Workload-aware Trie Indexes for XML

Sofía Brenes, Yuqing Wu, Hyungdae Yi
Indiana University, Bloomington
{sbrenesb, yuqwu, yih}@cs.indiana.edu

ABSTRACT

Well-designed indexes can dramatically improve query performance. In the context of XML, structural indexes have proven to be particularly effective in supporting efficient XPath queries - the core of all XML queries, by capturing the structural correlation between data components in an XML document. The duality of space and performance is an inevitable trade-off at the core of index design. It has been established that query workload can be leveraged to balance this trade-off and maximize the throughput of a group of queries. In this paper, we propose a family of novel workload-aware indexes by taking advantage of the recently proposed Trie indexes for XML. In particular, we propose the $\mathcal{WP}[k]$-Trie, the $\mathcal{AWP}[k]$-Trie, and the $\mathcal{W}[k]$-Trie indexes, which use the $\mathcal{P}[k]$-Trie framework to index frequent label-paths and a carefully selected complimentary set of label-paths. When a $\mathcal{WP}[k]$-Trie index is available, all frequent path queries are guaranteed to be evaluated in one index lookup, and all core XPath queries are guaranteed to be evaluated with index-only plans. With further consideration of the representativeness of label-paths in the index and proper annotations, the $\mathcal{AWP}[k]$ and $\mathcal{W}[k]$-Trie indexes are able to improve query evaluation performance by efficiently singling out queries with empty results and enabling more efficient query decompositions, with the $\mathcal{W}[k]$-Trie minimizing the space requirements of the index.

1. INTRODUCTION

With the explosive growth of data and search on the Internet, as well as growing demands in business, government, and science for managing a myriad of data, XML has emerged as the data format for representing, storing, and querying semi-structured data. As this trend is likely to continue, developing efficient query evaluation techniques for XML, especially XPath [18] queries which are the core of all XML queries, is critical.

Indexes have proved to be of significant importance in improving the query performance of XPath queries [8]. Specifically, structural indices, such as DataGuides [7], the 1-index [14] and the $A[k]$-index [13] were proposed to capture the structural correlation between data components, which is natural to the semi-structured data format of XML. The DataGuides [7] and the 1-index [14] are very fine in the way they partition the data. The size of these indexes can be as large as the data itself , which makes them less practical. For example, consider the sample XML document as shown in Figure 1, the corresponding strong DataGuide has as many nodes as the document itself. The $A[k]$-index [13] used a k parameter to control the degree of local bi-similarity of the partition and hence the size of the index, but sacrifices the ability to answer queries longer than k without accessing the data. In addition, all the indexes above require data validation for answering queries with branching predicates. The F&B-index [12] tried to address the branching predicate problem by indexing incoming and outgoing paths to a node, however, its huge space requirements limit its usefulness [19]. The $\mathcal{P}[k]$-Trie index [2] indexes $\mathcal{P}[k]$-partition classes of node pairs in a trie structure, using the reversed label-paths as keys. Path queries, long or short, with or without predicates, can be answered with index-only plans when the $\mathcal{P}[k]$-Trie index ($k \geq 1$) is available.

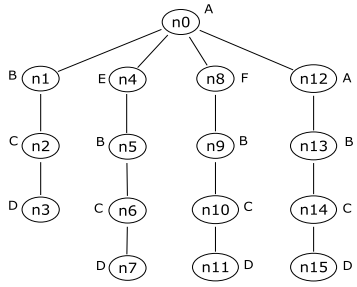

Figure 1: A Sample XML Document

Good index design requires a careful balance between the size of the index and the precision provided in query evaluation. It has been general wisdom [3, 9, 20] that not all query patterns are of equal importance: some are queried more frequently than others. Workload information can be leveraged to mitigate the trade-off between index size and precision, producing an index that yields better overall throughput, while maintaining the index footprint under control.

In previous research, several workload-aware indexes for XML have been proposed [5, 16, 4, 10]. The $D[k]$-index [4] and the $M[k]$-index [10] introduce variations to the $A[k]$-index that allow different degrees of local bi-similarity for differ-

ent paths, using workload information in selecting the bisimilarity value used for each path. These indexes share the same costly drawback of the $A[k]$-index, as validations are still required for branching queries, and for paths that are not indexed. Xist [16] provides an index selection tool that uses workload information to select a set of linear paths to index, but it also requires validation to answer paths that are not indexed. APEX [5] indexes frequent paths individually and groups the partition classes associated with all other paths that share the same suffix with the frequent paths into a *remainder* class. This approach does reduce the size of the index to a certain degree, but introduces difficulty in query evaluation when paths that are clustered in a *remainder* class have to be distinguished. Furthermore, even though it is claimed that APEX can answer all queries with an index-only plan, longer, non-frequent path queries have to be evaluated by performing multiple join operations, since only edges, rather than paths, are stored in the index.

It has been clearly established that creating indices based on workload greatly improves the performance of XPath queries. While some workload-aware indexes exist, there is room of improvement in the areas of space efficiency and query efficiency. The goal of our research is to design workload-aware indices for XML that are (1) superb for answering frequent path queries and efficient for answering non-frequent path queries; (2) efficient in identifying queries that yield empty results; (3) efficient in update induced by the changes in either the workload or the data itself; (4) efficient in size and adjustable easily to space allowance.

We take advantage of the recently proposed $\mathcal{P}[k]$-Trie index [2], and use its framework to index a frequent label-path set and a selected set of *complementary* label-paths. The strategies utilized in the selection of the *complementary* label-path set reflect the space/performance trade-off. The simplest strategy indexes label-paths up to length k; we call this index the $\mathcal{WP}[k]$-Trie. As seen in [2], fairly good performance can be achieved with a modest k value, with the additional benefit of answering frequent path queries in one index lookup. A more sophisticated strategy studies the structural correlation of a set of paths. We propose the notion of a label-path being represented by a set of label-paths at a structural and an instance-level. The $\mathcal{AWP}[k]$-Trie uses this strategy to extend the label-path set indexed by the $\mathcal{WP}[k]$-Trie, and adds annotations to its index entries to help in query optimization. The $\mathcal{AWP}[k]$-Trie assists the query optimizer in easily identifying longer sub-queries in the index and queries that yield empty results, contributing to the significant improvement in query performance, especially for non-frequent queries. We also propose a variant of the $\mathcal{AWP}[k]$-Trie index, called the $\mathcal{W}[k]$-index, to further reduce the index size and improve query performance, by considering statistical information about both frequent and non-frequent queries.

Our contributions can be summarized as follows:

1. We propose a family of workload-aware Trie indexes, the $\mathcal{WP}[k]$-Trie, $\mathcal{AWP}[k]$-Trie and $\mathcal{W}[k]$-Trie, for indexing frequent label-paths while maintaining efficient support for non-frequent queries;
2. We exploit the concept of structural and data instance representativeness of a set of paths with respect to an XML document, in annotating index entries and in guiding the selection of the complimentary label-path sets, to obtain the optimal space/performance trade-off;
3. We present the query evaluation and optimization options provided by the workload-aware Trie indexes;
4. We discuss the strategies for efficiently constructing the workload-aware Trie indexes, as well as incremental maintenance algorithms that handle changes in the workload, data, or index configuration;
5. We perform extensive experiments to compare the proposed indexing and query evaluation techniques with existing techniques in terms of the space footprint, query performance and maintenance overhead.

The rest of this paper is organized as follows: we present the terminologies used in this paper in Section 2, followed by a formal definition of the problem and an overview of the workload-aware index management and query processing system in Section 3. The family of workload-aware Trie indexes, along with their construction, evaluation, and maintenance algorithms is presented in Sections 4 and 5. Finally, in Section 6, we present implementation and evaluation details as well as the results of our experiments. We conclude with the discussion of future work in Section 7.

2. PRELIMINARIES AND PROBLEM DEFINITION

2.1 XML Document and Label-path Based Partitions

We treat an XML document $\mathcal{X}$ as a node-labeled tree. Formally, we define it as a 4-tuple $\mathcal{X} = (V, Ed, r, \lambda)$, 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 $\mathcal{X}$, we define the *downward-paths* of $\mathcal{X}$, denoted $DownPaths(\mathcal{X})$, as a set of node pairs (m, n) where m is an ancestor of n. Furthermore, given a number $k \in \mathbb{N}$, $DownPaths(\mathcal{X}, k)$ represents the set of node pairs such that (1) $length(m, n) \leq k$, and (2) $(m, n) \in DownPaths(\mathcal{X})$.[1] Given two nodes m and n in $\mathcal{X}$, we define the *label-path* $LP(m, n)$ as the unique sequence of labels $(\ell_m, \ldots, \ell_n)$ that occur on the unique path from node m to node n. Given a node $n \in V$, and a number $k \in \mathbb{N}$, the *k-label-path* of n, denoted $LP(n, k)$, is the label-path of the unique downward path of length l into n where $l = min(height(n), k)$.[2]

DEFINITION 2.1. *Let $\mathcal{X} = (V, Ed, r, \lambda)$ be an XML document, and let $k \in \mathbb{N}$. Two 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., $LP(n_1, k) = LP(n_2, k)$. Two node pairs (m_1, n_1) and $(m_2, n_2) \in DownPaths(\mathcal{X}, k)$ 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., $LP(m_1, n_1) = LP(m_2, n_2)$.*

[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.

The $\mathcal{P}[k]$-*partition* ($\mathcal{N}[k]$-*partition*) of $\mathcal{X}$ is the partition on the node pairs (nodes) of an XML document $\mathcal{X}$ induced by the $\mathcal{P}[k]$-equivalence ($\mathcal{N}[k]$-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 k-label-path of the nodes in C. On the other hand, a k-label-path $lp \in DownPaths(\mathcal{X}, k)$ uniquely identifies an $\mathcal{N}[k]$-*partition* class, which we denote as $\mathcal{N}[k][lp]$. 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 lp in an XML document $\mathcal{X}$ uniquely identifies a $\mathcal{P}[k]$-*partition* class, denoted $\mathcal{P}[k][lp]$.

2.2 Trie Index and Query Evaluation

XPath queries are the core of almost all XML query languages. The core XPath expressions that are frequently studied [1] can be defined in algebraic format as

$$E ::= \emptyset \,|\, \varepsilon \,|\, \hat{\ell} \,|\, \downarrow \,|\, \downarrow^* \,|\, E \circ E \,|\, E[E] | E \cup E$$

The downward path algebra $\mathcal{D}$, as studied in [2, 6], which contains only label matching and downward navigation is the simplest form of a path query. The expressions of $\mathcal{D}$ are

$$E ::= \emptyset \,|\, \varepsilon \,|\, \hat{\ell} \,|\, \downarrow \,|\, E \circ E$$

We can use a label-path to represent a $\mathcal{D}$ expression, by capturing the label matching ($\hat{\ell}$) at each step of the navigation. For example, query $q = A \circ \downarrow \circ B \downarrow \circ C$ (whose corresponding XPath query is //A/B/C) can be represented by label-path (A, B, C). In the rest of the paper, we will no longer distinguish the XPath representation, algebraic representation and label-path representation of a path query.

Given an XML document $\mathcal{X}$ and two path queries p_1 and p_2, we say that p_1 contains p_2, denoted as $p_1 \succeq p_2$, if $p_1(\mathcal{X}) \supseteq p_2(\mathcal{X})$. Using the label-path representation, it is obvious that $p_1 \sqsubseteq^s p_2 \Rightarrow p_1 \succeq p_2$ (e.g. p_1 is a suffix of p_2). We use the symbol $\sqsubset^s$ to represent the *proper suffix* relation between two label-paths. Please note that $p_1 \sqsubset^s p_2 \Rightarrow p_1 \succ p_2$ is not always true.

Path and node semantics have been defined for these algebras. In simple terms, path semantics evaluate a query into a set of node pairs that match the starting and ending token of a query, while node semantics evaluate into a set of nodes that match the tail of the query. We further define the $\mathcal{D}[k]$ expressions to be the $\mathcal{D}$ expressions with no more than k $\downarrow$'s. Studies in [2, 6] show that any query in the core XPath algebra can be decomposed into sub-queries in $\mathcal{D}[k]$, which can be answered efficiently when the $\mathcal{N}[k]$ or $\mathcal{P}[k]$-partitions are available, and be stitched together by natural and structural joins.

THEOREM 2.1. *Let $\mathcal{X}$ be an XML document and E an expression in $\mathcal{D}[k]$, Let $LPS(E, X)$ be the set of label-paths in $\mathcal{X}$ that satisfy the node-labels and structural containment relationships specified by E. Then,*

$$E(X) = \bigcup_{lp \in LPS(E,X)} \mathcal{P}[k][lp]$$

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

The $\mathcal{P}[k]$-Trie index [2] organizes the $\mathcal{P}[k]$-partition classes of an XML document in a trie structure, using the inverted label-path as keys [3]. In other words, a $\mathcal{P}[k]$-Trie index indexes all $DownPaths(\mathcal{X}, k)$ of a document $\mathcal{X}$. The extent of an index entry labeled by $lp \in DownPaths(\mathcal{X}, k)$ is the $\mathcal{P}[k]$-partition class associated with label-path lp and can be retrieved by a single trie lookup with key lp^{-1}.

Because the $\mathcal{P}[k]$-Trie indexes node pairs, any core XPath expression can be evaluated with an index-only plan, by decomposing the query into sub-queries that are $\mathcal{D}[k]$ expressions, performing index lookup for each sub-query, and computing the result via natural join, structural join and projection operations.

3. PROBLEM OVERVIEW

3.1 Problem Definition

The workload of a set of queries Q can be represented in various ways. We use a set of label-paths to represent the workload and define the frequent label-path set F_Q as a set of label-paths whose appearance in Q reaches a certain threshold. F_Q represents a set of $\mathcal{D}$ queries that are considered to be frequent, either by themselves, or as part of more complicated XPath queries. The task of analyzing Q and generating F_Q is usually done by a workload processor. While this is an interesting problem, it is not the focus of this paper. We will use the abbreviation F to represent F_Q.

The research problem we are addressing is: *given a document $\mathcal{X}$ and a workload F, design workload-aware indexes for $\mathcal{X}$ that support efficient evaluation of core XPath queries with a very small space and maintenance overhead.* To be more specific, we expect the workload-aware index to be (1) superb for answering path queries that contain the frequent label-path(s) and efficient for answering all core XPath queries; (2) efficient in identifying queries that yield empty results; (3) efficient in index updates induced by changes in either the workload or the data itself; (4) efficient in storage overhead and adjustable easily to space allowance.

3.2 System Overview

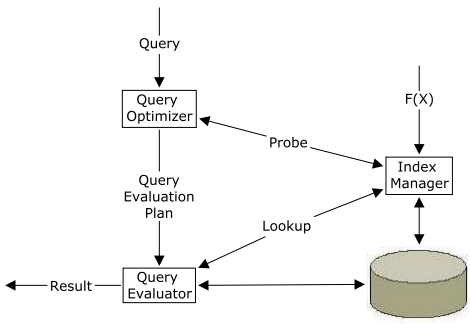

Figure 2: Workload-aware Index Management and Query Evaluation

Figure 2 shows the architecture of the information management and query processing module of an XML database management system that features workload-aware indexes. There are three major components that work together to

[3]We use lp^{-1} to represent the inverted path of label-path lp

evaluate queries: the index manager, query evaluator, and query optimizer. The index manager is at the center of this study. It uses the workload F to construct workload-aware indexes based on the XML documents stored in the database, and updates these indexes whenever the workload, the data, or the index configuration changes. It provides index access to the query evaluator through a *lookup* operation that takes a label-path as input and produces a set of node (node pair) identifiers. It also provides index availability information to the query optimizer through a *probe* operation, allowing the query optimizer to take full advantage of the indexes and generate an optimal evaluation plan.

In our family of workload-aware indexes, we index all label-paths in F plus a selected set of *complementary* label-paths. The strategies utilized in the selection of the *complementary* label-path set reflect the space/performance trade-off that is at the core of index design.

4. WORKLOAD ENHANCED $\mathcal{P}[k]$-TRIE

4.1 Extending the $\mathcal{P}[k]$-Trie with Workload

The $\mathcal{P}[k]$-Trie index [2] significantly out-performs the $A[k]$-index [13] by enabling index-only evaluation plans for all path queries. However, it also suffers from the same space / performance dilemma as the $A[k]$-index: the degree of local bi-similarity of all index entries is limited by the k parameter. This problem is especially evident when most frequent label-paths are short save for a few exceptions. Therefore, choosing a small k value results in a smaller index, but at the expense of not being able to answer some frequent label-path queries efficiently; choosing a large k value results in a much larger index, where most index entries are rarely used.

To get the best of both worlds, we take advantage of two important properties of the $\mathcal{P}[k]$-Trie index: (1) the independence of the trie branches in terms of the degree of local bi-similarity; (2) the fact that the $\mathcal{P}[k]$-Trie index supports index-only evaluation plans for all core XPath queries and performs reasonably well with a modest k value, as long as $k \geq 1$. Thus, we propose to index all frequent label-paths, and all paths of length $\leq k$ (with a rather small k value).

Given a data set for an XML document $\mathcal{X}$, it is reasonable to assume that the frequent queries are those that are *meaningful*. Therefore, without loss of generality, we assume that the workload is a subset of $DownPaths(\mathcal{X})$.

DEFINITION 4.1. *Given an XML document $\mathcal{X}$ and a workload F, the* k-extension *of F in $\mathcal{X}$, denoted $Ext^{\mathcal{X},k}(F)$, is the union of F and all downward paths of length $\leq k$: $Ext^{\mathcal{X},k}(F) = F \cup DownPaths(\mathcal{X}, k)$.*

For the convenience of the discussion, for a label-path set S, we use $|S|$ to represent the number of label-paths in S, and use $length(S)$ to represent the length of the longest label-path of the set, e.g. $length(S) = \max_{p \in S}(|lp|)$. From the definitions above, it is obvious that

- $|Ext^{\mathcal{X},k}(F)| \leq |F| + |DownPaths(\mathcal{X}, k)|$; and
- $k \leq length(Ext^{\mathcal{X},k}(F)) \leq max(k, length(F))$.

The $\mathcal{WP}[k]$-Trie index ($\mathcal{W}$orkload-aware $\mathcal{P}[k]$-Trie index) is the simplest form of a $\mathcal{P}[k]$-based, workload-aware index. It indexes the label-paths in the *k-extension* of a given workload, where the k value can be configured based on the query workload and the space allowance for the index.

DEFINITION 4.2. *Given an XML document $\mathcal{X}$, a workload F and a parameter k, the $\mathcal{WP}[k]$-Trie index of $\mathcal{X}$ that is sensitive to F is a trie index that indexes the label-paths in $Ext^{\mathcal{X},k}(F)$: the index keys are the inverted label-paths in $Ext^{\mathcal{X},k}(F)$ and the extent of each index entry associated with a label-path lp is $\mathcal{P}[l][lp]$, the $\mathcal{P}[l]$-partition class of $\mathcal{X}$ with $l = length(Ext^{\mathcal{X},k}(F))$.*

EXAMPLE 4.1. *Consider the example XML document $\mathcal{X}$ as shown in Figure 1. The $\mathcal{WP}[1]$-Trie index of $\mathcal{X}$ that is sensitive to workload $F = \{(A, A, B, C, D), (A, F, B, C, D)\}$ is shown in Figure 3. Please note that there is no index entry associated with label-paths (B,C,D), (A,B,C,D) and (F,B,C,D), since they are not part of $Ext^{\mathcal{X},1}(F)$.*

4.2 Query Evaluation with the $\mathcal{WP}[k]$-Trie

The $\mathcal{WP}[k]$-Trie index provides a *lookup* function that is similar to that of the $\mathcal{P}[k]$-Trie index, or any traditional trie index. Given a $\mathcal{WP}[k]$-Trie index T and a label-path query lp, the lookup function $T[lp]$ retrieves the extent of the index entry that can be located with key lp^{-1}, or $\emptyset$ otherwise.

Let's consider evaluating a label-path query lp against the XML document $\mathcal{X}$, on which a $\mathcal{WP}[k]$-Trie index is available [4]. It is not always the case that an index entry can be allocated that matches lp. It would be the task of the query optimizer to perform query decomposition and generate an optimal evaluation plan. As discussed in Section 3, besides index *lookup*, our workload-aware index framework also provides an index *probe* function, which assists the query optimizer in accomplishing its task. The index *probe* function takes a label-path lp as input and returns the label-path that is the best (longest) match to lp, or $\emptyset$ when there is enough information in the index to determine that $lp(\mathcal{X}) = \emptyset$.

Given a label-path lp and a $\mathcal{WP}[k]$-Trie index T of $\mathcal{X}$ that is sensitive to workload F, we say that lp incurs an *index hit* if $lp \in Ext^{\mathcal{X},k}(F)$. We call it a *structural hit* if there exists a label-path $lp' \in Ext^{\mathcal{X},k}(F)$ such that $lp \sqsubseteq^s lp'$. We call it a *true structural hit* if it is a *structural hit*, but not an *index hit*. We call the longest suffix of lp that can cause a structural hit in T the *structHit-suffix* of lp in T and the longest suffix of lp that can cause an index hit in T the *idxHit-suffix* of lp in T.

LEMMA 4.1. *Let T be a $\mathcal{WP}[k]$-Trie index of XML document $\mathcal{X}$ that is sensitive to workload F. Given a label-path query lp, let lp_s be the structHit-suffix of lp in T and lp_i be the idxHit-suffix of lp in T, then,*

- $|lp| \geq |lp_s| \geq |lp_i|$;

[4]It has been proved in [2] that the $\mathcal{P}[k]$-Trie index is capable of supporting index-only plans for answering any core XPath queries, therefore, in this paper, we focus only on the efficient evaluation of label-path queries.

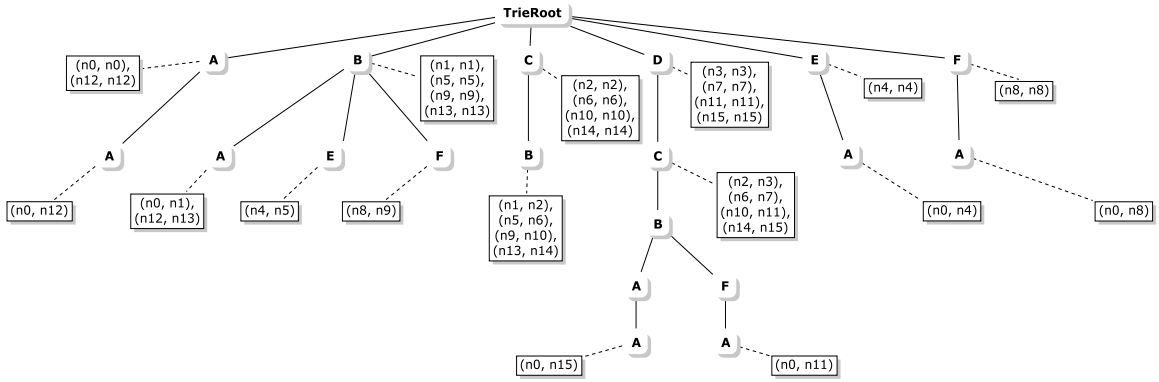

Figure 3: $\mathcal{WP}[1]$-Trie for workload $F = \{(A, A, B, C, D), (A, F, B, C, D)\}$

- *lp incurs an index hit in T iff* $lp = lp_s = lp_i$;
- *lp incurs a true structural hit in T iff* $|lp| = |lp_s| > |lp_i|$;
- $lp \neq lp_s \wedge |lp_s| < k \Rightarrow lp(\mathcal{X}) = \emptyset$.

The correlation between a label-path query lp and its match in a $\mathcal{WP}[k]$-Trie index T can be summarized in the cases illustrated in Figure 4. Here, bold circles represent label-paths that are indexed; thin circles represent nodes in the index that serve as structural components; dotted circles represent label-path queries.

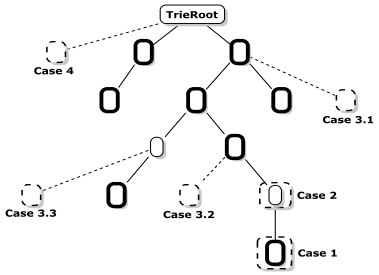

Figure 4: Cases in Evaluating a Path Query in a Workload-aware Trie Index

We now discuss how a path query can be evaluated in each of these cases. The basic idea behind the *probe* function is to return lp when it incurs an index hit; $\emptyset$ when the trie structure and k value can help decide that the result is $\emptyset$; and for all other cases, the *idxHit-suffix* of lp. Unless the *idxHit-suffix* itself is a frequent path, the *probe* function will return a label-path of length k.

Upon receiving the result of $probe(lp)$, the query optimizer will be able to generate one of the following evaluation plans: (1) if $probe(lp) = lp$, it will place an index access operator for lp; (2) if $probe(lp) = \emptyset$, it will conclude that the query result is empty and immediately inform the query evaluator; and (3) if $probe(lp) = lp'$ is a proper suffix of lp, it will decompose the query into two sub-queries: the returned path lp' and the rest of the label-path $\overline{lp'}$ [5], and then place an index access operator for lp' into the evaluation plan. The index *probe* is then repeated on $\overline{lp'}$. The behavior of the *probe* function is summarized in Table 1.

[5]Assuming $p \sqsubset^s lp$, we use $\overline{p}$ to represent a sub-query resulting from the decomposition of lp, such that $\overline{p} \circ p = lp$ and $\overline{p}$ and p share the token on their border.

Case#	Case Description	$probe(lp)$	$lp(\mathcal{X})$
Case 1	$lp = lp_s = lp_i$	lp	$T[lp]$
Case 2	$lp = lp_s \neq lp_i$	lp_i	$\overline{lp_i}(\mathcal{X}) \bowtie T[lp_i]$
Case 3.1	$lp \neq lp_s \wedge \lvert lp_s\rvert < k$	$\emptyset$	$\emptyset$
Case 3.2	$lp \neq lp_s \wedge lp_s = lp_i \wedge \lvert lp_s\rvert \geq k$	lp_s	$\overline{lp_s}(\mathcal{X}) \bowtie T[lp_s]$
Case 3.3	$lp \neq lp_s \wedge lp_s \neq lp_i \wedge \lvert lp_s\rvert \geq k$	lp_i	$\overline{lp_i}(\mathcal{X}) \bowtie T[lp_i]$
Case 4	$\lvert lp_s\rvert = \lvert lp_i\rvert = 0$	$\emptyset$	$\emptyset$

Table 1: Evaluating Label-path Queries with the $\mathcal{WP}[k]$-Trie Index

For any path query $lp \in F$, it is guaranteed that $lp \in Ext^{\mathcal{X},k}(F)$ and the evaluation of lp falls in Case 1. Therefore, it is guaranteed that lp can be answered with one index lookup. In addition, given that the $\mathcal{P}[k]$-partition of an XML document indexes the partition classes of node pairs, as stated and proved in [2], all core XPath queries can be evaluated by an index-only plan when $k \geq 1$. This result can be easily extended to the $\mathcal{WP}[k]$-Trie index, since the label-path set indexed by the $\mathcal{WP}[k]$-Trie index is a superset of the set indexed by a $\mathcal{P}[k]$-Trie index. In other words, the $\mathcal{P}[k]$-Trie index is a special case of the $\mathcal{WP}[k]$-Trie index, where $F \subseteq DownPaths(\mathcal{X}, k)$. Therefore, the following theorem stands.

THEOREM 4.1. *Given an XML document $\mathcal{X}$ and a workload F, when the $\mathcal{WP}[k]$-Trie index of $\mathcal{X}$ that is sensitive to F is available, all path queries in F can be answered in one index lookup and all core XPath queries can be answered with index-only plans.*

EXAMPLE 4.2. *Given the XML document $\mathcal{X}$ as shown in Figure 1, assume that the $\mathcal{WP}[1]$-Trie index as shown in Figure 3 is available. Let's consider a few queries against $\mathcal{X}$.*

Q_1 = *//A/F/B/C/D. Case 1,* $probe(Q_1) = (A, F, B, C, D)$, *then* $Q_1(\mathcal{X}) = T[(A, F, B, C, D)] = \{(n_0, n_{11})\}$.

Q_2 = *//E/D. Case 3.1.* $probe(Q_2) = \emptyset$, *then* $Q_2(\mathcal{X}) = \emptyset$.

Q_3 = *//A/B/C/D. Case 2,* $probe(Q_3) = (C, D)$. *The query optimizer will decompose Q_3 into two sub-queries (A, B, C) and (C, D). It repeats index probe on the label-path (A, B, C) and receives (B, C) as a result. Through*

another query decomposition, it probes (A, B) and receives (A, B) as a result. The evaluation plan will then be $Q_3(\mathcal{X}) = T[(A, B)] \bowtie T[(B, C)] \bowtie T[(C, D)]$.

Q_4 *= //D/A/B/C/D. Case 3.3, probe$(Q_4) = (C, D)$. The first few rounds of index probe and query decomposition are similar to those of Q_3. However, the last probe of (D, A) yields $\emptyset$. Therefore, the optimizer concludes that $Q_4(\mathcal{X}) = \emptyset$ without invoking any evaluation.*

4.3 $\mathcal{WP}[k]$-Trie Construction and Maintenance

Given an XML document $\mathcal{X}$ and a workload F, the $\mathcal{WP}[k]$-Trie index can be constructed with one depth-first traversal of $\mathcal{X}$, with the help of a stack whose size is bounded by the height of $\mathcal{X}$. The construction proceeds as follows:

Step 1 : Initialize the *labels of interest* (LOI) to be the set of tail labels of the label-paths in F.

Step 2 : Traverse $\mathcal{X}$. At any time during the traversal, when node n is being visited, the node label and node id of each node on the path $LP(r, n)$ are maintained in a stack, with the root node r at the bottom.

- **Step 2.1** : For each $i \in [0, k]$, compose a label-path lp using the labels of the top i nodes in the stack; insert the node pair $(stack[i].id, stack[0].id)$ into the index with key lp^{-1}.
- **Step 2.2** : If $\lambda(n) \in LOI$, for each $j \in [k{+}1, stackHeight]$, compose lp as in Step 2.1. If $lp \in F$, insert the corresponding node pair into the index with key lp^{-1}.

Indexes should always be updated simultaneously when data changes. As to the changes that can occur in an XML document, we limit our discussion to the insertion or deletion of a leaf node, since the insertion, deletion of sub-trees and the modification of nodes can be treated as a sequence of basic insertion and deletion operations.

Workload-aware indexes must be able to adapt quickly and efficiently when the workload changes in order to provide continuous support for efficient query evaluation, especially for frequent queries. In addition, an index should be easily adjusted when the space allowance changes, in the case of workload-aware indexes, to further benefit the evaluation of non-frequent queries. A key merit of the $\mathcal{P}[k]$-Trie index is that all core XPath queries can be evaluated by index-only plans. Therefore, information preserved in the index is sufficient for all index maintenance, including those changes triggered by adding and removing frequent label-paths and the refinement or compression of the index triggered by modifying the k value, without the need to access the XML document.

Document changes - add a leaf node: Assume that a node n_{new} is inserted into the XML document to be a child element of node n. Assume that the path between the root node r and n is $(n_0, n_1, \ldots, n_l)$ ($n_0 = r$ and $n_l = n$) and the label-path $LP(r, n) = (\lambda(n_0), \lambda(n_1), \ldots, \lambda(n_l))$. For the convenience of the discussion, we assign $n_{l+1} = n_{new}$. The modification to the $\mathcal{WP}[k]$-Trie will be local to the branch rooted at $\lambda(n_{new})$ and involves a few insertions to the index. Every node pair (n_i, n_{new}) $(l+1-k \leq i \leq l+1)$ will be inserted into the index with the label-path $(\lambda(n_i), \lambda(n_{i+1}), \ldots, \lambda(n_{l+1}))^{-1}$ as index key. For $j \in [0, l{-}k]$, the node pair (n_j, n_{new}) is inserted into the index only when the label-path $(\lambda(n_j), \lambda(n_{j+1}), \ldots, \lambda(n_{l+1})) \in F$.

Document changes - remove a leaf node: Adjustments in the index triggered by the deletion of a leaf node are limited to the index branch rooted at $\lambda(n_{rmv})$, assuming n_{rmv} is the node to be deleted. It is exactly the opposite of the process triggered by the insertion of a leaf node.

Workload changes - add a frequent label-path: Assume that a new label-path lp is added to the workload F. We take advantage of the information present in the $\mathcal{WP}[k]$-Trie and evaluate lp as a path query. If the result is not empty, we insert the resulting node pairs into the index, with lp^{-1} as key.

Workload changes - remove a frequent label-path: When a label-path lp is removed from the workload F, no action is needed if $lp \in DownPaths(\mathcal{X}, k)$; otherwise, the index entry associated with lp is removed.

Index Configuration Change - change k value: A nice property of the $\mathcal{P}[k]$-Trie index is that its layers are independent. The $\mathcal{P}[k]$-Trie index is simply the top k layers of the $\mathcal{P}[k{+}1]$-Trie index. Taking advantage of this property, the adjustment needed for decreasing the k value from k_1 to k_2 is simply removing the bottom $k_1 - k_2$ layers of the trie structure, with the exception that the index entry associated with a label-path lp will not be removed if $lp \in F$. Increasing the k value is exactly the opposite. To add a new layer at level k, the content of the new index entries can be computed by performing natural join between the entries on level k_1 and k_2, where $k_1 + k_2 = k$.

4.4 Properties and Limitations

The idea behind the $\mathcal{WP}[k]$-Trie index is straightforward. The $\mathcal{WP}[k]$-Trie is easy to construct and maintain and bears the following properties: Given an XML Document $\mathcal{X}$ a workload F, and T the $\mathcal{WP}[k]$-Trie index of $\mathcal{X}$ that is sensitive to F, then,

- The height of T, therefore, the maximum search length during lookup, is bounded by $length(Ext^{\mathcal{X},k}(F))$, and $length(Ext^{\mathcal{X},k}(F)) \leq max(k, length(F))$;
- The number of index entries in T is no larger than $|DownPaths(\mathcal{X}, k)| + |F|$. Consider that under most circumstances $|DownPaths(\mathcal{X}, k)| \gg |F|$, thus the size of T is close to the size of the $\mathcal{P}[k]$-Trie of $\mathcal{X}$.

Although the $\mathcal{WP}[k]$-Trie index with a very modest k value can help evaluate all frequent path queries in one index lookup and evaluate all core XPath queries with an index-only plan, it still presents some limitations:

Limitation #1 The information kept in the index is not sufficient to always determine whether a path query results in an empty set (as per Case 3.2), but has to do so through multiple query decomposition and *probe* operations, as illustrated by Q_4 in Example 4.2.

Limitation #2 For non-frequent path queries whose length is larger than k, it is almost always the case that it will be decomposed into a sequence of sub-queries of length k, unless a frequent label-path happens to be its suffix. This may result in a significant number of joins in the query evaluation process when the k value is small. Query Q_3 in Example 4.2 is such an example.

Limitation #3 Indexing all label-paths of length $0 \backsim k$ in the $\mathcal{WP}[k]$-Trie significantly increases the size of the index. This results in "over-populated" layers at the top of a $\mathcal{WP}[k]$-Trie index, while some of these index entries are rarely used.

5. ANNOTATED $\mathcal{WP}[k]$-TRIE INDEX

The limitations of the $\mathcal{WP}[k]$-Trie index outlined above indicate that there is still space for improvement in the design of the workload-aware Trie indexes with respect to the space/performance trade-off. In this section, we propose the $\mathcal{AWP}[k]$-Trie index ($\mathcal{A}$nnotated $\mathcal{WP}[k]$-Trie index) as the solution.

5.1 The Study of Representativeness

We first address limitation #1 of the $\mathcal{WP}[k]$-Trie index, which is the inability to identify some path queries whose result is empty. The root of this problem is that by indexing only a subset of the downward label-paths beyond length k, the index no longer "fully represents" the structural distribution of the XML document.

DEFINITION 5.1. *Given a label-path $lp \in DownPaths(\mathcal{X})$ and a set of label-paths S, we say that lp is structurally represented by S with respect to $\mathcal{X}$, denoted $lp \prec^s S$, if for every label-path lp' in $DownPaths(\mathcal{X})$ which is one step longer than lp and has lp as a suffix, either $lp' \in S$ or lp' is a suffix of a label-path in S.*

Given an XML document $\mathcal{X}$ and a workload F, for every lp that is either in $Ext^{\mathcal{X},k}(F)$ or is the suffix of a label-path in $Ext^{\mathcal{X},k}(F)$, we can easily compute the boolean flag $struct(lp) = lp \prec^s Ext^{\mathcal{X},k}(F)$. This flag will allow us to identify more queries that result in $\emptyset$ than the $\mathcal{WP}[k]$-Trie.

LEMMA 5.1. *Given an XML document $\mathcal{X}$ a workload F, and a path query lp, we can conclude that $lp(\mathcal{X}) = \emptyset$ if its structHit-suffix in the $\mathcal{W}[k]$-Trie index is a proper suffix of lp and it is structurally represented by $Ext^{\mathcal{X},k}(F)$. E.g.*

$$lp_s \sqsubset^s lp \wedge lp_s \prec^s Ext^{\mathcal{X},k}(F) \Rightarrow lp(\mathcal{X}) = \emptyset$$

We now address limitation #2 of the $\mathcal{WP}[k]$-Trie index, which is the potential over-shredding of a path query in query evaluation. The root of the problem is that the evaluation of a path query always resorts to query decomposition when it does not incur an index hit in the probe. Since only frequent paths of length $> k$ are indexed, non-frequent queries are decomposed into a sequence of sub-queries of length k. This is clearly not the most efficient evaluation plan for some queries, especially those that incur a true structural hit and whose structHit-suffix is structurally represented in $Ext^{\mathcal{X},k}(F)$. Q_3 in Example 4.2 is such a case.

Let's recall Theorem 2.1 introduced in Section 2. We would like to be able to identify all queries that can be evaluated using Theorem 2.1, e.g, those queries whose label-paths in $LPS(E, \mathcal{X})$ are all indexed, facilitating evaluation whenever possible.

DEFINITION 5.2. *Given an XML document $\mathcal{X}$, a set of label-paths S and a label-path $lp \in DownPaths(\mathcal{X})$, we say that lp is represented at the instance level by S with respect to $\mathcal{X}$, denoted $lp \prec^i S$, if there exists a label-path set $S' = \{lp' | lp' \in S \wedge lp \sqsubseteq^s lp'\}$, such that $lp(\mathcal{X}) = \bigcup_{lp' \in S'} (lp'(\mathcal{X}))$.*

Note that S' does not necessarily include all label-paths that have lp as their suffix, but rather, there exists a smallest S' which includes only the "closest" label-paths that have lp as their suffix.

To summarize our study of label-path representativeness, we define the notion of *full representativeness* as follows:

DEFINITION 5.3. *Given an XML document $\mathcal{X}$, a set of label-paths S and a label-path $lp \in DownPaths(\mathcal{X})$, we say that lp is* fully represented *by S with respect to $\mathcal{X}$, denoted $lp \prec S$, if $lp \prec^s S \wedge lp \prec^i S$.*

5.2 $\mathcal{AWP}[k]$-Trie Index

Equipped with the notions of structural and instance level representativeness of a label-path by a label-path set, we are now ready to define the notion of a *self-sustaining* label-path set and the *self-sustaining closure* of a label-path set.

DEFINITION 5.4. *Given an XML document $\mathcal{X}$ and a label-path set S, we say that S is* self-sustaining *with respect to $\mathcal{X}$ if for any label-path lp that is a suffix of a path in S, it is the case that (1) $lp \in S$; or (2) $lp \prec S$.*

DEFINITION 5.5. *Given an XML document $\mathcal{X}$ and label-path set S, we define the* self-sustaining closure *of S, denoted $S^{\mathcal{X}+}$, to be the minimum among all sets that has S as a subset and is* self-sustaining *with respect to $\mathcal{X}$.*

The $\mathcal{AWP}[k]$-Trie index of an XML document $\mathcal{X}$, sensitive to workload F, indexes the label-paths in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$, including annotations that reflect the structural representativeness property for each suffix label-path in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$.

DEFINITION 5.6. *Given an XML document $\mathcal{X}$, a workload F, and a parameter k, the $\mathcal{AWP}[k]$-Trie index of $\mathcal{X}$ that is sensitive to F is a trie index that indexes the label-paths in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$: the index keys are the inverted label-paths in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$ and the extent of each index entry associated with a label-path lp is $\mathcal{P}[l][lp]$, the $\mathcal{P}[l]$-partition class of $\mathcal{X}$ with $l = length((Ext^{\mathcal{X},k}(F))^{\mathcal{X}+})$. A* struct *flag is associated with each label-path that is either in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$ or is a suffix path of a label-path in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$ to indicate whether it is structurally represented by $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$.*

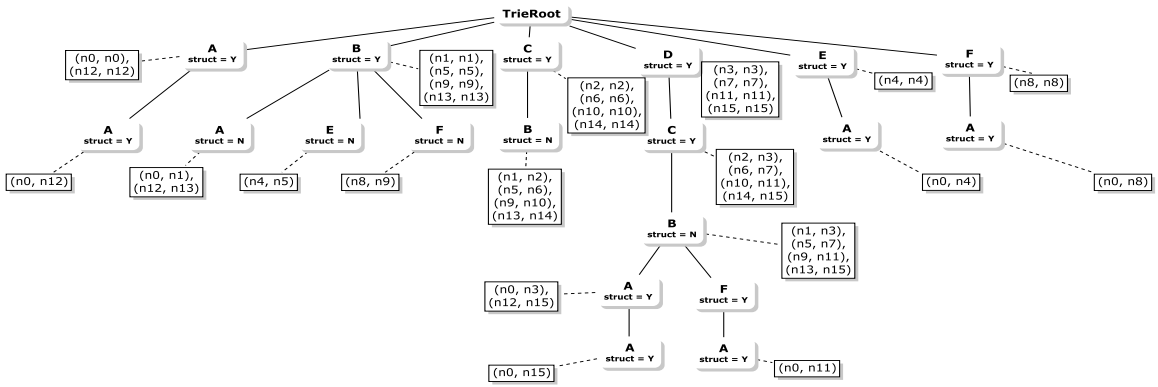

Figure 5: $\mathcal{AWP}[1]$-Trie of $\mathcal{X}$ with $F = \{(A, A, B, C, D), (A, F, B, C, D)\}$.

EXAMPLE 5.1. *Consider the example XML document $\mathcal{X}$ shown in Figure 1. The $\mathcal{AWP}[1]$-Trie index of $\mathcal{X}$ that is sensitive to workload $F = \{(A, A, B, C, D), (A, F, B, C, D)\}$ is shown in Figure 5.*

In this index structure, every node has an associated struct *flag. For example, $struct((A, A)) = TRUE$, since there is no label-path in $DownPaths(\mathcal{X})$ that has (A, A) as a proper suffix; $struct((B, C, D)) = FALSE$, since the label-path (E, B, C, D) of $DownPaths(\mathcal{X})$ is not in $Ext^{\mathcal{X},1}(F)^{\mathcal{X}+}$; and $struct((A, B, C, D)) = TRUE$, since (A, A, B, C, D), the only label-path in $DownPaths(\mathcal{X})$ that has (A, B, C, D) as a proper suffix is in $(Ext^{\mathcal{X},1}(F))^{\mathcal{X}+}$.*

Comparing this index to the $\mathcal{WP}[1]$-Trie for the same document and workload (as shown in Figure 3), two label-paths that were not indexed in the $\mathcal{WP}[1]$-Trie index are indexed here: $lp_1 = (B, C, D)$, since $lp_1 \not\prec^s (Ext^{\mathcal{X},1}(F))$, and $lp_2 = (A, B, C, D)$, since $lp_2 \not\prec^i (Ext^{\mathcal{X},1}(F))$.

5.3 Query Evaluation with the $\mathcal{AWP}[k]$-Trie

We define two lookup methods for the $\mathcal{AWP}[k]$-Trie index: direct and sub-tree lookup.

Given an $\mathcal{AWP}[k]$-Trie T and a label-path query lp, the direct lookup of lp, denoted $T[lp]$, retrieves the extent of an index entry that can be located with key lp, or $\emptyset$ otherwise. It provides the answer to $lp(\mathcal{X})$ under path semantics.

The sub-tree lookup of lp in T, denoted $\widehat{T}[lp]$, computes $lp(\mathcal{X})$ using the Theorem 2.1. The extents of the proper index entries in the sub-tree rooted at the index node that matches lp participate in a union on their second projections. If lp is not structurally represented in the index, then Theorem 2.1 cannot be applied to lp and its result is NULL. A sub-tree lookup provides the answer to $lp(\mathcal{X})$ under node semantics.

Please note that only the extents of the "closest" index entries on each branch in the sub-tree rooted at the index entry associated with lp are needed in the sub-tree lookup. Formally, we define the sub-tree lookup as follows:

$$\widehat{T}[lp] = \bigcup_{\substack{lp' \in \{p \mid p \in (Ext^{\mathcal{X},k}(F))^{\mathcal{X}+} \wedge lp \sqsubseteq^s p \wedge \\ \neg\exists p'' \in (Ext^{\mathcal{X},k}(F))^{\mathcal{X}+} (lp \sqsubseteq^s p'' \wedge p'' \sqsubset^s lp')\}}} \pi_2(T[lp'])$$

In order for the query optimizer to take full advantage of the two lookup functions, the $\mathcal{AWP}[k]$-Trie index provides a *probe* function that takes a label-path as input and returns the best (longest) label-path for both direct and sub-tree lookup. For the clarity of the discussion, we use $probe_d(lp)$ to represent the returned label-path for direct lookup, and $probe_s(lp)$ for the label-path returned for the sub-tree lookup. $probe_s(lp)$ is set to NULL if *probe* can not propose a sub-tree lookup better than what is proposed for a direct lookup. Upon receiving the probe result, it will be up to the query optimizer to decide how to utilize the information and generate an efficient evaluation plan.

Given an XML document $\mathcal{X}$ and a workload F, Table 2 presents the behavior of the *probe* function and the evaluation plan(s) that can be generated for evaluating path query lp with the help of an $\mathcal{AWP}[k]$-Trie index of $\mathcal{X}$ that is sensitive to F. Again, the cases illustrated in Figure 4 are the only cases that may occur, but now we can take into consideration the *struct* flags in the index nodes.

The $\mathcal{AWP}[k]$-Trie index can still answer frequent queries with a direct index lookup, but can perform much better than the $\mathcal{WP}[k]$-Trie on non-frequent queries. We are able to address limitation #1 of the $\mathcal{WP}[k]$-Trie index with the introduction of the *struct* flag. We are also able to address limitation #2 with the sub-tree lookup and a more sophisticated *probe* function. Moreover, we are indexing more label-paths, increasing the likelihood and frequency of cases where the length of lp_s and lp_i is longer than k, further relieving the over-decomposition problem.

In addition, by supporting the evaluation of a path query under the node semantics for longer sub-queries, the $\mathcal{AWP}[k]$-Trie opens the door for more complicated query optimization, and in most cases, more efficient query evaluation plans. For example, for Case 3.3, when $struct(lp_s) = FALSE$, the default evaluation plan is

$$\overline{lp_i}(\mathcal{X}) \bowtie T[lp_i]$$

However, in case there is a significant difference between the cardinalities of $lp_s(\mathcal{X})$ and $lp_i(\mathcal{X})$, a much more efficient evaluation plan is

$$\overline{lp_i}(\mathcal{X}) \bowtie (T[lp_i] \bowtie \widehat{T}[lp_s])$$

Case#	Case Description	$struct(lp_s)$	Index Probe		Query Evaluation
			$probe_d(lp)$	$probe_s(lp)$	
Case 1	$lp = lp_s = lp_i$		lp	NULL	$lp(\mathcal{X}) = T[lp]$
Case 2	$lp = lp_s \neq lp_i$	$TRUE$	lp_i	lp_s	$lp(\mathcal{X}) = \overline{lp_i}(\mathcal{X}) \bowtie T[lp_i]$ $lp^{node}(\mathcal{X}) = \widehat{T}[lp_s]$
		$FALSE$	lp_i	NULL	$lp(\mathcal{X}) = lp_i(\mathcal{X}) \bowtie T[lp_i]$
Case 3.1	$lp \neq lp_s \wedge \lvert lp_s\rvert < k$		$\emptyset$	$\emptyset$	$\emptyset$
Case 3.2	$lp \neq lp_s \wedge lp_s = lp_i \wedge \lvert lp_s\rvert \geq k$	$TRUE$	$\emptyset$	$\emptyset$	$\emptyset$
		$FALSE$	lp_s	NULL	$lp(\mathcal{X}) = \overline{lp_s}(\mathcal{X}) \bowtie T[lp_s]$
Case 3.3	$lp \neq lp_s \wedge lp_s \neq lp_i \wedge \lvert lp_s\rvert \geq k$	$TRUE$	$\emptyset$	$\emptyset$	$\emptyset$
		$FALSE$	lp_i	lp_s	$lp(\mathcal{X}) = \overline{lp_i}(\mathcal{X}) \bowtie T[lp_i]$ $lp(\mathcal{X}) = \overline{lp_i}(\mathcal{X}) \bowtie (T[lp_i] \bowtie \widehat{T}[lp_s])$
Case 4	$\lvert lp_s\rvert = \lvert lp_i\rvert = 0$		$\emptyset$	$\emptyset$	$\emptyset$

Table 2: Evaluating Label-path Queries with $\mathcal{AWP}[k]$-Trie Index

which uses the result of $lp_s(\mathcal{X})$ to filter $lp_i(\mathcal{X})$ before it participates in the join with the results of other sub-queries.

EXAMPLE 5.2. *Let's again consider the queries discussed in Example 4.2. There is no change to the evaluation of Q_1 and Q_2, but Q_3 and Q_4 will be evaluated more efficiently with the help of the $\mathcal{AWP}[1]$ index shown in Figure 5.*

- *Q_3 = //A/B/C/D. Case 2, $struct((A,B,C,D)) = TRUE$. Therefore, $probe_d(Q_3) = (B,C,D)$ and $probe_s(Q_3) = (A,B,C,D)$. So, we can evaluate query Q_3 under node semantics: $Q_3^{node}(\mathcal{X}) = \widehat{T}[Q_3] = \{n_3, n_{15}\}$.*
- *Q_4 = //D/A/B/C/D. Case 3.2. The* structHit-suffix *of Q_4 in T is (A,B,C,D) and $struct((A,B,C,D)) = TRUE$. Therefore, $probe_p(Q_4) = \emptyset$, and upon receiving this information, the conclusion is immediately made that $Q_4(\mathcal{X}) = \emptyset$.*

Please note that with the $\mathcal{AWP}[1]$-Trie index, all four example queries are either determined to have empty results, or can be evaluated by one index lookup.

5.4 $\mathcal{AWP}[k]$-Trie Construction and Maintenance

The difficulty in the construction of an $\mathcal{AWP}[k]$-Trie index lies in determining the structural and instance level representativeness of the label-paths, and hence the membership in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$. However, with some careful bookkeeping, we can accomplish this in a single scan over the XML document, at the same time that the index is populated. The construction algorithm for the $\mathcal{AWP}[k]$-Trie index is shown in Figure 6. Given an XML document $\mathcal{X}$, a workload F and a value k, we employ the following bookkeeping agents in the process of constructing an $\mathcal{AWP}[k]$-Trie index:

all_paths is used to keep track of the suffixes of all paths in $Ext^{\mathcal{X},k}(F)$. It is initialized with the label-paths in F and their suffixes, and is later populated during the file scan.

Ext is used to keep track of the label-paths that should be in $(Ext^{\mathcal{X},k}(F))^{\mathcal{X}+}$. It is initialized with the label-paths in F and their suffixes of length $\leq k$. During the file scan, a label-path lp in $\mathcal{X}$ is added to *Ext* if $|lp| \leq k$ or if lp fails the structural or instance-level representativeness test.

iRep_undecided is used to keep track of the set of label-paths whose instance-level representativeness is yet to be determined. It is initialized with all the label-paths that satisfy the following criteria: (1) longer than k; (2) is not a frequent label-path but is a proper suffix of a frequent label-path; and (3) bears the label of the root node as its first token. No more label-paths will be added to *iRep_undecided* during the scan of the XML document. A label-path is removed from this set when it fails the structural or instance-level representativeness test.

sRep_undecided is used to keep track of the set of label-paths whose structural representativeness is yet to be determined. It is initialized with all the label-paths in F and their suffixes of length $\geq k$. During the file scan, a label-path lp in $\mathcal{X}$ is added to *sRep_undecided* if $lp \notin Ext$ and $|lp| = k$. A label-path is removed from this set when it fails the structural representativeness test.

To ensure efficient index construction, we want to perform the representativeness tests only when it is absolutely necessary. Given a label-path lp, a structural representativeness test is triggered only when its *struct* flag is undecided and when there is a known path in $\mathcal{X}$ that contains lp. An instance-level representativeness test is triggered only when a node pair that involves the root node is inserted into the index and $lp \in$ *iRep_undecided*. At the end of the file scan, any label-path that has not been assigned a FALSE *struct* flag, can be assumed to be structurally represented. Any label-path with a *struct* flag set to TRUE that is not in *Ext* will be removed from the index.

The maintenance of the $\mathcal{AWP}[k]$-Trie index triggered by changes in data, workload, and index configuration is very similar to that of the $\mathcal{WP}[k]$-Trie index. Slight adjustments are needed to conduct the structural and instance-level representativeness tests on related label-paths. The scope of the impact is limited only to the label-paths in question and their suffixes. Due to the space limitations, details are omitted.

5.5 Properties of $\mathcal{AWP}[k]$-Trie Index

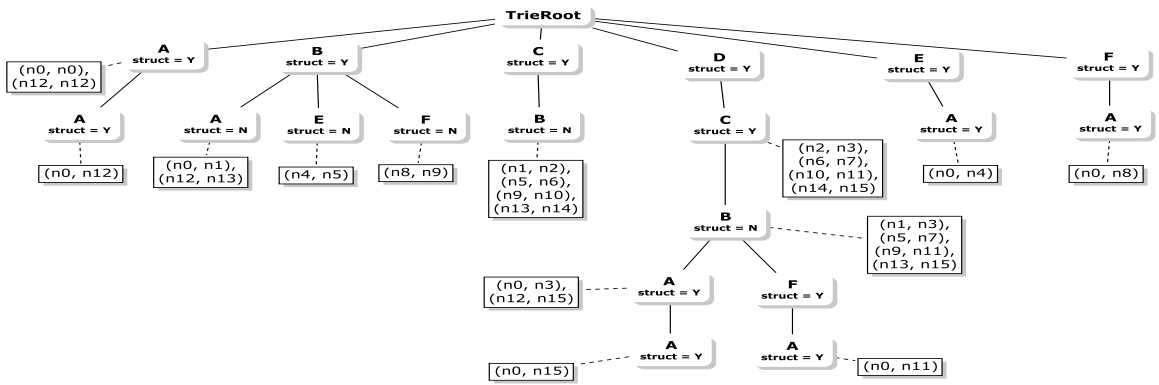

Figure 7: $\mathcal{W}[1]$-Trie with $F = \{(A, A, B, C, D), (A, F, B, C, D)\}$.

The $\mathcal{AWP}[k]$-Trie index provides significant improvements in query performance, especially for non-frequent queries, over the $\mathcal{WP}[k]$-Trie index. It also keeps a nice property of the $\mathcal{WP}[k]$-Trie index, in that the height of the index is still bounded by $max(k, length(F))$, limiting the maximum search length during lookup. The number of index entries in an $\mathcal{AWP}[k]$-Trie index T is no larger than $|DownPaths(\mathcal{X}, k)| + \sum_{lp \in F} (|lp| - k)$. Usually, $|DownPaths(\mathcal{X}, k)| \gg |F|$, thus the size of T is close to the size of the $\mathcal{P}[k]$-Trie index of $\mathcal{X}$.

While the $\mathcal{AWP}[k]$-Trie index significantly improves the $\mathcal{WP}[k]$-Trie, it still does not fix limitation #3 of the $\mathcal{WP}[k]$-Trie - the over-population of the label-paths shorter than k - as pointed out in Section 4.4. We propose a variant of the $\mathcal{AWP}[k]$-Trie to address this issue.

5.6 $\mathcal{W}[k]$-Trie Index

We now propose a variant of the $\mathcal{AWP}[k]$-Trie index to further improve space efficiency while introducing a minimum impact to query performance. Careful selection of an appropriate k value is crucial for any workload-aware Trie index to ensure its effectiveness. While the k value may reflect index space considerations, it is also possible for it to represent the length of the majority of the non-frequent label-paths, a value that can be easily obtained by the workload analyzer. If we make such an assumption, we can define a restricted *complementary* label-path set that emphasizes the importance of non-frequent label-paths of length 1 and k, avoiding over-population of other layers in the trie.

Given an XML document $\mathcal{X}$ and a frequent label-path set F, we define the *restricted k-extension* of F, denoted $Ext^{\mathcal{X},(1,k)}(F)$, as the union of F and all downward label-paths in $\mathcal{X}$ that are of length 1 and k. The $\mathcal{W}[k]$-Trie index indexes all label-paths in the *self-sustaining closure* of $Ext^{\mathcal{X},(1,k)}(F)$, with annotations that indicate the *structural representativeness* of the label-paths in $Ext^{\mathcal{X},(1,k)}(F)$ and their suffixes.

EXAMPLE 5.3. *Figure 7 shows the $\mathcal{W}[1]$-Trie index for the document $\mathcal{X}$ shown in Figure 1 that is sensitive to workload $F = \{(A, A, B, C, D), (A, F, B, C, D)\}$. Note that there are no index entries associated with any label-paths of length 0 (except for (A) which fails the instance-level representativeness test) since they are not part of $Ext^{\mathcal{X},(1,1)}(F)$.*

The structure of the $\mathcal{AWP}[k]$ and the $\mathcal{W}[k]$-Trie index is exactly the same for label-paths with length $\geq k$. A label-path lp with length $< k$ will only be indexed in the $\mathcal{W}[k]$-Trie if $lp \in F$, if its length is 1, or if lp does not pass the instance-level representativeness test in $Ext^{\mathcal{X},(1,k)}(F)$. Therefore, the $\mathcal{W}[k]$-Trie index is much more "sparse" on the top k levels and much smaller in size when compared to its $\mathcal{WP}[k]$-Trie and $\mathcal{AWP}[k]$-Trie counterparts.

With regards to query evaluation, the $\mathcal{W}[k]$-Trie index is also able to answer frequent path queries with a single index lookup, and it possesses the same ability in quickly identifying queries that result in $\emptyset$ as the $\mathcal{AWP}[k]$-Trie. The $\mathcal{W}[k]$-Trie index provides the same *lookup* and *probe* functions as the $\mathcal{AWP}[k]$-Trie index, which assist the query optimizer in generating efficient evaluation plans. The only potential flaw of the $\mathcal{W}[k]$-Trie index is that when compared to the $\mathcal{AWP}[k]$-Trie, it may result in over-shredding of queries with length less than k. However, if the k value does represent the length of the majority of non-frequent label-paths, it may indeed provide better query decomposition and improve query performance over the $\mathcal{AWP}[k]$-Trie index, whose k value is chosen because of space considerations.

Due to space limitations, we omit the construction and maintenance details of the $\mathcal{W}[k]$-Trie index, which are simple extensions of the corresponding algorithms for the $\mathcal{AWP}[k]$-Trie index.

6. EXPERIMENTAL EVALUATION

6.1 System Setup

In order to test the effectiveness and efficiency of the workload-aware Trie indexes proposed in this paper, we compare them against each other, and against two existing indexes: the $\mathcal{P}[k]$-Trie index [2] and APEX [5]. All indexes were implemented using Timber [11], a native XML database system, following the architecture described in Figure 2. We chose APEX as our baseline comparison as it is a workload-aware structural index that also claims to provide index-only query evaluation plans for any XPath query.

The experiments were conducted on the DBLP data set [17], the NASA data set [15], and synthetic data sets we generated, on a Microsoft Windows XP computer, with an Intel Pentium 4 3.2GHz CPU, 2GB of available RAM, and

```
AWP[k]-Trie Index Construction
Input:  XML document X, workload F, k.
{
  \* Initializations *\
  all_paths = F ∪ {lp | ∃lp' ∈ F(lp ⊑^s lp')};
  Ext = F ∪ {lp | |lp| ≤ k ∧ ∃lp' ∈ F(lp ⊑^s lp')};
  iRep_undecided = {lp | |lp| > k ∧ lp ∉ F ∧ ∃lp' ∈ F(lp ⊏^s lp')
        ∧ the first token of lp is λ(r)};
  sRep_undecided = {lp | |lp| ≥ k ∧ ∃lp' ∈ F(lp ⊑^s lp')};
  \* File Scan and Index Construction *\
  \* Traverse the XML document in a depth first fashion.  *\
  \* At any time during the traversal, when node n is being *\
  \* visited, the node label and node id of each node on the*\
  \* path LP(r, n) are maintained in a stack, with the root *\
  \* node at the bottom.*\
  for ( n ∈ X ) {
    for (i = 0, ..., stackHeight) {
      \* Insert index entry, populate book-keeping agents *\
      if (i > k ∧ stack[0].label ∉ LOI) break;
      node_pair = (stack[i].id, stack[0].id);
      lp = (stack[i].label, stack[i − 1].label, ..., stack[0].label);
      if (i > k ∧ lp ∉ all_paths) break;
      else {
        insert lp into index, with extent node_pair;
        if (lp ∉ all_paths) all_paths = all_paths ∪ {lp};
        if (i = k ∧ lp ∉ Ext)
          sRep_undecided = sRep_undecided ∪ {lp};
        if (i < k) set struct(lp) = TRUE ;
        if (i ≤ k) Ext = Ext ∪ {lp};
      }
      \* Instance-level representativeness test *\
      if (stack[i].id = r.id ∧ lp ∈ iRep_undecided) {
        iRep_undecided = iRep_undecided − {lp};
        Ext = Ext ∪ {lp};
      }
      \* Structural representativeness test *\
      if (i ≠ stackHeight ∧ lp ∈ sRep_undecided) {
        lp' = stack[i + 1].label + lp;
        if (lp' ∉ all_paths) {
          sRep_undecided = sRep_undecided − {lp};
          iRep_undecided = iRep_undecided − {lp};
          Ext = Ext ∪ {lp}; set struct(lp) = FALSE;
        }
      }
    }
  }
  \* Set struct flags and remove extra index entries *\
  for (lp ∈ sRep_undecided − Ext) {
    set struct(lp) = TRUE;
    remove index entry lp from the index.
  }
}
```

Figure 6: $\mathcal{AWP}[k]$-Trie Construction

Timber's default settings. The statistical information of the DBLP and NASA data sets, on which results will be reported, is summarized in Table 3.

Data Set	# of Nodes	Max Depth	Avg Depth
DBLP	1.3M	6	2.9
NASA	1.4M	8	5.6

Table 3: XML Data Sets

We designed and implemented a query set generator that takes the workload F, the downward paths of an XML document, and a few configuration parameters as input, and generates a set of XPath queries against the given XML document, where the label-paths in F appear frequently (e.g. above the given threshold) and all other label-paths appear non-frequently.

Three different sets of frequent label-paths were used in index construction, including a *deep* F with a few label-paths whose length is $\gg k$, a *wide* F with a decent number of label-paths with length $> k$, and a *mixed* F with a combination of longer and shorter paths.

The results obtained for all combinations of data set, workload and query sets exhibited similar trends in both index construction and query evaluation. Due to space considerations, we present the results for the NASA file with the *mixed* F.

6.2 Index Construction

The depth-first traversal algorithm was used for the construction of all indexes. In particular, the $\mathcal{P}[k]$-Trie and APEX were constructed following the algorithms defined in [2] and [5] respectively. The $\mathcal{WP}[k]$, $\mathcal{AWP}[k]$ and $\mathcal{W}[k]$ Trie indexes were constructed following the corresponding algorithms as described in Sections 4.3, 5.4, and 5.6, respectively. For the $\mathcal{P}[k]$-Trie and the workload-aware indexes, we constructed each type of index with parameter $k = 1, 2, 3$. The time spent on index construction and the number of index entries are shown in Figures 8 and 9.

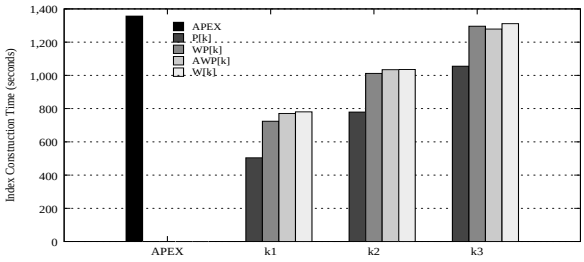

Figure 8: Index Construction Time (NASA Data Set)

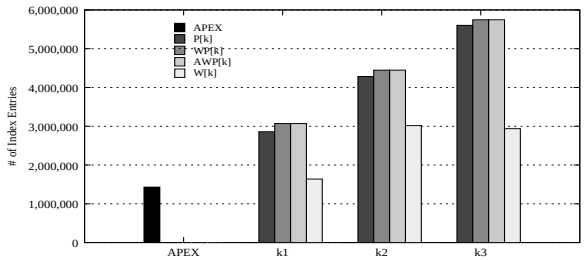

Figure 9: Index Size (NASA Data Set)

The results show that for the Trie index family, both construction time and index size are directly correlated to the parameter k. While the workload-aware Trie indexes are slightly more expensive to construct compared to their $\mathcal{P}[k]$-Trie peers, they outshine or are at lease comparable to their $\mathcal{P}[k]$-Trie peers in terms of the index size requirements.

Naturally, the workload-aware Trie indexes are more expensive to construct, compared to the $\mathcal{P}[k]$-Trie index, due to the additional label-paths in F that need to be considered and indexed while traversing the document. However, there is no significant difference in the construction time between the three workload-aware indexes. Even though the structural and instance-level representativeness needs to be computed for every label-path in the index, resulting in additional label-paths to be indexed, the careful book-keeping in the construction algorithms of the $\mathcal{AWP}[k]$ and $\mathcal{W}[k]$-Trie succeeds in keeping this overhead to the minimum.

The result in Figure 9 echoes our analysis that the size of the $\mathcal{WP}[k]$-Trie and the $\mathcal{AWP}[k]$-Trie indexes should be comparable to the $\mathcal{P}[k]$-Trie index with the same k value, since the selection strategies used in constructing the *complementary* set of label-paths that are indexed ensure that we find the minimal extensions required to reflect the workload and improve index performance. Most importantly, the size of the $\mathcal{W}[k]$-Trie index is *significantly* smaller than other indexes in the Trie family with the same k value. On average, the $\mathcal{W}[k]$-Trie index provides a size reduction of 40% compared to the $\mathcal{P}[k]$-Trie, and the $\mathcal{W}[1]$-Trie index posts only a 15% size increase compared to APEX.

6.3 Query Evaluation

The query set generation tool we designed can generate simple path queries that are frequent label-paths or non-frequent label-paths, and more complicated XPath queries with branch predicates that consist of sub-queries that are both frequent and non-frequent label-paths. We are interested in how different indexes answer these queries.

We will first discuss a few scenarios in which the experiments verified the behavior of the indexes that are well understood and expected:

(1) As expected, APEX and the workload-aware Trie indexes perform equally well when answering a frequent label-path query, with a single index lookup. However, when the query is longer than k, the $\mathcal{P}[k]$-Trie evaluates the query as a group of sub-queries, each shorter or equal to k, and uses natural joins to compute the query result from the results of the sub-queries. The severeness of the impact depends on the k configuration of the index and the length of the query.

(2) When non-frequent label-path queries longer than k do not share a suffix with any label-path in the workload, its evaluation on the workload-aware Trie indexes is the same as on the $\mathcal{P}[k]$-Trie index. The query performance relies purely on the k value of the index that is available. As APEX indexes nodes rather than node pairs, it is only capable of evaluating sub-queries of length 0, resulting in much worse performance in this case.

(3) For queries that yield empty results, under most circumstances, the $\mathcal{AWP}[k]$-Trie and $\mathcal{W}[k]$-Trie are capable of identifying the empty queries in the *probe* process, while the other indexes have to resort to query evaluation, resulting in a significant decrease in their performance. Again APEX is penalized the most for resorting to sub-queries of length 0 and multiple join operations.

It is clear that the workload-aware Trie indexes we proposed work equally as well as APEX for frequent queries, is superb in identifying queries with empty results, and is at least as good as the $\mathcal{P}[k]$-Trie index for non-frequent queries that are not related to any frequent label-path. To better understand the unique merits of the workload-aware Trie indexes, we are more interested in the performance of non frequent queries that relate to frequent label-paths to a certain degree. This includes path queries that are not frequent themselves, but are related to a frequent label-path, and more complicated XPath queries that are composed of both frequent and non-frequent label-paths. We show the query performance of two representative queries on these indexes in Figures 10 and 11.

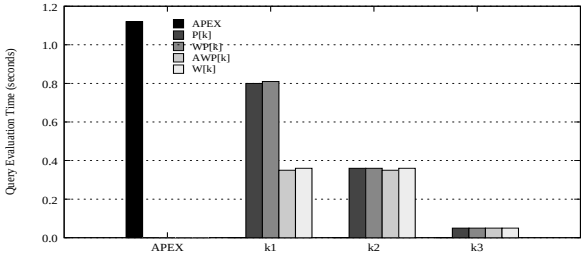

Figure 10: Query Performance q_1 (Case 3.2)

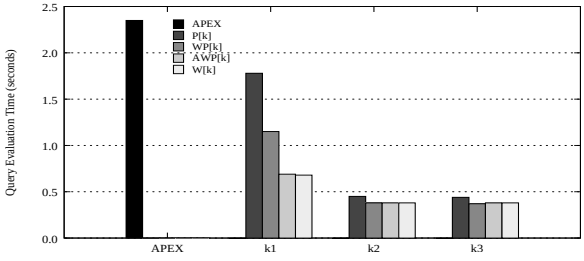

Figure 11: Query Performance q_2 (XPath query with branching predicates and both frequent and non-frequent label-paths

q_1 is a label-path query that is not frequent. In addition, the *structHit-suffix* of q_1 is not a frequent label-path itself. APEX is not able to answer the query directly, so again it has to perform the largest number of sub-query evaluations. Since the *structHit-suffix* of q_1 is not frequent, both the $\mathcal{P}[k]$ and $\mathcal{WP}[k]$-Trie index must decompose q_1 into sub-queries of length k, resulting in similar performances. However, a suffix of q_1 is included in the *complementary* set of label-paths for the $\mathcal{AWP}[k]$ and $\mathcal{W}[k]$-Trie, enabling a much longer label-path to be returned in the *probe*, hence less fragmentation in query decomposition and better performance.

q_2 presents an interesting case in which the query is an XPath query with branching predicates and contains a frequent and a non-frequent label-path. APEX is able to reproduce the parent-child relationships between nodes, but not beyond. Indeed, it is an index based on node partitions rather than node pair partitions. Thus, even if it is able to answer the frequent label-path sub-query in a single index lookup, it must still perform multiple join operations to relocate the ancestor nodes required for joining with the result of the other sub-query. The performance of evaluating this query with the $\mathcal{P}[k]$-Trie index depends on the k value, but it is guaranteed to be better than that of APEX, since we always require $k \geq 1$ for any $\mathcal{P}[k]$-Trie. The $\mathcal{WP}[k]$-Trie index outperforms the $\mathcal{P}[k]$-Trie by answering the frequent branch in a single index lookup. Meanwhile, the $\mathcal{AWP}[k]$ and $\mathcal{W}[k]$-Trie indexes are able to use sub-tree evaluation on the non-frequent sub-query, further improving query performance.

To better understand the space/performance trade-off we made in the design of the $\mathcal{W}[k]$-Trie index, we specifically tested a group of queries whose length is less than k but not equal to 1, to measure the impact of removing the top few layers of the Trie on the evaluation of such queries. Through

the use of sub-tree lookup operations, the $\mathcal{W}[k]$-Trie was found on average to be only 15% slower than the Trie indexes but was still 5% faster than APEX for theses cases, a reasonable compromise considering that for other cases the $\mathcal{W}[k]$-Trie index performs as well as the $\mathcal{AWP}[k]$-Trie, while being the most space efficient of the Trie index family.

7. CONCLUSIONS

In this paper, we introduce a family of workload-aware Trie indexes, their data structures, their support for query evaluation and optimization, as well as their construction and maintenance algorithms. The workload-aware indexes are capable of answering all frequent path queries with one index lookup and capable of answering all core XPath queries with index-only plans. The three indexes in the family differ in their space/performance trade-offs: the $\mathcal{WP}[k]$-Trie is simple and easy to construct and maintain, while the $\mathcal{AWP}[k]$-Trie is able to provide better performance, especially for non-frequent path queries. The $\mathcal{W}[k]$-Trie further optimizes space efficiency by considering additional query statistics, minimizing the index space requirements while retaining the query evaluation advantages present in the $\mathcal{AWP}[k]$-Trie. The workload-aware Trie indexes open the door for more complicated XPath query optimization strategies. Analyzing query history to generate a frequent label-path set, conducting cost-based optimizations, and estimating the result size of intermediate queries through the collection of statistical information are all future directions for the workload-aware Trie indexes.

8. REFERENCES

[1] M. Benedikt, *et al.* Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1), 2005.

[2] S. Brenes, *et al.* Trie Indexes for Efficient XML Query Evaluation. In *WebDB*, 2008.

[3] S. Chaudhuri, *et al.* Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, 2007.

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

[5] C.-W. Chung, *et al.* APEX: An Adaptive Path Index for XML Data. In *SIGMOD*, 2002.

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

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

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

[9] M. Hammer, *et al.* Index Selection in a Self-Adaptive Database Management System. In *SIGMOD*, 1976.

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

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

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

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

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

[15] NASA XML Group. NASA Data Set. http://xml.nasa.gov/xmlwg.

[16] K. Runapongsa, *et al.* XIST: An XML Index Selection Tool. In *XSym*, 2004.

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

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

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

[20] P. S. Yu, *et al.* On Workload Characterization of Relational Database Environments. *IEEE Trans. Softw. Eng.*, 18(4), 1992.