$ $\mathcal{P}[2][(B,B,C)] = \{(B_4,C_4)\}$

Table 2: The $\mathcal{P}[1]$ and $\mathcal{P}[2]$-partitions and label-paths of the sample XML document

EXAMPLE 4.2. *Tables 1 and 2 show the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-partitions of the sample XML document shown in Figure 1 for $k = 1, 2$.*

4.2 Families of XPath Algebras

The syntax and *path semantics* of the XPath algebra, as proposed in [11] and [13], are defined as follows:

$$
\begin{aligned}
\varepsilon(X) &= \{(n,n) \mid n \in V\} \\
\emptyset(X) &= \emptyset \\
\downarrow(X) &= Ed \\
\uparrow(X) &= Ed^{-1} \\
\ell(X) &= \{(n,n) \mid n \in V \;\&\; \lambda(n) = \ell\} \\
E_1 \circ E_2(X) &= \{(n,m) \mid \exists w\colon (n,w) \in E_1(X) \,\&\, (w,m) \in E_2(X)\} \\
E_1[E_2](X) &= \{(n,m) \in E_1(X) \mid \exists w\colon (m,w) \in E_2(X)\} \\
E_1 * E_2(X) &= E_1(X) * E_2(X) \qquad \text{where } * \text{ is } \cap,\ \cup \text{ or } -
\end{aligned}
$$

The *node semantics* of an expression E on an XML document X, denoted $E(X)[\mathbf{nodes}]$[3] is defined as the set $\{n \mid \exists m : (m,n) \in E(X)\}$.

We focus our study on a few sub-algebras of XPath. The $\mathcal{D}$ algebra consists of the expressions in the XPath algebra without occurrences of the set operators, predicates ($[]$), or the $\uparrow$ primitive. The $\mathcal{D}^{[]}$ algebra consists of the $\mathcal{D}$ algebra plus predicates. As in [9], the $\mathcal{D}^{[]}[k]$ algebras are recursively defined on k as follows:

1. $\mathcal{D}^{[]}[0]$ is the set of expressions in the $\mathcal{D}^{[]}$ without occurrences of the $\downarrow$ primitive.
2. For $k \geq 1$,
   (a) if $E \in \mathcal{D}^{[]}[k-1]$, then $E \in \mathcal{D}^{[]}[k]$;
   (b) $\downarrow \in \mathcal{D}^{[]}[1]$;
   (c) if $E_1 \in \mathcal{D}^{[]}[k_1]$, $E_2 \in \mathcal{D}^{[]}[k_2]$, and $k_1 + k_2 \leq k$, then $E_1 \circ E_2 \in \mathcal{D}^{[]}[k]$ and $E_1[E_2] \in \mathcal{D}^{[]}[k]$.
3. No other expressions are in $\mathcal{D}^{[]}[k]$.

EXAMPLE 4.3. *Given an XPath algebra expression $E = A\circ\downarrow\circ B\circ\downarrow\circ C$ we have that $E \notin \mathcal{D}^{[]}[1]$ but $E \in \mathcal{D}^{[]}[2]$.*

The $\mathcal{D}[k]$ algebra is defined as the set of expressions in the $\mathcal{D}^{[]}[k]$ algebra without occurrence of predicates. Similarly, we can define the $\mathcal{U}$, $\mathcal{U}^{[]}$, $\mathcal{U}^{[]}[k]$ and $\mathcal{U}[k]$ algebras, which feature the $\uparrow$ primitive instead of $\downarrow$.

We now define the *DownUp* algebra in terms of expressions in the $\mathcal{D}$ algebra and $\mathcal{U}$ algebra: given a list of expressions $E_1, E_2, \ldots E_m$ such that E_i is an expression either in the $\mathcal{D}$ algebra or in the $\mathcal{U}$ algebra, then the expression

$$E_1 \circ E_2 \circ \cdots \circ E_{m-1} \circ E_m$$

is defined as an expression in the *DownUp* algebra. A *DownUp[k]* algebra is defined as a *DownUp* algebra in which each *run* (E_i) is an expression in either $\mathcal{D}[k]$ or $\mathcal{U}[k]$.

[3]Since it will always be clear from the context, we will often use the notation $E(X)$ to denote $E(X)[\mathbf{nodes}]$.

	XPath	$\mathcal{D}$ Expression	LPS	Evaluation
q_1	$//A/A/B$	$A\circ\downarrow\circ A\circ\downarrow\circ B$	{(A,A,B)}	$q_1(X)[\textbf{nodes}] = \mathcal{N}[2][(A,A,B)](X)$ $q_1(X) = \mathcal{P}[2][(A,A,B)](X)$
q_2	$//A/*/B$	$A\circ\downarrow\circ\downarrow\circ B$	{(A,A,B), (A,B,B)}	$q_2(X)[\textbf{nodes}] = \mathcal{N}[2][(A,A,B)](X) \cup \mathcal{N}[2][(A,B,B)](X)$ $q_2(X) = \mathcal{P}[2][(A,A,B)](X) \cup \mathcal{P}[2][(A,B,B)](X)$

Table 3: Evaluating $\mathcal{D}$ expressions using $\mathcal{N}[2]$ and $\mathcal{P}[2]$-partitions

It follows that each expression of the $\mathcal{D}^{[]}$ algebra can be translated into an equivalent *DownUp* expression with the following at the core of this translation:

LEMMA 4.1. *Let D be an XML document and let E_1 and E_2 be expressions in the $\mathcal{D}^{[]}$ algebra. Then*

$$E_1[E_2](D) = E_1(D) \circ E_2(D) \circ (E_2(D))^{-1}.$$

Lemma 4.1 provides the foundation that allows us to transform an XPath expression with predicates to an expression without predicates. Furthermore, an arbitrary XPath expression can be decomposed into sub-queries that are linear and whose length is $\leq k$. Thus, Proposition 4.1 and Corollary 4.1 follow.

PROPOSITION 4.1. *For each expression $E \in \mathcal{D}^{[]}$ there exists an equivalent expression F_E in the DownUp algebra.*

COROLLARY 4.1. *For each expression $E \in \mathcal{D}^{[]}$ there exists an equivalent expression F_E in the DownUp[k] algebra.*

4.3 Coupling $\mathcal{D}^{[]}$ Algebras with the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-partitions of an XML Document

We now discuss how the $\mathcal{D}^{[]}$ algebras can be coupled with the $\mathcal{N}[k]$-*partition* and $\mathcal{P}[k]$-*partition* of an XML document.

Given an XML document $X = (V, Ed, r, \lambda)$ and an expression $E \in \mathcal{D}$, we define the label-path set of E in an XML document X (denoted $LPS(E, X)$) as the set of label-paths in X that satisfy the node-labels and structural containment relationships specified by E.

EXAMPLE 4.4. *For the sample XML document in Figure 1 and the XPath expression $//A/*/B$ the corresponding $LPS(E, X) = \{(A, A, B), (A, B, B)\}$*

Given that a label-path uniquely identifies an $\mathcal{N}[k]$-*partition* class of nodes (or a $\mathcal{P}[k]$-*partition* class of node pairs) of an XML document, expressions in $\mathcal{D}$ can be evaluated via unions of the corresponding partition classes. Given an XML document X and an expression $E \in \mathcal{D}$,

$$E(X)[\textbf{nodes}] = \bigcup_{lp \in LPS(E,X)} \mathcal{N}[\infty][lp](X)$$
$$E(X) = \bigcup_{lp \in LPS(E,X)} \mathcal{P}[\infty][lp](X)$$

We now define the refinement relationship ($\prec$) between two equivalence relations:

DEFINITION 4.3. *Given a data model M and two equivalence relations $\equiv_{\mathcal{A}}$ and $\equiv_{\mathcal{B}}$, we say that $\mathcal{A} \prec \mathcal{B}$ if for each data instance m over M, the equivalence relation $\equiv_{\mathcal{A}}$ is a refinement of the equivalence relation $\equiv_{\mathcal{B}}$. We say $\mathcal{A} \simeq \mathcal{B}$ if $\mathcal{A} \prec \mathcal{B}$ and $\mathcal{B} \prec \mathcal{A}$.*

We can prove that under both the node and path semantics of XPath:

$$\mathcal{D}[k] \simeq \mathcal{N}[k] \quad \text{and} \quad \mathcal{D} \simeq \mathcal{N}[\infty]$$
$$\mathcal{D}[k] \simeq \mathcal{P}[k] \quad \text{and} \quad \mathcal{D} \simeq \mathcal{P}[\infty]$$

Given that $\mathcal{D} \simeq \mathcal{U}$ and $\mathcal{D}[k] \simeq \mathcal{U}[k]$, similar coupling relationships can be established between the $\mathcal{U}$ algebra and the $\mathcal{N}[k]$ and $\mathcal{P}[k]$- partitions. This leads to the design principle that *indexes based on the $\mathcal{N}[k]$ and $\mathcal{P}[k]$- partitions are suitable for answering queries in $\mathcal{D}$ and $\mathcal{U}$*. For example, given the document shown in Figure 1, the evaluation of two sample $\mathcal{D}$ queries is shown in Table 3.

5. RESEARCH DIRECTION

5.1 Trie Indexes for XML

Good index structures facilitate efficient query evaluation by strategically partitioning, labeling and organizing the source data such that:

1. the unions of some partition classes are the proper super set of the result of queries;
2. such partition classes can be easily located via search over the partition labels;
3. the partition classes are fine enough such that the search space is efficiently reduced via index access; and
4. partition classes that are likely to contribute to a query answer are stored physically together to reduce I/O cost.

In [13], it is suggested that the study of the XPath algebra could have an impact on indexing XML documents. This step was first taken in [9] by analyzing the connection between certain upward XPath algebras and structural indexes such as the $\mathcal{A}[k]$-index and DataGuides. As previously mentioned, we proved that *the $\mathcal{N}[k]$-partition and $\mathcal{A}[k]$-partition are the same* [9] and established the refinement and coupling relationships between the $\mathcal{U}$-algebras and the $\mathcal{N}[k]$ and $\mathcal{P}[k]$ partitions.

Since $\mathcal{D}[k] \simeq \mathcal{N}[k] \simeq \mathcal{A}[k]$ and $\mathcal{U} \simeq \mathcal{D}$, DataGuides and the $\mathcal{A}[\infty]$-index are suitable for answering queries in $\mathcal{D}$ and $\mathcal{U}$. The $\mathcal{A}[k]$-index trades the degree of bi-similarity in favor of a smaller size of the index graph, and is suitable for answering queries in $\mathcal{D}[k]$ and $\mathcal{U}[k]$, but not $\mathcal{D}$ and $\mathcal{U}$ when $k < height(D)$. This echoes the findings discussed in Section 4.2 and 4.3.

Here, we take a further step in leveraging the above discovery in proposing new index structures for efficient XML query evaluation.

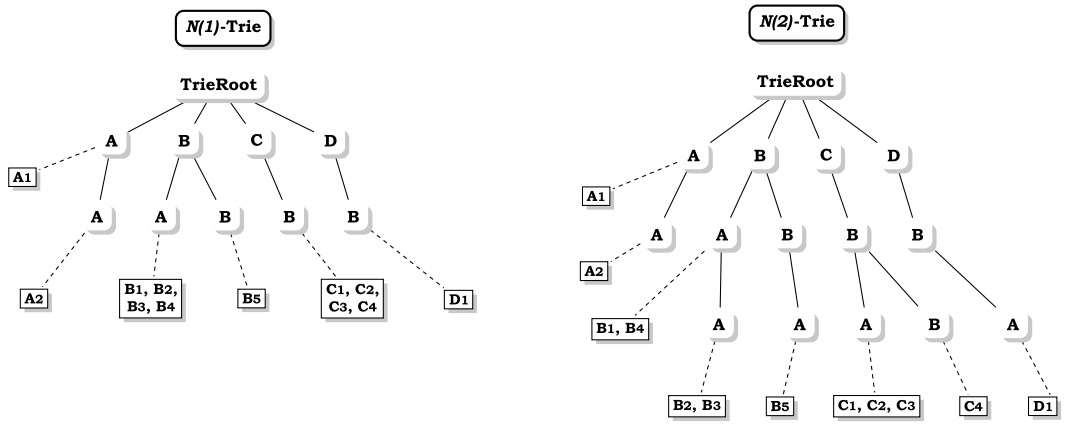

Figure 2: $\mathcal{N}[1]$ and $\mathcal{N}[2]$-Trie Index

5.1.1 $\mathcal{N}[k]$-Trie Index

Focusing on the $\mathcal{N}[k]$ partition of data nodes, we propose to organize the $\mathcal{N}[k]$-*partition* classes in a trie structure, using the reversed label-path of each partition class as the key. We use a reversed label-path because the answer to an expression requires sharing of a common suffix, and a trie structure guarantees sharing of a common prefix. We denote this index an $\mathcal{N}[k]$-Trie index.

EXAMPLE 5.1. *Figure 2 shows the $\mathcal{N}[1]$ and $\mathcal{N}[2]$-Trie indexes of the sample XML document shown in Figure 1. Notice that all leaf nodes in an $\mathcal{N}[k]$-Trie have associated partition blocks, while non-leaf nodes only have associated partition blocks when the trie node label is the same as the root of the XML document. The height of the trie is no larger than k and the trie reflects all paths in the document of length $\leq k$. See section 5.1.3 for more properties of the $\mathcal{N}[k]$-Trie.*

A carefully designed index lookup algorithm distinguishes between two types of lookup operations in the trie structure: a normal trie lookup, and a subtree lookup.

Given an $\mathcal{N}[k]$-Trie index, the normal trie lookup is suitable for answering chain queries that are of length k, by returning the index item associated with the reversed path expression.

The subtree lookup is suitable for answering chain queries that are shorter than k, by returning the index items in the subtree rooted at the node associated with the reversed path expression.

5.1.2 $\mathcal{P}[k]$-Trie Index

None of the structural indexes in the literature feature both a manageable size and the capability of answering queries in the more general algebra of $\mathcal{D}^{[]}$ efficiently with an index-only plan.

A query q in $\mathcal{D}^{[]}$ can be decomposed into multiple *runs* (sub-queries) of queries in $\mathcal{D}$ and $\mathcal{U}$. To obtain the result of q, the results of these *runs* need to be stitched together, requiring the intermediate results to be in the form of *pairs* rather than *nodes*. In other words, it requires that the sub-queries be evaluated in the *path* semantics, rather than the *node* semantics.

The $\mathcal{P}[\infty]$-*partition* serves this purpose well. As per Corollary 4.1, an index structure based on the $\mathcal{P}[k]$-*partition* with a modest k would serve just as well. Thus, we propose the $\mathcal{P}[k]$-Trie index which organizes the $\mathcal{P}[k]$-*partition* classes of an XML document in a trie structure, using the reversed label-paths as the keys.

EXAMPLE 5.2. *Figure 3 shows the $\mathcal{P}[1]$ and $\mathcal{P}[2]$-Trie indexes of the sample XML document. Please note that unlike the $\mathcal{N}[k]$-Trie,* every *node in the $\mathcal{P}[k]$-Trie has an associated partition block. As in the $\mathcal{N}[k]$-Trie, the height of the trie is no larger than k, and the trie reflects all paths in the document of length $\leq k$. In fact, the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie have the same structure. See section 5.1.3 for more properties of the $\mathcal{P}[k]$-Trie.*

5.1.3 Trie Index Structural Properties

We summarize some properties of the trie indexes that may provide insight into the study of the workload-aware Trie indexes and statistical summary structures for XML:

1. The height of an $\mathcal{N}[k]$-Trie ($\mathcal{P}[k]$-Trie) index is no larger than k.
2. The number of entries in an $\mathcal{N}[k]$-Trie is no larger than $k \times |\mathcal{N}[k]\text{-partition}(X)|$.
3. The structure of a $\mathcal{P}[k]$-Trie of a document X is exactly the same as the structure of an $\mathcal{N}[k]$-Trie of X.
4. For a given XML document X, the $\mathcal{N}[k-1]$-Trie is a strict compression of the $\mathcal{N}[k]$-Trie. The $\mathcal{N}[k]$-Trie is a strict refinement of the $\mathcal{N}[k-1]$-Trie.
5. The $\mathcal{P}[k]$-Trie of X is a sub-structure of the $\mathcal{P}[k+1]$-Trie of X. To be more precise, the $\mathcal{P}[k]$-Trie of X is a sub-structure that contains the top k layers of the $\mathcal{P}[k+1]$-Trie of X.

5.1.4 Query Evaluation

Let's consider evaluating some typical $\mathcal{D}$ and $\mathcal{D}^{[]}$ queries in the sample XML document. The evaluation plans for evaluating these queries using the indexes discussed above are summarized in Table 4. Here, **I** represents an *index-only plan*, **I+V** represents *index access plus validation*, **MC/SC** represents *accessing multiple/single partition class(es) in the index*. When multiple partition classes are involved, **ML/SL** represents *multiple/single lookup*, and **RA/SA** represents *random/sequential access*.

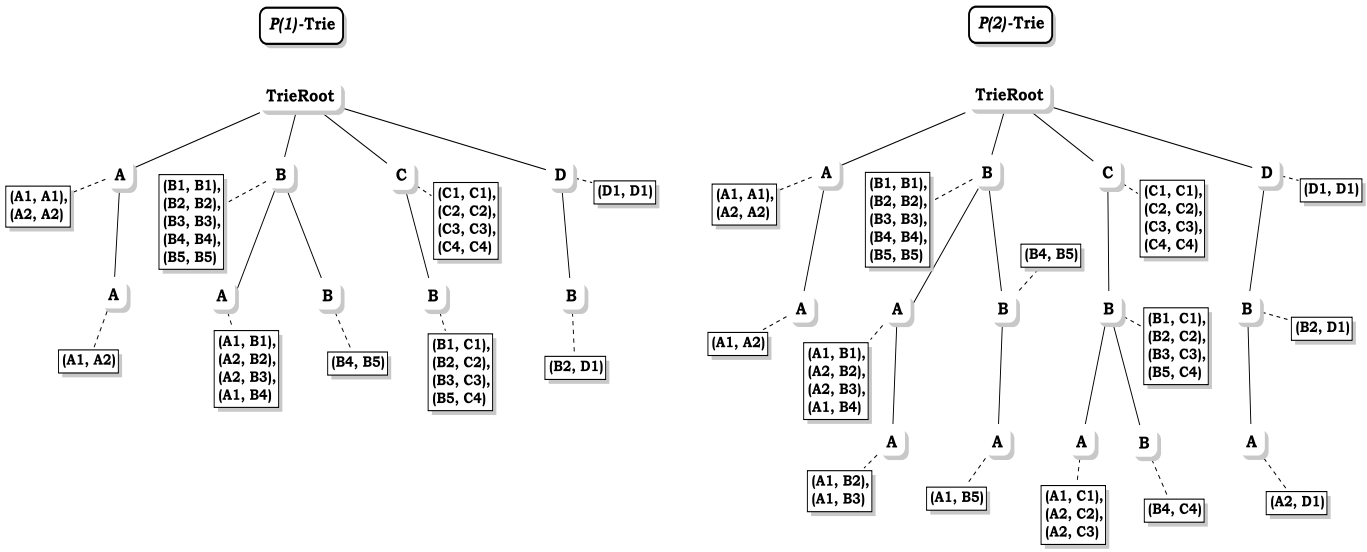

Figure 3: $\mathcal{P}[1]$ and $\mathcal{P}[2]$-Trie Index

Query	Query Class	DataGuides [10]	$\mathcal{A}[2]$-index [18]	$\mathcal{N}[2]$-Trie	$\mathcal{P}[2]$-Trie
$//A/B$	$\mathcal{D}[1]$	I, MC (ML,RA)	I, MC (ML, RA)	I, MC (SL, SA)	I, SC
$//A/B/B/D$	$\mathcal{D}[3]$	I, SC	I+V, SC	I+V, SC	I, SC
$//A/B[D]/C$	$\mathcal{D}^{[]}[2]$	I+V, SC	I+V, SC	I+V, SC	I, SC

Table 4: Evaluating expressions using structural indexes

The example illustrates that all queries of $\mathcal{D}^{[]}$ can be evaluated with index-only plans when the $\mathcal{P}[k]$-Trie index ($k > 1$) is available. Because the $\mathcal{P}[k]$-Trie contains partition information under path semantics, it is possible to decompose the original query into sub-queries that can be answered with an index-only plan, performing join on the partial results to obtain the answer to the original query. This can be applied to queries with length larger than k, or queries that contain predicates, '//' and '*' in the middle. How to best decompose a query is an interesting optimization question that deserves further research.

Clearly, all these facts point to a trade-off of space and query efficiency between the $\mathcal{N}[k]$-Trie and $\mathcal{P}[k]$-Trie indexes that deserve further study.

5.1.5 Trie Index Implementation

We are implementing our Trie indexes using Timber [15], a native XML database system developed at the University of Michigan. Using Timber has allowed the focus of the implementation to remain on index construction and query processing, as Timber already provides the necessary framework for storing and querying XML documents.

Our algorithms for partition and index creation were integrated into Timber's Index Manager module; while the Query Evaluator module was modified so that we may perform queries and extract information from our indexes.

We have created Trie indexes for the DBLP bibliography XML file (130MB in size), and are executing different types of queries to evaluate performance. Each query is evaluated using: no index, Timber's default element tag index, the $\mathcal{A}[k]$-index, the $\mathcal{N}[k]$-Trie index, and the $\mathcal{P}[k]$-Trie index, with $k = 0 \ldots 7$ for the former three.

Preliminary results show that the performance of both $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie indexes present an improvement over the $\mathcal{A}[k]$-index, as well as showing important differences among themselves.

We are also developing update algorithms for the Trie indexes. The structural properties identified in Section 5.1.3 will allow us to develop simple refinement and compression algorithms that will easily convert a k Trie to a $k+1$ Trie or vice-versa.

Regarding Trie index maintenance, if the source XML document is modified, then the Trie indexes will face the same challenges as any label-path based index. Previous solutions such as [17, 35] have already been proposed, and we must investigate which approach most benefits our index structure.

5.2 Workload-aware Structural Summaries

The $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie indexes proposed provide a dramatical performance improvement for evaluating XPath queries. Another useful feature of these indexes is that the trie branches are independent from one another, allowing us to construct and maintain indexes in which the local bi-similarity is different for label-paths with different importance. This makes the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie family a suitable platform for the design of workload-aware indexes for XML.

We will investigate how to choose the optimal bi-similarity for the label paths, given a database, a query workload, and an upper bound for space. We will also study how to represent the degree of bi-similarity in the index to minimize lookup and validation in query evaluation, and how to maintain such indexes dynamically when there are data or query workload changes.

5.3 Statistical Summaries for XML

The query decomposition and optimization techniques discussed above are within the scope of cost-based query optimization. Cost-based optimization strategies depend on an accurate cost model and cardinality estimates to perform well. The Trie structure we proposed provides a suitable framework for collecting and presenting statistical information about an XML document. Since neighboring sub-trees in a trie are independent from each other, statistical information can be specific to each path.

We will investigate how statistical information can be collected and stored at various levels of granularity, how to maintain the statistical summary when data changes, and how to accurately estimate the cardinality of path expressions, in both node and path semantics, leveraging the coupling theory we have discovered.

6. CONCLUSION

Currently there are two leading trends in the design of XML repositories: reuse existing relational database technology and adapt it for use with XML or implement native XML database management systems. Efficient query evaluation is at the core of database system design, whether in a relational or XML context. We propose to draw from the strength of successful research and practice in relational databases and apply the same methodology to the study of the XPath query language and the design of summary structures for efficient XML query processing. The research methodology, preliminary results, and future research directions we presented here are all independent of the storage model and can be easily adopted in both.

We presented the preliminary results of our work that find a coupling between fragments of the XPath algebra and a structural partition of XML documents. These results have guided the design of a new structural index family, the Trie indexes. The methodology we have used in the design and study of the Trie indexes makes a first attempt at bridging the gap between the theoretical and engineering work in the context of XML structural indexes.

We outlined research directions that will continue to take advantage of this methodology in the design of structural indexes, workload-aware structural indexes, and statistical summary structures, along with their corresponding access and maintenance algorithms.

7. ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Yuqing Melanie Wu and Prof. Dirk Van Gucht for their continued support and guidance in my research. I would also like to thank my fellow student Pablo Santa Cruz for his contribution in the implementation of the trie indexes. Finally, I thank the reviewers and workshop attendees for their comments, which have helped me improve the quality of the paper.

8. WORKSHOP FEEDBACK

The feedback received at the EDBT 2008 Ph.D. workshop was very useful for our research by encouraging the general direction we have taken, and providing general suggestions.

Reviewers pointed out the importance of considering Trie index maintenance. As pointed out in Section 5.1.5, this is an area where we can benefit from existing work, and which we are currently focusing on.

With respect to our future work in the Statistical Summaries, we received questions about the types of statistics that we intend to calculate and use. Since we believe our $\mathcal{P}[k]$-Trie index is very efficient in query decomposition, evaluation, and the sub-sequent join of the sub-query results, we will start by keeping simple cardinality statistics that can help us select the best query evaluation plan with respect to the selectivity of the join operators.

An important observation made was that even if we are currently implementing our Trie indexes in Timber, our approach is not limited to a native XML database. Therefore, the Trie indexes can be potentially implemented in other platforms, where their efficiency can be further evaluated.

Finally, we would like to thank the organizers of the workshop for their effort and the Fondation Michel Métivier for their support.

9. REFERENCES

[1] S. Abiteboul. Querying Semi-Structured Data. In *ICDT*, 1997.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, 2002.

[3] M. Benedikt, W. Fan, and F. Geerts. XPath Satisfiability in the Presence of DTDs. In *PODS*, 2005.

[4] M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. In *ICDT*, 2003.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD*, 2002.

[6] Q. Chen, A. Lim, and K. W. Ong. D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data. In *SIGMOD*, 2003.

[7] C.-W. Chung, J.-K. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. In *SIGMOD*, 2002.

[8] M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *ICDE*, 1998.

[9] G. H. L. Fletcher, D. V. Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A Methodology for Coupling Fragments of XPath with Structural Indexes for XML Documents. In *DBPL*, 2007.

[10] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructures Databases. In *VLDB*, 1997.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, 2005.

[12] G. Gou and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10), 2007.

[13] M. Gyssens, J. Paredaens, D. V. Gucht, and G. H. Fletcher. Structural Characterizations of the Semantics of XPath as Navigation Tool on a Document. In *PODS*, 2006.

[14] H. He and J. Yang. Multiresolution Indexing of XML for Frequent Queries. In *ICDE*, 2004.

[15] H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel,

D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The International Journal on Very Large Data Bases*, 11:274–291, 2004.

[16] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Queries. In *SIGMOD*, 2002.

[17] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for Structure Indexes. In *VLDB*, 2002.

[18] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *ICDE*, 2002.

[19] C. Koch. Processing Queries on Tree-Structured Data Efficiently. In *PODS*, 2006.

[20] M. Marx and M. de Rijke. Semantic Characterizations of Navigational XPath. *SIGMOD Record*, 34(2), 2005.

[21] G. Miklau and D. Suciu. Containment and Equivalence for a Fragment of XPath. *Journal of the ACM*, 51(1), 2004.

[22] T. Milo and D. Suciu. Index Structures for Path Expressions. In *ICDT*, 1999.

[23] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT Workshops*, 2002.

[24] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD Conference*, 2004.

[25] N. Polyzotis and M. Garofalakis. XSketch Synopses for XML Data Graphs. *ACM Transactions on Database Systems*, 31(3), 2006.

[26] P. Rao and B. Moon. PRIX: Indexing And Querying XML Using Prüfer Sequences. In *ICDE*, 2004.

[27] K. Runapongsa, J. M. Patel, R. Bordawekar, and S. Padmanabhan. XIST: An XML Index Selection Tool. In *XSym*, 2004.

[28] W3C Consortium. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20, 2007.

[29] W3C Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, 2007.

[30] W3C Consortium. XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/, 2007.

[31] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *SIGMOD*, 2003.

[32] W. Wang, H. Jiang, H. Wang, X. Lin, H. Lu, and J. Li. Efficient Processing of XML Path Queries Using the Disk-Based F&B Index. In *VLDB*, 2005.

[33] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *ICDE*, 2003.

[34] Y. Wu, J. M. Patel, and H. V. Jagadish. Using Histograms to Estimate Answer Sizes for XML Queries. *Information Systems*, 28(1-2), 2003.

[35] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental Maintenance of XML Structural Indexes. In *SIGMOD Conference*, 2004.