A Methodology for Coupling Fragments of XPath with Structural Indexes for XML Documents

Non Peer-reviewed author version

FLETCHER, George H L; VAN GUCHT, Dirk; WU, Yuqing; GYSSENS, Marc; BRENES, Sofia & PAREDAENS, Jan (2007) A Methodology for Coupling Fragments of XPath with Structural Indexes for XML Documents. In: Database Programming Languages, 11th International Symposium, DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers. p. 48-65.

DOI: 10.1007/978-3-540-75987-4_4
Handle: http://hdl.handle.net/1942/7970

A Methodology for Coupling Fragments of XPath with Structural Indexes for XML Documents

George H.L. Fletcher[1], Dirk Van Gucht[2], Yuqing Wu[2], Marc Gyssens[3], Sofía Brenes[2], and Jan Paredaens[4]

[1] Washington State University, Vancouver
gfletcher@acm.org
[2] Indiana University, Bloomington
{vgucht,yuqwu,sbrenesb}@cs.indiana.edu
[3] Hasselt University & Transnational University of Limburg
marc.gyssens@uhasselt.be
[4] University of Antwerp
jan.paredaens@ua.ac.be

1 Introduction

Supporting efficient access to XML data using XPath [3] continues to be an important research problem [6, 12]. XPath queries are used to specify node-labeled trees which match portions of the hierarchical XML data. In XPath query evaluation, indices similar to those used in relational database systems – namely, value indices on tags and text values – are first used, together with structural join algorithms [1, 2, 19]. This approach turns out to be simple and efficient. However, the structural containment relationships native to XML data are not directly captured by value indices.

To directly capture the structural information of XML data, a family of structural indices has been introduced. DataGuide [5] was the first to be proposed, followed by the 1-index [13], which is based on the notion of bi-simulation among nodes in an XML document. These indices can be used to evaluate some path expressions accurately without accessing the original data graph. Milo and Suciu [13] also introduced the 2-index and T-index, based on similarity of pairs (vectors) of nodes. Unfortunately, these and other early structural indices tend to be too large for practical use because they typically maintain too fine-grained structural information about the document [9, 16].

To remedy this, Kaushik et al. introduced the $A(k)$-index which uses a notion of bi-similarity on nodes relativized to paths of length k [10]. This captures localized structural information of a document, and can support path expressions of length up to k. Focusing just on local similarity, the $A(k)$-index can be substantially smaller than the 1-index and others.

Several works have investigated maintenance and tuning of the $A(k)$ indices. The $D(k)$-index [15] and $M(k)$-index [8] extend the $A(k)$-index to adapt to query workload. Yi et al. [18] developed update techniques for the $A(k)$-index and 1-index. Finally, the integrated use of structural and value indices has been explored [11], and there have also been investigations on covering indices [9, 16] and index selection [14, 17].

The introduction of structural indices for XML data has lead to significant improvements in the performance of XPath query evaluation. As was demonstrated empirically, the performance benefits of these indices are most dramatic

when queries "match" the index definitions [10]. To date, however, there lacks a formal understanding of this notion of queries matching indices. This leads to some fundamental questions about using structural indices in query evaluation:

1. For which fragments of XPath are particular structural indices *ideally* suited?
2. For these fragments, how are its expressions efficiently evaluated with the index?
3. Can the answers to these questions be bootstrapped to provide general techniques for evaluation of arbitrary XPath expressions?

In this paper, we present a methodology for investigating such questions and apply it to the important special case of the $A(k)$-indices. For question (1), we begin by noting that the $A(k)$-index of a document induces a partitioning on its nodes. Recently, an approach has been proposed for considering partitioning XML documents based on notions of query indistinguishability of nodes and paths, relative to particular fragments of XPath [7]. If we apply this approach to show that there exists a fragment of XPath which induces a partition identical to the $A(k)$-partition, then we can speak of an "ideal" match between the index and this fragment. Given this ideal coupling, we can then turn to a principled investigation of questions (2) and (3). A main contribution of this paper is the identification of such a fragment of XPath.

Before going into the technical details of the various steps we take in our methodology, we illustrate the general approach with a simple example coming from relational databases. Note that the results in this example are well-known, and as such do not add to the results of this paper.

1.1 A Motivating Example

Consider the B$^+$-tree index on a column A of a relation R [4]. Clearly, this index induces a partition on the tuples of R: tuples t_1 and t_2 in R will be in the same partition block[5] if and only if $t_1(A) = t_2(A)$. We will call this partition the B^+-*tree-partition* on column A of R, and denote it as $\mathsf{Btree}(A, R)$. (For emphasis, observe that a B$^+$-tree *index* on A of R is different than the $\mathsf{Btree}(A, R)$-*partition*. The first is a tree data structure, whereas the second is a partition on R.)

Next, consider the relational algebra, and in particular its sub-algebra consisting of the *range queries*. In this example, we focus on such queries as they are specified on attribute A of R. We will denote this class of queries by $\mathsf{RangeQ}(A, R)$. Its queries are of the form $\sigma_{((a_1 \leq A \leq a_2)\ \texttt{or}\ \cdots\ \texttt{or}\ (a_{2n-1} \leq A \leq a_{2n}))}(R)$.[6]
The $\mathsf{RangeQ}(A, R)$ algebra defines a partition on R, called the $\mathsf{RangeQ}(A, R)$-*partition* of R, and is defined as follows: tuples t_1 and t_2 in R are placed in the same block of the $\mathsf{RangeQ}(A, R)$-partition if for *any* query Q in $\mathsf{RangeQ}(A, R)$, $t_1 \in Q(R)$ if and only if $t_2 \in Q(R)$. In other words, t_1 and t_2 can not be distinguished by any query in $\mathsf{RangeQ}(A, R)$, i.e., either t_1 and t_2 are both in $Q(R)$, or they are both not in $Q(R)$. An important property of the $\mathsf{RangeQ}(A, R)$-partition is that for each query $Q \in \mathsf{RangeQ}(A, R)$, their exists a subset of blocks in the partition such that $Q(R)$ is the union of these blocks.

[5] "Block" stands for an element of a partition, not be confused with a block on a disk.
[6] For simplicity, we will assume that all the a_i values occur in the A-column of R.

A natural question that arises now is to ask if the Btree-partition and the RangeQ-partition are the same. It should come as no surprise that this is indeed the case. This is captured in the following theorem.

Theorem 1. [Btree-RangeQ **Coupling Theorem**] *Let R be a relation and let A be one of its attributes. The* $\mathsf{Btree}(A, R)$*-partition and the* $\mathsf{RangeQ}(A, R)$*-partition are the same.*

Proof. We give a proof of this statement, not because it is difficult, but because its structure reveals the strategy that we will follow to prove an analogous theorem for the XML case (Theorem 4).

1. Let tuples t_1 and t_2 be in the same block of the $\mathsf{Btree}(A, R)$-partition. Then, by definition, $t_1(A) = t_2(A)$. Consider now an arbitrary range query Q. Then clearly, if $t_1(A)$ (and therefore also $t_2(A)$) is in the range of Q then t_1 and t_2 are both in $Q(R)$, but if $t_1(A)$ is not in the range of Q, then they are both not in $Q(R)$. Consequently, t_1 and t_2 are in the same block of the $\mathsf{RangeQ}(A, R)$-partition.
2. Let tuples t_1 and t_2 be in different blocks of the $\mathsf{Btree}(A, R)$-partition. Then, by definition, $t_1(A) \neq t_2(A)$. Let $a = t_1(A)$. Then the range query $\mathtt{label}_a := \sigma_{A=a}(R)$ has t_1 in its result, but not t_2. Thus t_1 and t_2 are in different blocks of the $\mathsf{RangeQ}(A, R)$-partition, and the proof is done.

An immediate consequence of Theorem 1 is that each range query evaluated on R is equal to the union of a family of blocks of the $\mathsf{Btree}(A, R)$-partition.

Theorem 2. [Btree-RangeQ **Block-Union Theorem**] *Let R be a relation, let A be one of its attributes, and let $Q \in \mathsf{RangeQ}(A, R)$. Then there exists a class $\mathcal{B}_Q$ of partition blocks of the* $\mathsf{Btree}(A, R)$*-partition such that $Q(R) = \bigcup_{B \in \mathcal{B}_Q} B$.*

Note that the Btree-RangeQ Block-Union Theorem can provide guidance and insight in the processing of queries in richer relational fragments.

In the second part of the proof of Theorem 1, observe that the range query $\mathtt{label}_a$ has the property that it uniquely identifies the block of the $\mathsf{RangeQ}(A, R)$-partition consisting of the tuples of R that are indistinguishable from t_1 by any query in $\mathsf{RangeQ}(A, R)$. We will call the query $\mathtt{label}_a$, a *labeling query* and its defining a-value a *label*. Now as a consequence of Theorem 2 we have that evaluating a range query $Q \in \mathsf{RangeQ}(A, R)$ can be done by forming a union of such labeling expressions applied to R.

Theorem 3. [Btree-RangeQ **Label-Union Theorem**] *Let R be a relation and A one of its attributes. Then for each query $Q \in \mathsf{RangeQ}(A, R)$, there is a set of labeling queries $\mathcal{L}_Q \subseteq \mathsf{RangeQ}(A, R)$ such that $Q(R) = \bigcup_{\mathtt{label} \in \mathcal{L}_Q} \mathtt{label}(R)$.*

Obviously, in practice we do not want to evaluate the labeling queries $\mathtt{label} \in \mathcal{L}_Q$ directly on R, but rather we would want a data structure that stores each result $\mathtt{label}(R)$. If such a data structure supports efficient look-up of the tuples in the partition block associated with each labeling expression $\mathtt{label}$, then evaluating Q can be done by simply streaming out these tuples. Of course, such a data structure is the B^+-tree index. So, in a formal sense we have shown that range queries match ideally with B^+-tree indexes, which of course is a well-known fact.

1.2 Paper Overview

We proceed as in this motivating example, for structural indices and the XPath query language. Specifically:

- We introduce the family of $P(k)$-partitions, which are derivatives of the family of $A(k)$-partitions. It turns out that this new class of partitions is fundamental for establishing the results which follow.
- We then introduce a family of upward XPath algebras, $\mathscr{U}(k)$, and show that the $P(k)$-partition and the partition induced by the $\mathscr{U}(k)$ algebra are the same. As a consequence of this we have that the evaluation of a $\mathscr{U}(k)$ query is equal to the union of some blocks of the $P(k)$-partition.
- Based on this result, we then develop guidelines for the use of a $P(k)$-partition in the evaluation of general XPath queries.
- Following this, we show that for each block in the $P(k)$-partition a labeling expression in $\mathscr{U}(k)$ can be constructed which uniquely identifies the block. Thus, we conclude that each query in $\mathscr{U}(k)$ can be rewritten as the union of some $\mathscr{U}(k)$ block labeling expressions.

These results indicate research directions into new data structures to support efficient evaluation of general XPath queries.

2 Coupling Indices and XPath Fragments

In this section, we set out to apply the methodology described in the motivating relational example to the XML case.

2.1 The XML data model

We begin by introducing the document data model that will be used in this paper. Our data model is a simplified version of the XML data model wherein we view a document as a labeled tree.

Definition 1. *A* document D *is a 4-tuple* (V, Ed, r, λ)*, with* V *the finite set of nodes,* $Ed \subseteq V \times V$ *a tree of parent-child edges,* $r \in V$ *the root, and* $\lambda\colon V \to \mathcal{L}$ *a node-labeling function into a countably infinite set of labels* $\mathcal{L}$*.*

Given a document, it is useful to introduce the concept of its paths. We define the set of *paths* of a document D, denoted $\mathsf{Paths}(D)$, as the set $V \times V$. So, for us a path is not a sequence of nodes, but rather a pair. This makes sense however, since a pair of nodes $(n, m) \in \mathsf{Paths}(D)$ identifies the unique path from node n to node m in D. The set of *downward-paths*, $\mathsf{DownPaths}(D)$, consists of the paths (n, m) where n is an ancestor of m. Similarly, the set of *upwards-paths*, $\mathsf{UpPaths}(D)$, consists of the paths (n, m) where n is a descendant of m. Furthermore, for $k \in \mathbb{N}$, $\mathsf{DownPaths}(D, k)$ ($\mathsf{UpPaths}(D, k)$) are those paths in $\mathsf{DownPaths}(D)$ (in $\mathsf{UpPaths}(D)$, respectively) of length at most k. For example, in document D of Figure 1 the path (n_1, n_1) is a member of both $\mathsf{DownPaths}(D, 0)$ and $\mathsf{UpPaths}(D, 0)$, the paths (n_1, n_1), (n_1, n_4), and (n_1, n_9) are in $\mathsf{DownPaths}(D, 2)$, and their corresponding inverse paths (n_1, n_1), (n_4, n_1), and (n_9, n_1) are in $\mathsf{UpPaths}(D, 2)$. The paths (n_9, n_{12}) and (n_1, n_{19}) are in neither $\mathsf{DownPaths}(D, 2)$ nor $\mathsf{UpPaths}(D, 2)$.

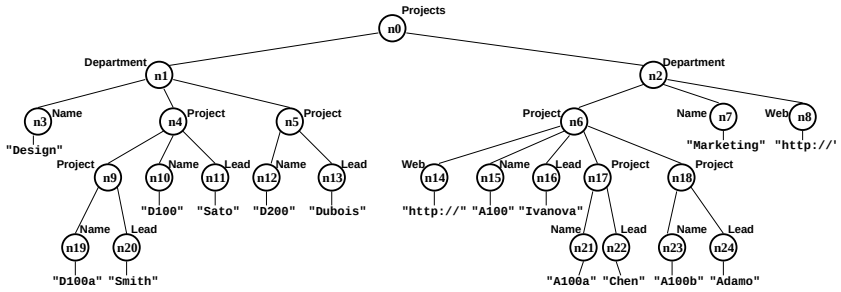

Fig. 1. An XML document. For reference, non-leaf nodes are given unique IDs.

2.2 The $A(k)$-partition of a Document

Given a labeled semi-structured document[7] and a natural number k, Kaushik et al. [10] introduced the $A(k)$-*index* for this document.

The index is built on the partition induced by a certain bi-similarity equivalence relation on the nodes in the document. When specialized to a document, as defined here, the definition of this bi-similarity equivalence is as follows.

Definition 2. *Let $D = (V, Ed, r, \lambda)$ be a document, $n_1, n_2 \in V$, and let $k \in \mathbb{N}$. We say that n_1 and n_2 are* $A(k)$-equivalent *in D, denoted $n_1 \equiv_{A(k)} n_2$, if*

1. *$\lambda(n_1) = \lambda(n_2)$; and*
2. *if $k \geq 1$ then*
   (a) *n_1 has a parent in D if and only if n_2 has a parent in D; and*
   (b) *if n_1 has parent p_1 and n_2 has parent p_2, then $p_1 \equiv_{A(k-1)} p_2$.*

We call the partition induced by $\equiv_{A(k)}$ on V the $A(k)$-partition *of D.*

A more intuitive reading of this definition is that nodes n_1 and n_2 belong to the same block of the $A(k)$-partition, if the label sequences associated with their incoming paths in D of length at most k are the same. Also note that the $A(k+1)$-partition of a document is a refinement of the $A(k)$-partition.

Example 1. Figure 2 illustrates (ignoring for now the edges between the blocks), for $k = 0, 1$, and 2, the $A(k)$-partition of the Design Department sub-tree rooted at node n_1 in the document of Figure 1.

Following Kaushik et al. [10], the $A(k)$-*index* of a document D is a *graph* wherein each node is a block of the $A(k)$-partition of D, and an edge exists from a block B_1 to a block B_2 if there exists a parent-child edge in D from a node in B_1 to a node in B_2. So, the $A(k)$-index can be thought of as a representation of the $A(k)$-partition and how its blocks can be related in accordance with the document D. The $A(k)$-indexes for $k = 0, 1, 2$ are visualized in Figure 2 on the Design Department sub-tree of the document of Figure 1. Note that if k is equal to the height of the document, then the $A(k)$-index corresponds to the 1-index proposed by Milo and Suciu [13] and the strong DataGuide proposed by Goldman and Widom [5].

[7] A semi-structured document does not need to be a tree. In particular, it is possible that a node has multiple parents.

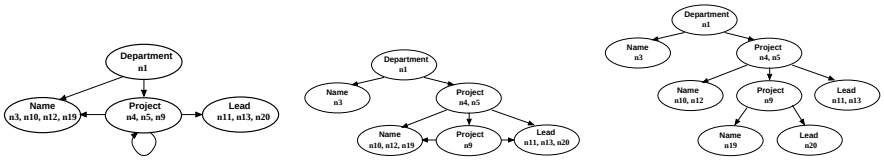

Fig. 2. $A(k)$-indices ($k = 0, 1, 2$) associated with their corresponding $A(k)$-partitions for the "Design" Department sub-tree in the document of Figure 1.

2.3 The $P(k)$-partition of a Document

The $A(k)$-partitions of a document D are partitions on its *nodes*. We will need another family of partitions, the $P(k)$-partitions, which, rather than being defined on nodes, are defined on the sets $\mathsf{UpPaths}(D, k)$, i.e., the sets of *upward-paths* of D of length at most k. As we will see, the $P(k)$-partitions are more fundamental than the $A(k)$-partitions for developing our results.

Definition 3. *Let D be a document, let $k \in \mathbb{N}$, and let (n_1, m_1) and (n_2, m_2) be two paths in $\mathsf{UpPaths}(D, k)$. We say that (n_1, m_1) and (n_2, m_2) are* $P(k)$-equivalent*, denoted $(n_1, m_1) \equiv_{P(k)} (n_2, m_2)$, if*

1. $n_1 \equiv_{A(k)} n_2$*; and*
2. $\mathtt{length}(n_1, m_1) = \mathtt{length}(n_2, m_2)$.[8]

We call the partition induced by $\equiv_{P(k)}$ on $\mathsf{UpPaths}(D, k)$ the $P(k)$-partition *of D.*

Example 2. Consider the sub-tree D' in the document of Figure 1 rooted at n_4. For $k = 0, 1$, we have that

1. the $P(0)$-partition on D' is the set $\{[(n_{19}, n_{19}), (n_{10}, n_{10})], [(n_{20}, n_{20}), (n_{11}, n_{11})], [(n_9, n_9), (n_4, n_4)]\}$; and
2. the $P(1)$-partition on D' is the set $\{[(n_{19}, n_{19}), (n_{10}, n_{10})], [(n_{20}, n_{20}), (n_{11}, n_{11})], [(n_9, n_9)], [(n_4, n_4)], [(n_{19}, n_9), (n_{10}, n_4)], [(n_{20}, n_9), (n_{11}, n_4)], [(n_9, n_4)]\}$. Notice that the block $[(n_9, n_9), (n_4, n_4)]$ of the $P(0)$-partition is split into two blocks of the $P(1)$-partition, namely $[(n_9, n_9)]$ and $[(n_4, n_4)]$. This is because $n_9 \equiv_{A_0} n_4$, but $n_9 \not\equiv_{A_1} n_4$.

Finally, we wish to observe that when k is equal to the height of a document D, then the $P(k)$-partition corresponds to the partitions induced by the 2-index on D proposed by Milo and Suciu [13].

2.4 The XPath-Algebra

We next present an algebraization [7] of the logical navigational core of XPath [6] which we adopt in this paper and define the paths and nodes-semantics of expressions in this algebra.

[8] As should be clear, $\mathtt{length}(n, m)$ denotes the length of the path in D from node n to node m.

$$\varepsilon(D) = \{(n,n) \mid n \in V\} \qquad E_1 \cup E_2(D) = E_1(D) \cup E_2(D)$$
$$\emptyset(D) = \emptyset \qquad E_1 \cap E_2(D) = E_1(D) \cap E_2(D)$$
$$\ell(D) = \{(n,n) \mid m \in V \text{ and } \lambda(n) = \ell\} \qquad E_1 - E_2(D) = E_1(D) - E_2(D)$$
$$\downarrow(D) = Ed$$
$$\uparrow(D) = Ed^{-1}$$
$$E_1 \diamond E_2(D) = \{(n,m) \mid \exists w\colon (n,w) \in E_1(D)\ \&\ (w,m) \in E_2(D)\}$$
$$E_1[E_2](D) = \{(n,m) \in E_1(D) \mid \exists w\colon (m,w) \in E_2(D)\}$$

Table 1. The XPath-Algebra Path-Semantics

Definition 4. *The* XPath-algebra *consists of the primitives* $\varepsilon, \emptyset, \ell, \downarrow$, *and* $\uparrow$ *together with the operations on expressions* $E_1 \diamond E_2$, $E_1[E_2]$, $E_1 \cup E_2$, $E_1 \cap E_2$, *and* $E_1 - E_2$. *Given a document* $D = (V, Ed, r, \lambda)$, *the* semantics *of an XPath-algebra expression* E *on* D, *denoted* $E(D)$, *is a subset of* $\mathsf{Paths}(D)$. *The semantics for each primitive and each operation is given in Table 1.*

The XPath-algebra semantics reflects a "global" perspective of expressions being evaluated on an entire document. There is also a "local" semantic perspective, in which expressions are viewed as working at a particular node, as follows.

Definition 5. *Let* E *be an XPath-algebra expression and let* $D = (V, Ed, r, \lambda)$ *be a document. For* $n \in V$, *the* local semantics *of* E *on* D *at* n, *denoted* $E(D)(n)$, *is the set* $\{m \in V \mid (n,m)) \in E(D)\}$.

Consider the XPath query /`Projects`/`Department`/`Project`[./`Project`] that retrieves all the projects of departments that have a sub-project. When applied to the document D of Figure 1, this query returns the set of nodes $\{n_4, n_6\}$. An XPath-algebra expression corresponding to this query can be formulated as `Projects` $\diamond \downarrow \diamond$ `Department` $\diamond \downarrow \diamond$ `Project`[$\downarrow \diamond$ `Project`]. According to the semantics of the XPath-algebra, the global semantics of this expression on D is the set of paths $\{(n_0, n_4), (n_0, n_6)\}$ whereas its local semantics at the root node n_0 is the set of nodes $\{n_4, n_6\}$, which, as intended, corresponds to the result set of the original XPath query.

2.5 Linking the $P(k)$-partition to the XPath Algebra

The $A(k)$-indexes were introduced to support efficient evaluation of certain path queries on XML documents. As was demonstrated empirically on a benchmark of queries, the performance benefits of these indexes were most dramatic when the queries "matched" the index definitions [10]. However, in that paper the concept of queries matching indexes was not formalized. A main theme of this paper is that we can indeed formalize this concept. More specifically, in the remainder of this section we identify a class $\mathscr{U}(k)$ of sub-algebras of the XPath-algebra whose queries ideally match up with the $P(k)$-partitions (and as such with the $A(k)$ indexes). The central idea behind this formalization comes from showing that the $P(k)$-partitions are identical to the partitions induced on the document by the $\mathscr{U}(k)$ algebras. These language induced partitions are defined using equivalence relations that define a pair of paths equivalent when they can

not be distinguished by the queries of the sub-algebras. i.e., they are either both in the answer of a query, or they are both not. Intuitively, such pairs are always processed together during query evaluation.

In the following two sections, we define the $\mathscr{U}(k)$ sub-algebras and show how the $P(k)$-partitions are identical to partitions induced by these algebras.

2.6 The $\mathscr{U}(k)$-algebras and their Associated $\mathscr{U}(k)$-partitions

In the example of Section 1.1, we considered the class of RangeQ relational queries and introduced the notion of RangeQ-partitions. In this section, we define the $\mathscr{U}(k)$ XPath-algebras, and then, in analogy with this example, define the associated $\mathscr{U}(k)$-partitions.

Definition 6. *We recursively define the* upward-k XPath algebras*, $\mathscr{U}(k)$ for each $k \in \mathbb{N}$, as follows. (Notice that the $\downarrow$ primitive can not be used in expressions of these algebras).*

1. *$\mathscr{U}(0)$ is the set of XPath-algebra expressions without occurrences of the "$\downarrow$" and "$\uparrow$" primitives.*
2. *For $k \geq 1$, $\mathscr{U}(k)$ is the smallest set of expressions satisfying*
   (a) *if $E \in \mathscr{U}(k-1)$, then $E \in \mathscr{U}(k)$;*
   (b) *$\uparrow \in \mathscr{U}(k)$;*
   (c) *if $E_1 \in \mathscr{U}(k)$ and $E_2 \in \mathscr{U}(k)$, then $E_1 \star E_2 \in \mathscr{U}(k)$, for $\star = \cup, \cap, -$; and*
   (d) *if $E_1 \in \mathscr{U}(k_1)$ and $E_2 \in \mathscr{U}(k_2)$, and $k_1 + k_2 \leq k$, then $E_1 \diamond E_2 \in \mathscr{U}(k)$ and $E_1[E_2] \in \mathscr{U}(k)$.*

Example 3. As an example of $\mathscr{U}(k)$ expressions, note that `Name` $\diamond \uparrow \diamond$ `Project` $\diamond \uparrow \diamond$ `Project` is in $\mathscr{U}(2)$ but not in $\mathscr{U}(1)$, the expression $\uparrow \diamond$ `Department` is in $\mathscr{U}(1)$ but not in $\mathscr{U}(0)$, and the combined expression `Name` $\diamond \uparrow \diamond$ `Project` $\diamond \uparrow \diamond$ `Project`$[\uparrow \diamond$ `Department`$]$ is in $\mathscr{U}(3)$ but not in $\mathscr{U}(2)$.

The following useful proposition about the $\mathscr{U}(k)$-algebras can be shown by a simple inductive argument.

Proposition 1. *Let D be a document, $k \in \mathbb{N}$, and $E \in \mathscr{U}(k)$. Then $E(D) \subseteq$ $\mathsf{UpPaths}(D, k)$.*

We are now ready to define the partitions associated with the $\mathscr{U}(k)$-algebras. Proposition 1 motivates us defining these partitions on $\mathsf{UpPaths}(D, k)$, just as with the $P(k)$-partitions. In the next section we will then show that it is these partitions that are identical with the $P(k)$-partitions.

Recall from the relational example, that we associated the RangeQ query language with the RangeQ-partition. This partition was defined such that each of its blocks grouped those tuples in a relation that could not be distinguished by the queries in RangeQ. We define the partitions associated with the $\mathscr{U}(k)$-algebras analogously.

Definition 7. *Let $D = (V, Ed, r, \lambda)$ be a document, and $k \in \mathbb{N}$. We say two paths (n_1, m_1) and (n_2, m_2) in* $\mathsf{UpPaths}(D, k)$ *are* $\mathscr{U}(k)$-equivalent*, denoted $(n_1, m_1) \equiv_{\mathscr{U}(k)} (n_2, m_2)$, if for any expression E in $\mathscr{U}(k)$, it is the case that $(n_1, m_1) \in E(D)$ if and only if $(n_2, m_2) \in E(D)$. We call the partition induced by $\equiv_{\mathscr{U}(k)}$ on* $\mathsf{UpPaths}(D, k)$ *the* $\mathscr{U}(k)$-partition *of D.*

2.7 The Coupling of $P(k)$ and $\mathscr{U}(k)$

We establish a coupling theorem for the $P(k)$ and $\mathscr{U}(k)$ partitions, in analogy to Theorem 1, as follows.

Theorem 4. [**Coupling Theorem**] *Let D be a document and $k \in \mathbb{N}$. The $P(k)$-partition of D and the $\mathscr{U}(k)$-partition of D are the same.*

Proof. (Sketch) Compared to the proof that shows that the Btree-partition and the RangeQ-partition are the same, the proof of Theorem 4 is considerably more involved. Nevertheless, the proof follows the same strategy. First, we show that if two paths (n_1, m_1) and (n_2, m_2) in $\mathsf{UpPaths}(D, k)$ are in the same block of the $P(k)$-partition, then they are also together in a block of the $\mathscr{U}(k)$-partition. In particular, we show by induction that for each expression $E \in \mathscr{U}(k)$, it is the case that $(n_1, m_1) \in E(D)$ if and only if $(n_2, m_2) \in E(D)$. The proof of this fact is given in the Appendix. Second, we show that if (n_1, m_1) and (n_2, m_2) are in two different blocks of the $P(k)$-partition, then they are also in two different blocks of the $\mathscr{U}(k)$-partition. This is shown by constructing an expression $\texttt{label} \in \mathscr{U}(k)$ such that $(n_1, m_1) \in \texttt{label}(D)$, but $(n_2, m_2) \notin \texttt{label}(D)$. The expression $\texttt{label}$ is of independent interest since it can be shown that it uniquely identifies (i.e., labels) the block of the $\mathscr{U}(k)$-partition of which (n_1, m_1) is a member. More precisely, $\texttt{label}(D)$ consists of exactly those paths in $\mathsf{UpPaths}(D, k)$ that can not be distinguished from (n_1, m_1) by the $\mathscr{U}(k)$-algebra. Section 4 is devoted to the existence and the construction of $\texttt{label}$.

As an immediate consequence, each $\mathscr{U}(k)$ query evaluated on a document D is equal to the union of a family of blocks of the $P(k)$-partition of D.

Theorem 5. [**Block-Union Theorem**] *Let D be a document, $k \in \mathbb{N}$, and $Q \in \mathscr{U}(k)$. Then there exists a class $\mathcal{B}_Q$ of partition blocks of the $P(k)$-partition of D such that $Q(D) = \bigcup_{B \in \mathcal{B}_Q} B$.*

In analogy with Theorem 2, the Block-Union Theorem provides insight into the processing of general XPath-algebra queries, as we see next.

3 XPath Query Evaluation with $P(k)$-Partitions

The results of Section 2 speak to answering $\mathscr{U}(k)$ queries directly using the $P(k)$-partition structure. In this section we consider the evaluation of general XPath algebra expressions and show how the results of Section 2 concerning the coupling between the $\mathscr{U}(k)$ and $P(k)$-partitions can be utilized in this case. Given an XPath expression and a $P(k)$-partition, the main idea is to identify

its $\mathscr{U}(k)$ sub-expressions or those that are easily converted to $\mathscr{U}(k)$ expressions using rewrite rules. For each such expression, we are then guaranteed by the Block-Union Theorem that its value is the union of an appropriate set of blocks of the $P(k)$-partition. If we then have a method to quickly identify and return partition blocks, we will have an efficient way of evaluating these expressions. We return to this issue in the next section. In this section, we focus on the development of general techniques for using $P(k)$-partitions in the evaluation of arbitrary XPath algebra expressions.

3.1 Evaluating Upward Expressions

If our given XPath expression is in fact a member of $\mathscr{U}(k)$ then no decomposition is necessary. However, if we consider a $\mathscr{U}(j)$ expression of the form $E = A_1 \diamond \uparrow \diamond \ldots \diamond \uparrow \diamond A_j$ where $j > k$, then such a query is not directly supported by the $P(k)$-partition. Nevertheless, we can decompose it into sub-expressions that are in $\mathscr{U}(k)$. For example, consider the $P(2)$-partition available and the expression $E_1 = A_1 \diamond \uparrow A_2 \diamond \uparrow \diamond A_3 \diamond \uparrow \diamond A_4$ in $\mathscr{U}(4)$, then E_1 contains sub-expressions $F_1 = A_1 \diamond \uparrow \diamond A_2 \uparrow \diamond A_3$, and $F_2 = A_3 \diamond \uparrow \diamond A_4$ which are both in $\mathscr{U}(2)$. As such, they can be directly evaluated using the $P(2)$-partition as $E_1(D) = F_1(D) \bowtie F_2(D)$.

3.2 Evaluating Downward Expressions

In practice, most XPath expressions use navigation just along the parent-child ($\downarrow$) axis. Consider the XPath sub-algebra $\mathscr{D}$ which is defined as the set of all XPath expressions in which the $\uparrow$ primitive does not appear (and the $\mathscr{D}(k)$ algebras defined analogously to the $\mathscr{U}(k)$ algebras). For such queries, we cannot directly utilize the Block-Union Theorem. However, we can convert downward navigation into upward navigation by using a technique which we will refer to as "inverting expressions." We will illustrate this technique on downward expressions with and without predicate operations. For this discussion, we consider downward expressions to be in the $\mathscr{D}(k)$-algebra which is defined in complete analogy with $\mathscr{U}(k)$, except that the $\downarrow$ primitive is permitted, but not the $\uparrow$ primitive.

Downward Expressions without Predicates Downward expressions without predicates can be "inverted" into expressions in corresponding upward expressions without predicates using the rewrite rules shown in Table 2.

So, given a downward expression $E \in \mathscr{D}(k)$ without predicates, we can rewrite E into E^{-1} which is in $\mathscr{U}(k)$ and also has no predicates. We can then obtain $E(D)$ by first computing $E^{-1}(D)$ and then considering its inverted result. Now since E^{-1} is an expression in $\mathscr{U}(k)$, we can directly apply the evaluation techniques for $\mathscr{U}(k)$ expressions discussed above.

$E \rightarrow E^{-1}$
$\epsilon \rightarrow \epsilon$
$\emptyset \rightarrow \emptyset$
$\downarrow \rightarrow \uparrow$
$\hat{\lambda} \rightarrow \hat{\lambda}$
$E_1 \cup E_2 \rightarrow E_1^{-1} \cup E_2^{-1}$
$E_1 \cap E_2 \rightarrow E_1^{-1} \cap E_2^{-1}$
$E_1 - E_2 \rightarrow E_1^{-1} - E_2^{-1}$
$E_1 \diamond E_2 \rightarrow E_2^{-1} \diamond E_1^{-1}.$

Table 2. Inversion Rewrite Rules for $\mathscr{D}$.

Downward Expressions with Predicates Now consider the evaluation of downward algebra expressions wherein predicate operations occur. A simple example is the expression $E_2 = \downarrow [\downarrow]$. Applied to a document, E_2 evaluates to the document's parent-child pairs for children that have at least one child themselves. As above, to evaluate E_2 on a document D, we could consider the concept of inverting E_2 into an expression $E_2^{-1} \in \mathscr{U}(2)$ such that $E_2(D) = (E_2^{-1}(D))^{-1}$. This approach does not work here because the inversion rules in Table 2 do not extend to include the predicate operation. In fact, we can construct a document D_2 such that for *each* expression $F \in \mathscr{U}(2)$, $E_2(D_2) \neq F(D)^{-1}$.

Clearly E_2 is equivalent to the XPath-algebra expression $\downarrow \diamond \downarrow \diamond \uparrow$.[9] Notice that this expression is neither a downward nor an upward expression. However its sub-expression $G_1 = \downarrow \diamond \downarrow$ is in $\mathscr{D}(2)$ and its sub-expression $G_2 = \uparrow$ is in $\mathscr{U}(1)$. Using the inversion technique described in Section 3.2 applied to G_1, the evaluation of $E_2(D)$ can be accomplished by computing the relation $(G_1^{-1}(D))^{-1} \bowtie G_2(D)$, and, as indicated in this section, the evaluations of $G_2(D) = \uparrow (D)$ and $G_1^{-1}(D) = \uparrow \diamond \uparrow (D)$ can be done by utilizing the Block-Union Theorem for the $P(2)$ and $P(1)$-partitions respectively.

Given that the selectivity of a longer path is no larger than that of short sub-paths of the path, evaluating G_1 reduces the search space to the minimum that can be obtained on such a chain expression. Starting from any given node, upward navigation in an XML data tree, unlike downward navigation, has one and only one route to follow, which is to reach its parent. Therefore, it is reasonable to claim that the result of $G_1^{-1}(D)$ is substantially smaller than that of $G_2(D)$, and the $\bowtie$ operation can be further optimized by $G_1^{-1}(D)$ followed by an upward navigation.

We will now consider a slightly more complicated downward expression $E_3 \in \mathscr{D}(3)$ which retrieves information about leaders of projects that have a subproject: $E_3 = \texttt{Department} \diamond \downarrow \diamond \texttt{Project}[\downarrow \diamond \texttt{Project}] \diamond \downarrow \diamond \texttt{Lead}$. E_3 can be represented as an expression pattern tree, as illustrated in Figure 3(a). The shaded node can be interpreted as the "answer" of E_3.

Assume that the $P(2)$-partition is available. Then, as shown in Figure 3(b), there are two natural chains of length 2 present in the pattern tree of E_3: G_1 and G_2. There are also natural chains of length 1 as shown in Figure 3(c): G_3, G_4, and G_5.

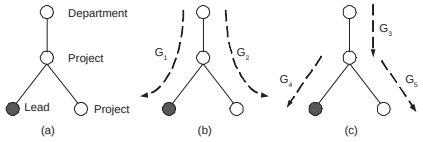

Fig. 3. Chain pattern tree for $\boldsymbol{E_3}$.

Using G_1, G_2, and G_4, the expression E_3 is equivalent to the expression H_1 defined as follows: $H_1 = ((G_1 \diamond \uparrow) \cap (G_2 \diamond \uparrow)) \diamond G_4$, and therefore, for a document D, $E_3(D)$ can be computed as follows:

$$E_3(D) = \begin{pmatrix} ((G_1^{-1}(D))^{-1} \bowtie \uparrow (D)) \\ \cap \\ ((G_2^{-1}(D))^{-1} \bowtie \uparrow (D)) \end{pmatrix} \bowtie (G_4^{-1}(D))^{-1}.$$

[9] Incidentally, it can be shown that each expression in the $\mathscr{D}(k)$ algebra can be converted into an alternating composition of $\mathscr{D}(k)$ and $\mathscr{U}(k)$ expressions all of which do **not** use predicates.

$$\mathtt{L}_{k,n,i} = \begin{cases} L_0 \diamond \uparrow \diamond L_1 \diamond \cdots \diamond \uparrow \diamond L_i[\uparrow \diamond L_{i+1} \ \diamond \cdots \diamond \uparrow \diamond L_\ell] & \text{if } i < \ell \\ L_0 \diamond \uparrow \diamond L_1 \diamond \cdots \diamond \uparrow \diamond L_\ell \diamond \uparrow^{i-\ell} & \text{if } i \geqslant \ell \end{cases}$$

Fig. 4. The $\boldsymbol{i^{\text{th}}}$ $\boldsymbol{k}$-ancestor label expression of node $\boldsymbol{n}$ having $\boldsymbol{k}$-ancestor label path $\boldsymbol{L_0, \ldots, L_\ell}$.

All sub-expressions in this transformed expression of E_3 are in $\mathscr{U}(2)$, and hence, as already discussed, can be evaluated using the Block-Union Theorem for $P(2)$.

Now assume that only the $P(1)$-partition is available. In this case, the longest path expressions that can take advantage of the partitions are those of length at most 1. Such expressions are G_3, G_4 and G_5. Using these sub-expressions, E_3 is equivalent with the expression H_2 defined as follows:

$$H_2 = (((G_3 \diamond G_4) \diamond \uparrow) \cap ((G_3 \diamond G_5) \diamond \uparrow)) \diamond G_4.$$

We have just observed how the Block-Union Theorem assists in the evaluation of XPath expressions. However, if we want to make efficient utilization of these ideas, we will need techniques for quickly identifying the $P(k)$-partition blocks associated with a query. We turn to this issue in the following section.

4 Labeling $P(k)$-partition Blocks

In Section 2 we investigated the *semantic* relationship between $\mathscr{U}(k)$-equivalence and the $P(k)$-partition. There is also an alternative *syntactic* characterization of this relationship which is critical in identifying the $P(k)$-partition blocks used in evaluating a query. In particular, we have that evaluation of a $\mathscr{U}(k)$ query on a document D can be done by forming a union of partition block labeling expressions applied to D, similarly to Theorem 3 for the range queries.

Theorem 6. [Label-Union Theorem] *Let D be a document and $k \in \mathbb{N}$. Then for each query $Q \in \mathscr{U}(k)$, a set of labeling queries $\mathcal{L}_Q \subseteq \mathscr{U}(k)$ can be constructed such that $Q(D) = \bigcup_{\mathtt{label} \in \mathcal{L}_Q} \mathtt{label}(D)$.*

The Label-Union Theorem is a crucial syntactic link between $P(k)$-partitions and the semantics of $\mathscr{U}(k)$ expressions, and is an immediate corollary of Theorem 5 and the following result.

Proposition 2. *Let D be a document and $k \in \mathbb{N}$. For each block B of the $P(k)$-partition of D, an expression $\mathtt{label}_B \in \mathscr{U}(k)$ can be constructed such that for each pair $(n, m) \in \mathsf{UpPaths}(D, k)$ it is the case that $(n, m) \in B$ if and only if $(n, m) \in \mathtt{label}_B(D)$.*

We now proceed with the proof of Proposition 2. First, we define in two steps the labeling expressions for partition blocks. Then, we make precise the relationship of these expressions to the partition blocks.

Step 1: Ancestor Path Expressions

Definition 8. *Let $D = (V, Ed, r, \lambda)$ be a document, $k \in \mathbb{N}$, and $n \in V$. Let the* k-ancestor label path of n *be the list of labels $L_0, \ldots, L_\ell$ of the nodes on the path from n up towards the root node r, of length $\ell = \min\{k, \mathsf{length}(n, r)\}$. For $i \leqslant k$, the i^{th}* k-ancestor label expression of n *is the $\mathscr{U}(k)$ expression $\mathtt{L}_{k,n,i}$ defined in Figure 4.*[10]

We observe that all members of a $P(k)$ partition block share a k-ancestor label expression. Namely, for a block B, all elements share the expression $\mathtt{L}_{k,n,\mathsf{length}(n,m)}$, where (n, m) is any member of block B. This observation follows directly from the definition of $P(k)$-equivalence and Definition 8.

Example 4. Consider the $P(1)$-partition of the small document in Figure 5, wherein each node has label `A`:

$$\big\{[(n_a, n_a)], [(n_b, n_b), (n_c, n_c)], \\ [(n_c, n_b), (n_b, n_a)]\big\}.$$

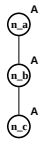

Fig. 5. Three-node document.

As noted above, we can associate with each block in this partition an $\mathtt{L}_{1,n,\mathsf{length}(n,m)}$ expression, for any element (n, m) in the block, as in Figure 6.

Note, however, that ancestor label expressions do *not* necessarily uniquely identify particular $P(k)$ blocks.

Partition Block	Expression
$[(n_a, n_a)]$	$\mathtt{L}_{1,n_a,0} = \mathtt{A}$
$[(n_b, n_b), (n_c, n_c)]$	$\mathtt{L}_{1,n_b,0} = \mathtt{A}[\uparrow \diamond \mathtt{A}]$
$[(n_c, n_b), (n_b, n_a)]$	$\mathtt{L}_{1,n_b,1} = \mathtt{A} \diamond \uparrow \diamond \mathtt{A}$

Fig. 6. Ancestor label expressions.

Example 5. Continuing Example 4, we note that expression $\mathtt{L}_{1,n_a,0} = \mathtt{A}$ for block $[(n_a, n_a)]$ evaluates on the document D as $\mathtt{A}(D) = \{(n_a, n_a), (n_b, n_b), (n_c, n_c)\}$, and hence does not uniquely identify its block. This is due to the fact that $\mathtt{L}_{1,n_a,0}$ is not selective enough. In particular, all blocks, with 1-ancestor label expressions having as a *prefix* expression $\mathtt{L}_{1,n_a,0}$, will also appear in the evaluation of $\mathtt{L}_{1,n_a,0}$. For example, the 1-ancestor labeling expression $\mathtt{A}[\uparrow \diamond \mathtt{A}]$ for block $[(n_b, n_b), (n_c, n_c)]$ has as a prefix the 1-ancestor labeling expression `A` for block $[(n_a, n_a)]$, and therefore both blocks appear in the evaluation of `A`. We pursue a remedy for this problem in the next step.

Step 2: Partition Labeling Expressions To tighten up ancestor label expressions, we need two tools. To compare these expressions, we introduce the following notion of expression prefixes.

Definition 9. *Let D be a document, $i, k \in \mathbb{N}$, $i \leqslant k$, and m and n be nodes in D. For i^{th} k-ancestor label expressions $\mathtt{L}_{k,m,i}$ and $\mathtt{L}_{k,n,i}$, we denote by $\mathtt{L}_{k,m,i} \prec \mathtt{L}_{k,n,i}$ that the k-ancestor label path of node m is a* prefix *of the k-ancestor label path of node n.*

Example 6. In Example 5, we observed that $\mathtt{L}_{1,n_a,0} \prec \mathtt{L}_{1,n_b,0}$.

[10] Where $\uparrow^0 = \varepsilon$ and for $i > 0$, $\uparrow^i = \underbrace{\uparrow \diamond \cdots \diamond \uparrow}_{i \text{ times}}$.

Relationship to Partition Blocks To precisely single out blocks of a $P(k)$-partition, we introduce the following class of expressions derived from the $\mathtt{L}_{k,n,i}$ expressions above. The trick is to eliminate all spurious node pairs introduced from blocks with prefixing ancestor label expressions.

Definition 10. *Let $D = (V, Ed, r, \lambda)$ be a document and let $k \in \mathbb{N}$. Then the k-partition labeling expression for (n, m), with $(n, m) \in \mathsf{UpPaths}(D, k)$, is the $\mathscr{U}(k)$ expression $\mathtt{label}_{k,(n,m)} = \mathtt{L}_{k,n,l} - \bigcup_{n' \in V \ \&\ \mathtt{L}_{k,n,l} \prec \mathtt{L}_{k,n',l}} \mathtt{L}_{k,n',l}$, where $l = \mathsf{length}(n, m)$.*

Example 7. Since $\mathtt{L}_{1,n_a,0} \prec \mathtt{L}_{1,n_b,0}$, as we observed in Example 6, we have that $\mathtt{label}_{1,(n_a,n_a)} = \mathtt{L}_{1,n_a,0} - \mathtt{L}_{1,n_b,0} = \mathtt{A} - \mathtt{A}[\uparrow \diamond \mathtt{A}]$, and clearly this expression evaluated on document D of Figure 5 gives us precisely the $P(1)$-partition block for pair (n_a, n_a), namely $\mathtt{label}_{1,(n_a,n_a)}(D) = [(n_a, n_a)]$, as desired.

By construction of partition labeling expressions, it is easy to see that for a given block B of a $P(k)$-partition of document D, each $(n, m) \in B$ has the same label. Furthermore, by an examination of the definition of $\mathtt{label}_{k,(n,m)}$, it is straightforward to show that it is indeed the case that $(n, m) \in \mathtt{label}_{k,(n,m)}(D)$. In other words, we have that for each block B, an expression $\mathtt{label}_B \in \mathscr{U}(k)$ can be constructed such that $\mathtt{label}_B(D) = B$, completing the proof of Proposition 2. These are precisely the labeling expressions of Theorem 6.

5 Towards indexes: $A(k)$-based, or $P(k)$-based?

In Section 3, we argued that many XPath queries can be evaluated by (1) discovering appropriate blocks of $P(k)$-partitions and (2) assembling these blocks, typically through unions and joins, into the final answer. Step (1) was accomplished through decomposition and inversion techniques. Relative to a $P(k)$-partition, these techniques yield expressions in $\mathscr{D}(k)$ and $\mathscr{U}(k)$ without predicate operations. Through the Label-Union Theorem developed in Section 4, we know that these expressions can be associated with label expressions, which are syntactic objects that identify the relevant blocks. Thus, to develop an index structure to support these evaluations, we need a data structure that organizes these label expressions and their associated partition blocks in a way that allows fast look up. Given the simplicity of the labeling expressions, this is entirely feasible. In fact, we are currently implementing such a index structure, and plan to report on its performance. One of the potential drawbacks of such an index structure is that it can be large: for a given k, its size is $O(k|V|)$ where V is the set of nodes of the document. However, we believe that in practice, storing such indexes will only be necessary for small k values, and as such their size is nearly linear in the size of the document.

Of course, it is also possible to develop indexes that are based on the $A(k)$-partitions. In fact, the $A(k)$-index introduced by Kaushik et al. [10] is an example of this. This index has several very desirable properties: (1) its size is $O(|V|)$ and (2) for expressions in $\mathscr{U}(k)$ without predicates wherein exactly k "$\uparrow$" primitives occur, simple navigations through the index yield their results. However, it has

also some significant limitations. For example, consider an expression without predicates in $\mathscr{U}(j)$, $j > k$, that utilizes j "$\uparrow$" primitives. Such an expression can be written in the form $E_1 \diamond E_2$ where $E_1 \in \mathscr{U}(k)$ and $E_2 \in \mathscr{U}(j-k)$. Now the $A(k)$-index can determine the set of nodes that are the result of evaluating E_1 on the document. However now, starting from these nodes, E_2 is to be evaluated, and this can only be done by accessing and navigating the original document tree. (Notice that an index based on the $P(k)$-partitions does not suffer from this problem because it never requires extra navigation in the document.) A very similar problem occurs with expressions that have predicates. Consider an expression in $\mathscr{U}(j)$ of the form $E_1[E_2]$, where $E_1 \in \mathscr{U}(k)$ and $E_2 \in \mathscr{U}(j-k)$. Again, the $A(k)$-index can support E_1 well and retrieve the set of nodes that are the result of its evaluation. But again, to process the predicate $[E_2]$, it is necessary to navigate the original document. (Notice, again, that the $P(k)$-based indexes do not suffer from this problem.)

From this discussion, we conclude that for many reasons, $P(k)$-based indexes are to be preferred over $A(k)$-based indexes, especially when only small k's are sufficient.

6 Future Directions

In this paper, we take a fresh step towards establishing connections between the theoretical study of query languages and engineering research on the design and implementation of XML database systems. These connections hinge on a new methodology for coupling *index*-induced partitions and *language*-induced partitions of an XML document. To take full advantage of the $P(k)$-partitions introduced here and their block labeling expressions, we next need a data structure that is capable of locating all partition blocks based on label look-up, and in which the partition blocks that participate in the evaluation of a query are stored close to each other and can be located with a minimum number of label look-ups. Currently, we are focusing efforts towards the development of a data structure which satisfies these requirements.

References

[1] S. Al-Khalifa *et al.* Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] N. Bruno *et al.* Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[3] J. Clark and S. D. (eds.). XML path language (XPath) version 1.0. `http://www.w3.org/TR/XPATH`.

[4] D. Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[5] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[6] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[7] M. Gyssens *et al.* Structural Characterizations of the Semantics of XPath as Navigation Tool on a Document. In *ACM PODS*, pages 318–327, 2006.

[8] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *IEEE ICDE*, 2004.

[9] R. Kaushik *et al.* Covering indexes for branching path queries. In *SIGMOD*, 2002.
[10] R. Kaushik *et al.* Exploiting local similarity for efficient indexing of paths in graph structured data. In *IEEE ICDE*, 2002.
[11] R. Kaushik *et al.* On the integration of structure indexes and inverted lists. In *ACM SIGMOD*, 2004.
[12] C. Koch. Processing queries on tree-structured data efficiently. In *ACM PODS*, pages 213–224, 2006.
[13] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.
[14] M. M. Moro *et al.* Tree-pattern queries on a lightweight XML processor. In *VLDB*, 2005.
[15] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.
[16] P. Ramanan. Covering indexes for XML queries: Bisimulation - simulation = negation. In *VLDB*, 2003.
[17] K. Runapongsa, J. M. Patel, R. Bordawekar, and S. Padmanabhan. XIST: An XML index selection tool. In *XSym*, pages 219–234, 2004.
[18] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintenence of XML structural indexes. In *ACM SIGMOD*, pages 491–502, 2004.
[19] C. Zhang *et al.* On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.

Appendix

Structural Characterizations of $\mathscr{U}(k)$ Indistinguishability

We prove the following fact needed in establishing Theorem 4: on a fixed document, for each $k \geqslant 0$ it is the case that $P(k)$-equivalence (of node pairs) implies indistinguishability in the $\mathscr{U}(k)$ algebra (of node pairs).

Lemma A. *Let $D = (V, Ed, r, \lambda)$ be a document, $k \in \mathbb{N}$, and n_1, m_1, $n_2 \in V$ with m_1 is an ancestor of n_1 and $\mathsf{length}(n_1, m_1) \leq k$. If $n_1 \equiv_{A(k)} n_2$, then there exists $m_2 \in V$ such that m_2 is an ancestor of n_2 and $(n_1, m_1) \equiv_{P(\mathsf{length}(n_1,m_1))} (n_2, m_2)$.*[11] *Furthermore, $m_1 \equiv_{A(k-\mathsf{length}(n_1,m_1))} m_2$.*

Proof. By induction on k. For the base case, $k = 0$, clearly $m_1 = n_1$ and $\lambda(n_1) = \lambda(n_2)$. The statement holds for $m_2 = n_2$.

For $k \geq 1$, we can assume that the statement holds for $k-1$. If $n_1 \equiv_{A(k)} n_2$, then either (1) both n_1 and n_2 have no parents, or (2) they both have parents p_1 and p_2, respectively, such that $p_1 \equiv_{A(k-1)} p_2$ (by definition of $A(k)$ equivalence). In case (1), clearly $m_1 = n_1$ and the statement holds for $m_2 = n_2$. In case (2), $\mathsf{length}(p_1, m_1) \leq k-1$, and by the definition of $A(k)$ equivalence, $p_1 \equiv_{A(k-1)} p_2$. By the induction hypothesis, there exists an ancestor m_2 of p_2 such that $(p_1, m_1) \equiv_{P(\mathsf{length}(p_1,m_2))} (p_2, m_2)$ and $m_1 \equiv_{A(k-1-\mathsf{length}(p_1,m_1))} m_2$. It readily follows that $(n_1, m_1) \equiv_{P(\mathsf{length}(n_1,m_1))} (n_2, m_2)$ and
$m_1 \equiv_{A(k-\mathsf{length}(n_1,m_1))} m_2$.

[11] And even stronger, $(n_1, m_1) \equiv_{P(k)} (n_2, m_2)$.

Proposition A. *Let $D = (V, Ed, r, \lambda)$ be a document, $k \in \mathbb{N}$, $E \in \mathscr{U}(k)$, and $n_1, m_1,\ n_2, m_2 \in V$ such that m_1 is an ancestor of n_1 and m_2 is an ancestor of n_2, and $(n_1, m_1) \equiv_{P(k)} (n_2, m_2)$. If $(n_1, m_1) \in E(D)$, then $(n_2, m_2) \in E(D)$, and vice versa.*

Proof. First observe that it follows from $E \in \mathscr{U}(k)$ and $(n_1, m_1) \in E(D)$ that $\mathsf{length}(n_1, m_1) \leq k$, by Proposition 1.

The proof is by induction on k. The base case, $k = 0$, follows straightforwardly from the definition of $P(0)$-equivalence and a simple structural induction on expressions in $\mathscr{U}(0)$. Now assume that $k \geq 1$, and that the statement holds for $0, 1, 2, \ldots, k-1$. The proof goes by structural induction on expressions in $\mathscr{U}(k)$. Thus, let $E \in \mathscr{U}(k)$.

- $E \in \mathscr{U}(k-1)$. The statement holds by the induction hypothesis.
- $E = \uparrow$. If $(n_1, m_1) \in \uparrow (D)$, then m_1 is the parent of n_1. Since $(n_1, m_1) \equiv_{P(k)} (n_2, m_2)$, it follows in particular that m_2 is the parent of n_2. We conclude that $(n_2, m_2) \in \uparrow (D)$.
- $E = E_1 \cup E_2$, for E_1 and $E_2 \in \mathscr{U}(k)$. Suppose $(n_1, m_1) \in E(D)$. Then $(n_1, m_1) \in E_1(D)$ or $(n_1, m_1) \in E_2(D)$. Without loss of generality, assume $(n_1, m_1) \in E_1(D)$. Then by structural induction, $(n_2, m_2) \in E_1(D)$, and we conclude $(n_2, m_2) \in E(D)$.
- $E = E_1 \cap E_2$ or $E = E_1 - E_2$, for E_1 and $E_2 \in \mathscr{U}(k)$. Similar to the previous case.
- $E = E_1 \diamond E_2$, for $E_1 \in \mathscr{U}(k_1)$ and $E_2 \in \mathscr{U}(k_2)$, such that $k_1 + k_2 \leq k$. Suppose $(n_1, m_1) \in E(D)$. Then there is a node $w_1 \in V$ such that $(n_1, w_1) \in E_1(D)$ and $(w_1, m_1) \in E_2(D)$. By Lemma 1, $\mathsf{length}(n_1, w_1) \leq k_1$ and $\mathsf{length}(w_1, m_1) \leq k_2$. By Lemma A, there is a node $w_2 \in V$ such that $(n_1, w_1) \equiv_{P(\mathsf{length}(n_1, w_1))} (n_2, w_2)$, and $w_1 \equiv_{A(k - \mathsf{length}(n_1, w_1))} w_2$. Since, $k_2 \leq k - \mathsf{length}(n_1, w_1)$, by Lemma A, a node $m' \in V$ exists with $(w_1, m_1) \equiv_{P(\mathsf{length}(w_1, m_1))} (w_2, m')$.
  By $(n_1, w_1) \equiv_{P(\mathsf{length}(n_1, w_1))} (n_2, w_2)$, and $(w_1, m_1) \equiv_{P(\mathsf{length}(w_1, m_1))} (w_2, m')$, it is (definitions of $\equiv_{P(k_1)}$ and $\equiv_{P(k_2)}$) that $\mathsf{length}(n_2, w_2) = \mathsf{length}(n_1, w_1)$ and $\mathsf{length}(w_2, m') = \mathsf{length}(w_1, m_1)$.
  Consequently, $\mathsf{length}(n_2, m') = \mathsf{length}(n_1, m_1)$, and since m' is the unique ancestor at this length, we conclude that $m' = m_2$. Thus $(w_1, m_1) \equiv_{P(k_2)} (w_2, m_2)$. By the induction hypothesis, we can conclude that $(n_2, w_2) \in E_1(D)$ and $(w_2, m) \in E_2(D)$ and thus $(n_2, m_2) \in E(D)$.
- $E = E_1[E_2]$, for $E_1 \in \mathscr{U}(k_1)$ and $E_2 \in \mathscr{U}(k_2)$, such that $k_1 + k_2 \leq k$. Similar to the previous case.