Trie Indexes for Efficient XML Query Evaluation

Sofía Brenes, Yuqing Wu, Dirk Van Gucht, Pablo Santa Cruz
Indiana University, Bloomington
{sbrenesb, yuqwu, vgucht, psantacr}@cs.indiana.edu

ABSTRACT

As the number of applications that rely on XML data increases, so does the need for performing efficient XML query evaluation. A critical part of the solution involves providing new techniques for designing XML indexes and lookup algorithms. In this paper, we leverage the results of our research on coupling the partitions induced by fragments of XPath algebra and those induced by the structural properties of an XML document to lead the design of the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie indexes for XML. We present the rationale behind our approach, and detail the structure of the indexes, their features, and algorithms for evaluating XPath queries with index-only plans. We show that the $\mathcal{P}[k]$-Trie indexes are capable of answering XPath queries with arbitrarily complicated predicates using index-only plans and improving query performance in orders of magnitude over the $\mathcal{A}[k]$-index.

1. INTRODUCTION

As an increasing number of applications use XML as their data model, supporting efficient access to XML data continues to be one of the most important research problems in this domain. XPath is a query language to specify node sets in XML documents. It is a fundamental building block for other more powerful XML query languages such as XQuery and XSLT. Therefore, efficiently evaluating XPath queries is essential to enable fast processing of general XML queries.

The complexity of both XML documents and XPath query patterns pose significant challenges to supporting efficient evaluation of such queries using indexes. The structural relationships present in both XML data and XML queries require the indexes to effectively summarize such structural information and to support fast lookup. Therefore, a critical part of the solution involves proposing new techniques for XML indexing and lookup; as well as the query evaluation techniques and algorithms that take advantage of these indexes. Moreover, as in relational database systems, without workload analysis, it is difficult to predict the queries issued by users. The flexible environments in which XML data is mostly used, such as web databases, contribute to this unpredictability. Therefore, even though it is important to construct and maintain indexes that are designed to answer frequent queries of a certain type, it is critical to design and implement generic indexes that help evaluate the core of XPath queries in general.

EXAMPLE 1.1. *Given a simple XML document, whose tree representation is shown in Figure 1,* [1] *we are interested in answering*

1. *Queries of arbitrary length, such as //B, //A/A/B/D.*
2. *Queries with uncertainty introduced by a wildcard '*' or '//', such as //B/*/C, //A//D.*
3. *Queries with predicates, such as //A/B[D]/C.*
4. *Any combination of the above.*

1.1 Previous Work

There have been significant engineering efforts in developing novel index structures to improve the performance of XPath query evaluation [7]. Early approaches combined the use of traditional value indexes and structural join algorithms [1, 2]. These present a simple solution to the problem but cannot directly capture the structural relationships at the heart of both XML data and queries.

Among structural indexes, DataGuides [5] construct a complete summary of every path in an XML document; the 1-index, 2-index, and T-index [15] use templates to define the paths that are to be summarized. These early structural indexes are too large for practical use. As a remedy, the $\mathcal{A}[k]$-index [14] employed the notion of localized bi-similarity in grouping the nodes of an XML document into equivalence classes and organizing these in a directed graph. Since then, several works have proposed variations of the $\mathcal{A}[k]$-index that deal mainly with its maintenance and tuning by using query workload information [3, 9]. The Forward and Backward index (F&B-index) [12] extends the local similarity concept to include both the incoming and outgoing paths of nodes, with the purpose of answering branching queries. However, the F&B-index was found to be too large to fit in memory and did not provide fully optimized methods for answering queries [17].

Other directions of XML indexing techniques include indexing frequent sub-patterns, indexing XML queries as sequences, encoding-based indexes, and index selection tools. Most of these indexes focus on answering chain queries without predicates [7].

[1] Single characters represent element tags and subscripts identify instances of elements with the same tag.

Proceedings of the 11th International Workshop on Web and Databases (WebDB 2008), June 13, 2008, Vancouver, Canada

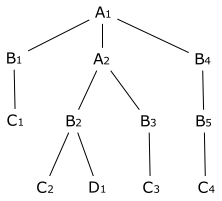

Figure 1: An Example XML Document

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

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

1.2 Our Contributions

We approach the problem of designing generic indexes for XML by leveraging the results of our previous studies [4] of coupling partitions induced by fragments of the XPath algebra and partitions induced by the structure of XML documents. More specifically, we

1. Propose a family of structural indexes for XML that use a trie structure to organize the partitions induced by node ($\mathcal{N}[k]$) and pair ($\mathcal{P}[k]$) equivalence relations.
2. Define efficient construction and lookup algorithms for the Trie indexes and identify a set of strategies for using the Trie indexes in XPath query evaluation.
3. Discuss the impact of the $\mathcal{P}[k]$-Trie index on query decomposition and generation of query evaluation plans.
4. Conduct extensive experiments to illustrate the properties of the Trie indexes and their efficiency in query evaluation.

The rest of the paper is organized as follows: we lay out the theoretical foundation of our ideas in Section 2, present the design of the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie indexes in Section 3, present the implementation details and experimental results in Section 4, and conclude the paper with a discussion of the results and the future research directions in Section 5.

2. PRELIMINARIES

In this section we will define the concepts that will facilitate the discussion throughout the rest of this paper.

2.1 Label-Path Based Partitions of an XML Document

We first define an XML document and its associated paths.

DEFINITION 2.1. *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, $Ed \subseteq V \times V$ the set of parent-child edges, $r \in V$ the root, and $\lambda\colon V \to \mathcal{L}$ a node-labeling function into the set of labels $\mathcal{L}$.*

Given an XML document X, we define the set of its paths, denoted $Paths(X)$, as the set $V \times V$. A pair of nodes $(m, n) \in Paths(X)$ identifies the unique path from node m to node n in X. The set of downward-paths, $DownPaths(X)$, contains the pairs of nodes (m, n) where m is an ancestor of n. Furthermore, for $k \in \mathbb{N}$, $DownPaths(X, k)$ represents the set of node pairs such that (1) $length(m, n) \leq k$, and (2) $(m, n) \in DownPaths(X)$.[2]

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

DEFINITION 2.2. *The label-path $LP(m, n)$ is the unique sequence of labels that occur on the unique path from node m to node n. Given a node $n \in V$, 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 $l = min(height(n), k)$.*[3]

In the XML document shown in Figure 1, the label-path between nodes A_2 and C_2 is $LP(A_2, C_2) = (A, B, C)$. Similarly, $LP(C_1, 1) = (B, C)$ and $LP(C_1, 5) = (A, B, C)$.

DEFINITION 2.3. *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 $\mathcal{N}[k][p]$.

DEFINITION 2.4. *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 $DownPaths(X, k)$ as follows: two pairs (m_1, n_1) and $(m_2, n_2) \in DownPaths(X, k)$ 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 $DownPaths(X, k)$ 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* and a k-label-path p in an XML document X uniquely identifies a *$\mathcal{P}[k]$-partition* class, denoted $\mathcal{P}[k][p]$.

EXAMPLE 2.1. *Table 2 shows the $\mathcal{N}[2]$ and $\mathcal{P}[2]$-partitions of the sample XML document shown in Figure 1.*

DEFINITION 2.5. *Given a data model M and two equivalence relations $\equiv_{\mathcal{R}_1}$ and $\equiv_{\mathcal{R}_2}$, we say that $\mathcal{R}_1$ refines $\mathcal{R}_2$ (denoted $\mathcal{R}_1 \prec \mathcal{R}_2$) if for each data instance m over M, the equivalence relation $\equiv_{\mathcal{R}_1}$ is a refinement of the equivalence relation $\equiv_{\mathcal{R}_2}$. We say $\mathcal{R}_1$ and $\mathcal{R}_2$ are equally refined (denoted $\mathcal{R}_1 \simeq \mathcal{R}_2$) if $\mathcal{R}_1 \prec \mathcal{R}_2$ and $\mathcal{R}_2 \prec \mathcal{R}_1$.*

For an XML document X, $\mathcal{N}[k] \prec \mathcal{N}[k-1]$. For any $k \in \mathbb{N}$ where $k \geq height(X)$, we have that $\mathcal{N}[k] \simeq \mathcal{N}[k+1]$ and $\mathcal{P}[k] \simeq \mathcal{P}[k+1]$. Furthermore, we proved that $\mathcal{N}[k] \simeq \mathcal{A}[k]$ [4], where $\mathcal{A}[k]$ is the node partition as defined in [14].

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

2.2 The $\mathcal{D}[k]$ and $\mathcal{U}[k]$ XPath Algebras

We now present the syntax and *path semantics* of the XPath algebra, which were introduced earlier in [6, 8].

DEFINITION 2.6. *The XPath algebra consists of the primitives $\varepsilon, \emptyset, \downarrow$, and ℓ together with the operations on expressions $E_1 \circ E_2$, $E_1[E_2]$, and $E_1 * E_2$. Given an XML document $X = (V, Ed, r, \lambda)$, the path semantics of an XPath algebra expression E on X, denoted $E(X)$, are defined as:*

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

The node semantics of an XPath algebra expression E on X, denoted $E^{nodes}(X)$, is the set $\{n \mid \exists m : (m,n) \in E(X)\}$.

DEFINITION 2.7. *The $\mathcal{D}$ (downwards) algebra consists of the expressions in the XPath algebra without occurrences of the set operators, predicates ($[]$), or the $\uparrow$ primitive. As in [4], the $\mathcal{D}[k]$ algebras are recursively defined on k as follows:*

1. *$\mathcal{D}[0]$ is the set of expressions in $\mathcal{D}$ 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 $E_1 \circ E_2 \in \mathcal{D}[k]$.*

Given an XML document $X = (V, Ed, r, \lambda)$, and an expression $E \in \mathcal{D}[k]$, we define the label-path set that corresponds to E in X, $LPS(E, X)$ as the set of label-paths in X that satisfy the node-labels and structural containment relationships specified by E. For the sample XML document in Figure 1 and the XPath expression $//A/*/B$ the corresponding $LPS(E, X) = \{(A, A, B), (A, B, B)\}$.

The upwards algebras $\mathcal{U}$ and $\mathcal{U}[k]$ are defined in the same way as the downwards algebras, but feature the $\uparrow$ primitive instead of $\downarrow$.

DEFINITION 2.8. *Let $X = (V, Ed, r, \lambda)$ be an XML document, and $k \in \mathbb{N}$. We say two paths (m_1, n_1) and $(m_2, n_2) \in DownPaths(X, k)$ are $\mathcal{D}[k]$-equivalent, denoted $(m_1, n_1) \equiv_{\mathcal{D}[k]} (m_2, n_2)$, if for any expression E in $\mathcal{D}[k]$ it is the case that $(m_1, n_1) \in E(X)$ if and only if $(m_2, n_2) \in E(X)$.*

The $\mathcal{D}[k]$-partition of X is then defined as the partition on $DownPaths(X, k)$ induced by this equivalence relation. It is straightforward to observe that the partitions associated with the $\mathcal{U}[k]$ algebra are the partitions of the inverted node pairs associated with the $\mathcal{D}[k]$ algebra.

2.3 Coupling Label-Path and Algebra Based Partitions of an XML Document

The following propositions [4] provide the formal work on which our index structure design and query evaluation algorithms are based.

PROPOSITION 2.1. *Let X be an XML document and $k \in \mathbb{N}$. The $\mathcal{P}[k]$-partition of X and the $\mathcal{D}[k]$-partition of X are the same.*

Recall that in both $\mathcal{N}[k]$ and $\mathcal{P}[k]$-partitions a label-path uniquely identifies its corresponding class of nodes (or node pairs). Therefore, we can evaluate expressions in $\mathcal{D}[k]$ via unions of the corresponding partition classes.

PROPOSITION 2.2. *Let X be an XML document and E an expression in $\mathcal{D}[k]$. Then,*

$$\begin{aligned}
E(X) &= \bigcup_{lp \in LPS(E,X)} \mathcal{P}[k][lp] \\
E^{nodes}(X) &= \bigcup_{lp \in LPS(E,X)} \mathcal{N}[k][lp]
\end{aligned}$$

Propositions 2.1 and 2.2 establish an important principle in query evaluation: *any $\mathcal{D}[k]$ (and therefore $\mathcal{U}[k]$) expression can be evaluated by unions of $\mathcal{P}[k]$-partition blocks under path semantics and $\mathcal{N}[k]$-partition blocks under node semantics.* This leads to the following design conclusion: *indexes based on the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-partitions are suitable for answering queries in $\mathcal{D}[k]$ and $\mathcal{U}[k]$.*

2.4 Evaluating XPath Expressions with Decomposition

Query expressions in $\mathcal{D}[k]$ or $\mathcal{U}[k]$ are important components of XPath queries. However, the majority of query requests are expressed in twig patterns, with arbitrarily complicated predicates. The $\mathcal{D}^{[]}[k]$ algebra is defined as the $\mathcal{D}[k]$ algebra, substituting $\mathcal{D}[k]$ with $\mathcal{D}^{[]}[k]$ and extending clause *(c)* to include $E_1[E_2] \in \mathcal{D}^{[]}[k]$.

DEFINITION 2.9. *Given a list of expressions $E_1, \ldots, E_m$ such that each E_i is an expression either in $\mathcal{D}$ ($\mathcal{D}[k]$) or $\mathcal{U}$ ($\mathcal{U}[k]$), then the expression $E_1 \circ \ldots \circ E_m$ is in the DownUp (DownUp[k]) algebra.*

PROPOSITION 2.3. *For each expression $E \in \mathcal{D}^{[]}[k]$ there exists an equivalent expression F_E in DownUp[k].*

It follows that each expression in $\mathcal{D}^{[]}$ can be translated into an equivalent *DownUp* expression, since $E_1[E_2](X) = E_1(X) \circ E_2(X) \circ (E_2(X))^{-1}$.

3. TRIE INDEXES

The propositions of Section 2.3 imply that (1) the $\mathcal{D}[k]$ and $\mathcal{U}[k]$ algebras are invertible to each other under the path semantics: a partition block $B \in \mathcal{D}[k]$ iff $B^{-1} \in \mathcal{U}[k]$; (2) the $\mathcal{D}[k]$-partition and the $\mathcal{P}[k]$-partition are the same; (3) each partition block is identified by a unique label-path; (4) the answer to each $\mathcal{D}[k]$ or $\mathcal{U}[k]$ expression is the union of the corresponding partition blocks.

To take full advantage of the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-partitions and their block labeling expressions, we need a data structure: (1) that is capable of locating all partition blocks using label-path lookups; and (2) in which the partition blocks that participate in the union operation that answer a query are stored close to each other and can be located with a minimum number of label-path lookups.

A trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are strings. Strings in a sub-tree share a common prefix which is represented by the incoming path to the root of the subtree, making the trie structure a natural choice to organize the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-partitions. Since all label-paths required to answer an expression share a common suffix, we use the reversed label-path of each partition block as the trie key.

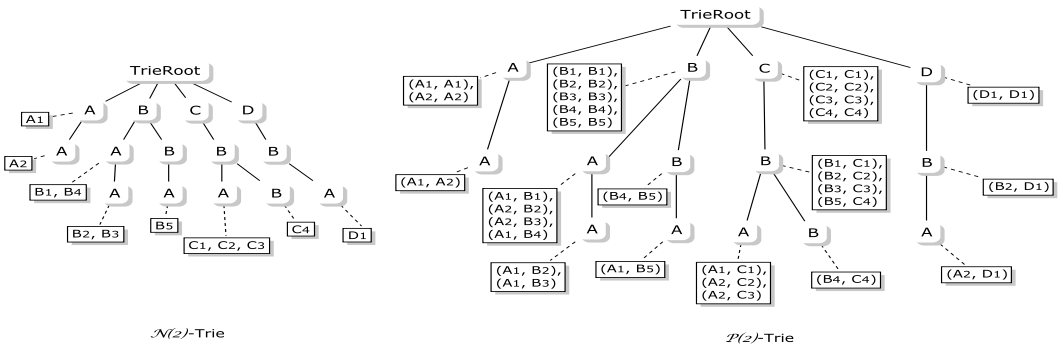

Figure 3: Sample $\mathcal{N}[2]$ and $\mathcal{P}[2]$-Trie Indexes for the XML source document in Figure 1

3.1 $\mathcal{N}[k]$-Trie Index

In an $\mathcal{N}[k]$-Trie we organize the $\mathcal{N}[k]$-partition blocks in a trie to facilitate quick lookup of groups of partition blocks.

DEFINITION 3.1. *Given an XML document X and an integer k, the $\mathcal{N}[k]$-Trie of X is a trie structure where the index keys are the inverted label-paths of the $\mathcal{N}[k]$-partition blocks. The index entry of each label-path p is equal to the contents of the corresponding $\mathcal{N}[k][p]$.*

Figure 3 shows a sample $\mathcal{N}[2]$-Trie. Given an XML document X, the properties of its $\mathcal{N}[k]$-Trie are:

1. The height of an $\mathcal{N}[k]$-Trie is no larger than k.
2. The number of nodes in an $\mathcal{N}[k]$-Trie is no larger than $|\mathcal{N}[k]\text{-partition}(X)$—.
3. Each leaf node (but only some non-leaf nodes) in an $\mathcal{N}[k]$-Trie has an associated $\mathcal{N}[k]$-partition block.
4. The $\mathcal{N}[k-1]$-Trie is a compression of the $\mathcal{N}[k]$-Trie.

3.1.1 Index Lookup

We define two types of lookup in an $\mathcal{N}[k]$-Trie of an XML document X: direct lookup and sub-tree lookup.

DEFINITION 3.2. *Given an $\mathcal{N}[k]$-Trie T and a reversed label-path p, the direct lookup of p in T is defined as $T[p] = \mathcal{N}[k][p]$. The sub-tree lookup of p in T is defined as $T_{st}[p] = \bigcup_{p'} \mathcal{N}[k][p']$ where p is the prefix of p'.*

A direct lookup will retrieve at most one partition block from an $\mathcal{N}[k]$-Trie. The lookup may return no results if no string key matches p in the $\mathcal{N}[k]$-Trie, or if the matching string key has no partition block associated with it. A sub-tree lookup will return the union of the partition blocks in the sub-tree rooted at the node that matches p. As stated in propositions 2.1 and 2.2, the result is precisely the answer to p against X, under node semantics.

3.1.2 Query Evaluation

Using the direct and sub-tree lookup operations, an $\mathcal{N}[k]$-Trie can be used to efficiently answer any $\mathcal{U}[k]$ queries under node semantics with only one index lookup. Given an XML document X and an $\mathcal{N}[k]$-Trie T of X, the node semantics answer to a query $p \in \mathcal{U}[k]$ and $q \in \mathcal{D}[k]$ are:

$$p(X) = \begin{cases} T[p] & \text{if } |p| = k \\ T_{st}[p] & \text{if } |p| < k \end{cases} \qquad q(X) = \begin{cases} T[q^{-1}] & \text{if } |q| = k \\ T_{st}[q^{-1}] & \text{if } |q| < k \end{cases}$$

3.1.3 Limitations of the $\mathcal{N}[k]$-Trie

Even though the $\mathcal{N}[k]$-Trie provides an efficient solution for evaluating queries of length at most k using a single index lookup (an improvement over the $\mathcal{A}[k]$-index), validation (accessing the source data) is still required in case the query has a predicate, a wildcard '*', or '//' in the middle, or if the length is larger than k. For instance, among the motivating examples, an $\mathcal{N}[2]$-Trie can efficiently answer query $//B$. However, for queries $//A/A/B/D$ and $//B[D]/C$, validation is required even with the help of the $\mathcal{N}[2]$-Trie.

3.2 $\mathcal{P}[k]$-Trie Index

We now introduce the $\mathcal{P}[k]$-Trie, which organizes the partition blocks of a $\mathcal{P}[k]$-partition in a trie structure.

DEFINITION 3.3. *Given an XML document X and an integer k, the $\mathcal{P}[k]$-Trie of X is a trie structure where the index keys are the inverted label-paths of the $\mathcal{P}[k]$-partition blocks. The index entry of each label-path p is equal to the contents of the corresponding $\mathcal{P}[k][p]$.*

Figure 3 shows a sample $\mathcal{P}[2]$-Trie. Notice the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie share the same structure but differ in the content of their index entries. For a given XML document X, the properties of its $\mathcal{P}[k]$-Trie are:

1. The height of a $\mathcal{P}[k]$-Trie is no larger than k.
2. The structure of a $\mathcal{P}[k]$-Trie of X is identical to the structure of an $\mathcal{N}[k]$-Trie of X.
3. The number of nodes in a $\mathcal{P}[k]$-Trie is no larger than $k \times |\mathcal{P}[k]\text{-partition}(X)$—.
4. All nodes in a $\mathcal{P}[k]$-Trie have associated partition blocks.
5. The $\mathcal{P}[k-1]$-Trie of X is the sub-structure of the $\mathcal{P}[k]$-Trie of X, and contains its top k layers.

3.2.1 Index Lookup

In a $\mathcal{P}[k]$-Trie we only need to define a direct lookup operation.

DEFINITION 3.4. *Given a $\mathcal{P}[k]$-Trie T and a reversed label-path p, the direct lookup of p in T (denoted $T[p]$) is defined as $T[p] = \mathcal{P}[k][p]$.*

A direct lookup will retrieve at most one partition block from a $\mathcal{P}[k]$-Trie. The lookup will return no results if no string key matches p in the $\mathcal{P}[k]$-Trie.

3.2.2 Query Evaluation

Using the direct lookup operation and query decomposition, the $\mathcal{P}[k]$-Trie can efficiently answer a larger number of query types. Given an XML document X and a $\mathcal{P}[k]$-Trie T of X, the path semantics answer to a query $q \in \mathcal{U}[k]$ is $q(X) = T[q]$ for $|q| \leq k$. If we have a query $q \in \mathcal{D}[k]$ then the answer will be $q(X) = T[q^{-1}]$ for $|q| \leq k$.

Complicated XPath queries, such as queries with length larger than k, with '*' or '//' in the middle, and queries with predicates can all be decomposed into sub-queries in $\mathcal{D}[k]$ (as shown in Section 2.4). For each such type of query, the result of evaluating q in X under path semantics is:

- If $q = //t_1/t_2/\ldots/t_n$ where $n > k$, then $q(X) = t_1(X) \bowtie t_2(X) \bowtie \ldots \bowtie t_n(X)$, where $|t_i| \leq k$ and the tail token of t_i matches the head token of t_{i+1}.
- If $q = //q_1//q_2$, then $q(X) = q_1(X) \bowtie_{sj} q_2(X)$.[4]
- If $q = //t_1/\ldots/t_m[t_b]/t_{m+1}/\ldots/t_n$, then $q(X) = q_1(X) \bowtie \pi_1(q_2(X)) \bowtie q_3(X)$, where $q_1 = //t_1/\ldots/t_m$, $q_2 = //t_m/t_b$, and $q_3 = //t_m/t_{m+1}/\ldots/t_n$.

Other operations such as projection or join (including structural join) may be required to compute the final answer to the query. However, in all these cases access to the source data can be omitted and *no validation is necessary*. This enables us to claim that using a $\mathcal{P}[k]$-Trie we can efficiently evaluate *any* XPath query in $\mathcal{D}^{[]}[k]$ with an index-only plan.

When compared to the $\mathcal{N}[k]$-Trie, the $\mathcal{P}[k]$-Trie (with the same parameter k) clearly demands more storage space, as shown in Figure 3. However, the fact that a $\mathcal{P}[k]$-Trie can support the evaluation of XPath queries with index-only plans justifies such overhead. More importantly, whether a query can be evaluated using an index-only plan using a $\mathcal{P}[k]$-Trie, is not determined by the value of k as long as $k > 0$. If $k = 0$ then the information in the index is not enough to join different partition blocks based on their ancestor information.

The structural properties of the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie index make it a simple process to adjust the k value of these indexes, with simple refinement and compression algorithms that will easily convert a $k-1$ Trie to a k Trie or vice-versa. If a change is made to the source XML document, the Trie indexes face the same challenges as any label-path based index and previously proposed solutions [13, 18] apply.

4. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

4.1 Prototype Implementation

For our implementation, we are using Timber [11], a native XML database system developed at the University of Michigan. As previously mentioned in Section 1.1, the $\mathcal{A}[k]$-index and its variants have been widely studied [3, 9, 12, 17]. Additionally, if we consider the fact that the $\mathcal{A}[k]$ and $\mathcal{N}[k]$-partitions are the same, but their index structures are different, it is of particular importance to compare the performance of the Trie indexes against that of the $\mathcal{A}[k]$-index. Therefore, we have extended Timber to include the implementation of the $\mathcal{A}[k]$-index (as described in [14]) and the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Tries for fair comparison. We take advantage of Timber's Index Manager, which uses the GiST system [10] to store indexes.

4.2 Experimental Setup

For our experiments, we used the DBLP data set [16], which features 25 unique element tags, with a maximum depth of 6, and an average depth of 2.9. The same trends were observed in DBLP data sets of different scale. Due to space limitations, we report only the results obtained on DBLP100M (130MB raw data, 3.3M nodes). All experiments were conducted on a computer running Microsoft Windows XP, with an Intel Pentium 4 3.2GHz CPU, 2GB of available RAM, and Timber's default settings. Each test (both in construction and query evaluation) was executed five times. The results (except in the case of index size) present an average of the values obtained.

[4]Where $\bowtie_{sj}$ represents structural join.

4.3 Trie Index Construction

Construction of the indexes requires two steps: (1) generating the $\mathcal{A}[k]$, $\mathcal{N}[k]$ and $\mathcal{P}[k]$-partitions and (2) inserting the resulting partition blocks into the index structures. In the case of the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie indexes, the partition blocks are inserted into a trie structure. For the $\mathcal{A}[k]$-index a directed graph must be constructed, which is more complicated than the trie structure. The partition generation algorithm described in [14] iteratively constructs the partition classes, analyzing every node and computing each node's successors in each iteration. Our algorithm uses a depth-first approach that traverses each node only once, using a stack to keep track of the current node's ancestors up to the root, presenting a significant performance improvement. We have used our partition generation algorithm for constructing the $\mathcal{A}[k]$-index as well as the Trie indexes.

We constructed the Trie indexes as well as the $\mathcal{A}[k]$-index with $k = 0 \ldots 5$. The index sizes, in terms of the number of index items, are shown in Figure 4. Figure 5 shows index construction times.

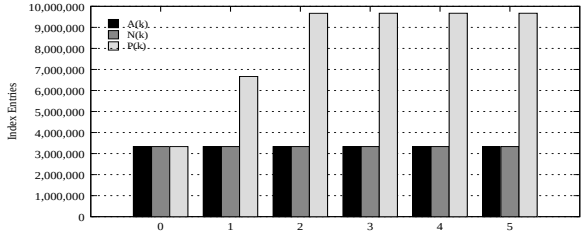

Figure 4: Index construction - Size

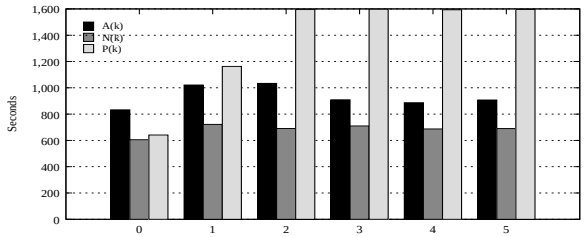

Figure 5: Index construction - Time

From the results, we can confirm that it is more expensive to store and construct a $\mathcal{P}[k]$-Trie index than an $\mathcal{N}[k]$-Trie or $\mathcal{A}[k]$-index. As expected, the size of a $\mathcal{P}[k]$-Trie increases linearly with the k value and is influenced by document structure, while the size of an $\mathcal{A}[k]$-index and an $\mathcal{N}[k]$-Trie depends only on the size of the XML document. The fact that the DBLP data is shallow, with an average depth of 2.9, contributes to the flattening of the size and construction time of the $\mathcal{P}[k]$-Trie index beyond $k = 3$, and also to a decrease in construction time for the $\mathcal{A}[k]$-index and the $\mathcal{N}[k]$-Trie, because of the reduced number of cases where a node's label-path needs to be shortened to match the value of k.

4.4 Efficiency in Query Evaluation

We now turn to query evaluation performance. We experimented with different types of queries to gauge the performance of the $\mathcal{N}[k]$ and $\mathcal{P}[k]$-Trie indexes versus the $\mathcal{A}[k]$-

index, but we were also interested in comparing the performance of the $\mathcal{N}[k]$ versus the $\mathcal{P}[k]$-Trie index. We evaluated ten different queries with varying characteristics. All exhibited the same behavior, and we report two illustrative results: a query with length 4 and a query with a nested predicate. Figures 6 and 7 show query evaluation times (in seconds) for the queries when using hot indexes.[5]

Queries are evaluated using Timber's physical plan interface. Since cost-based query optimization is not the focus of this paper, multiple physical plans were executed for each query and we report the results from the most efficient plans.

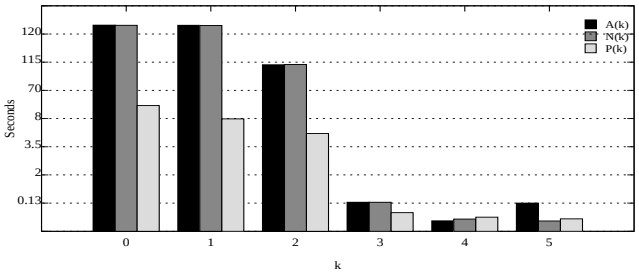

Figure 6: Query //dblp/inproceedings/title/i/sub

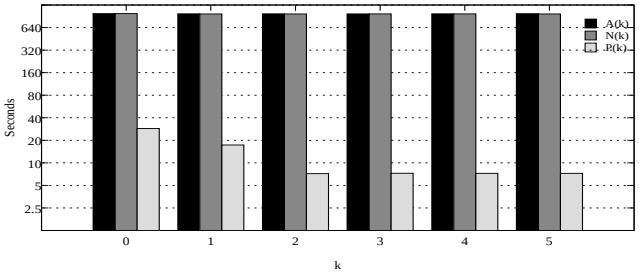

Figure 7: Query //dblp/inproceedings[title[i]/sub]/ee

The performance of the $\mathcal{N}[k]$-Trie and the $\mathcal{A}[k]$-index are comparable, except in the case where $k = 5$ in Figure 6, where the data structure used in the $\mathcal{N}[k]$-Trie results in more efficient evaluation of queries with length $< k$. The $\mathcal{P}[k]$-Trie consistently and significantly outperforms both, even for those with larger k values. This is due to the fact that index-only plans are enabled whenever a $\mathcal{P}[k]$-Trie ($k > 0$) is available, whether the query is longer than k or has a branch predicate, and validation is never needed.

The results also show that unlike the $\mathcal{A}[k]$-index and the $\mathcal{N}[k]$-Trie, the k value does not have such a dramatic influence in the performance of the $\mathcal{P}[k]$-Trie. This supports our argument that a modest k is sufficient for providing a significant performance improvement that justifies the overhead of constructing and maintaining a $\mathcal{P}[k]$-Trie.

We are currently designing further tests for the Trie indexes. We will follow the theoretical investigation of Trie index performance with tests over different types of data, such as XMark, and varying types of queries. These will include queries with '//' in the middle which can be efficiently evaluated in the $\mathcal{P}[k]$-Trie index by decomposing the query and combining the results of the sub-queries using structural join algorithms. We are also interested in comparing the Trie indexes with other approaches such as the D(k)-Index [3] and the F&B-index [12].

[5]A logarithmic scale is used to clearly show smaller performance times.

5. CONCLUSIONS

Leveraging the study that couples node and path partitions induced by an XML document and those induced by XPath algebras, we proposed novel structural indexes for XML which organize such partitions using a trie structure. In particular, we propose the $\mathcal{P}[k]$-Trie index which indexes node pairs that are k-bisimilar. Analytical and experimental results show that the $\mathcal{P}[k]$-Trie index enables XPath queries to be evaluated via index-only plans and provides query evaluation improvement by orders of magnitude when compared to the $\mathcal{A}[k]$-index. We continue to evaluate our indexes with varying data and query sets with different complexity and structural properties, attempting to provide a generic index structure that can perform efficient XML query evaluation for a large range of queries.

6. REFERENCES

[1] S. Al-Khalifa, *et al.* Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *ICDE*, 2002.

[2] N. Bruno, *et al.* Holistic Twig Joins: Optimal XML Pattern Matching. *SIGMOD*, 2002.

[3] Q. Chen, *et al.* D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data. *SIGMOD*, 2003.

[4] G. H. L. Fletcher, *et al.* A Methodology for Coupling Fragments of XPath with Structural Indexes for XML Documents. *DBPL*, 2007.

[5] R. Goldman, *et al.* DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *VLDB*, 1997.

[6] G. Gottlob, *et al.* Efficient Algorithms for Processing XPath Queries. *VLDB*, 2002.

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

[8] M. Gyssens, *et al.* Structural Characterizations of the Semantics of XPath as Navigation Tool on a Document. *PODS*, 2006.

[9] H. He, *et al.* Multiresolution Indexing of XML for Frequent Queries. *ICDE*, 2004.

[10] J. M. Hellerstein, *et al.* Generalized search trees for database systems. *VLDB*, pages 562–573, 1995.

[11] H. Jagadish, *et al.* TIMBER: A Native XML Database. *The International Journal on Very Large Data Bases*, 11:274–291, 2004.

[12] R. Kaushik, *et al.* Covering Indexes for Branching Path Queries. *SIGMOD*, 2002.

[13] R. Kaushik, *et al.* Updates for Structure Indexes. *VLDB*, 2002.

[14] R. Kaushik, *et al.* Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. *ICDE*, 2002.

[15] T. Milo, *et al.* Index Structures for Path Expressions. *ICDT*, 1999.

[16] Trier University. DBLP Data Set. http://dblp.uni-trier.de/xml, 2008.

[17] W. Wang, *et al.* Efficient Processing of XML Path Queries Using the Disk-Based F&B Index. *VLDB*, 2005.

[18] K. Yi, *et al.* Incremental Maintenance of XML Structural Indexes. *SIGMOD*, 2004.