# Covering Indexes for Branching Path Queries[*]

**Raghav Kaushik**[†]
University of Wisconsin
*raghav@cs.wisc.edu*

**Philip Bohannon**
Bell Laboratories
*bohannon@bell-labs.com*

**Jeffrey F Naughton**
University of Wisconsin
*naughton@cs.wisc.edu*

**Henry F Korth**
Bell Laboratories
*hfk@research.bell-labs.com*

## ABSTRACT

In this paper, we ask if the traditional relational query acceleration techniques of summary tables and covering indexes have analogs for branching path expression queries over tree- or graph-structured XML data. Our answer is yes — the forward-and-backward index already proposed in the literature can be viewed as a structure analogous to a summary table or covering index. We also show that it is the smallest such index that covers all branching path expression queries. While this index is very general, our experiments show that it can be so large in practice as to offer little performance improvement over evaluating queries directly on the data. Likening the forward-and-backward index to a covering index on all the attributes of several tables, we devise an index definition scheme to restrict the class of branching path expressions being indexed. The resulting index structures are dramatically smaller and perform better than the full forward-and-backward index for these classes of branching path expressions. This is roughly analogous to the situation in multidimensional or OLAP workloads, in which more highly aggregated summary tables can service a smaller subset of queries but can do so at increased performance. We evaluate the performance of our indexes on both relational decompositions of XML and a native storage technique. As expected, the performance benefit of an index is maximized when the query matches the index definition.

## 1. INTRODUCTION

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a labeled-tree or labeled-graph data model. A variety of languages have been proposed for this purpose all of which can be viewed as consisting of a *pattern language* and a *construction expression*. Fundamental to the pattern language is the *branching path expression*.

The idea behind evaluating branching path expressions is to find all ways of embedding the pattern in the data. An example of such a query is "find all parts consisting of both a nut and a bolt". The XPath W3C standard [3] makes posing branching path expressions very concise and natural. For example "//part[bolt and nut]" would express the above query in XPath (given suitable tag names). Because they lie at the core of most languages for processing XML data, efficient evaluation techniques for these languages will require efficient evaluation techniques for branching path expressions.

In this paper we ask if "covering indexes" can be used to accelerate the evaluation of such queries. Defining covering indexes to speed query performance is a well-known technique for SQL queries in relational database systems. Briefly, the idea is to define an index that "covers" all the attributes of a table that are referenced in a query. Then the query can be evaluated from the index alone, without consulting the table over which the index is defined. Since the index is expected to be much smaller than the table itself, this can provide impressive speedups.

Returning to our question, are there analogs to "covering indexes" for branching path expression queries over tree- or graph-structured XML data? The answer is certainly yes for simple path expressions — the strong DataGuide [6] and the 1-Index [11] can be viewed as covering indexes, since it is possible to answer queries over those indexes directly without consulting the base data. Unfortunately, these indexes can be large in practice. Our work pushes the frontier in two directions: we consider branching path expressions (rather than just simple path expressions), and we explore ways to reduce the size of covering indexes so that they become useful as query accelerators.

We show that the Forward and Backward-Index(F&B-Index), defined directly from ideas proposed in [2], can be viewed as a covering index for branching path expression queries. We also show that among a large and natural class of indexes, it is the smallest index that can cover *all* branching path expression queries. Unfortu-

nately, our experiments show that in practice, the size of the F&B-Index can approach the size of the base data itself. In this case, little performance gain is possible, since evaluating a query on this large index is just like evaluating the query on the base data.

This would seem to pose an unsolvable dilemma — the F&B-Index is the smallest covering index, yet it is too large to be useful. However, we can attack this problem in a way that is analogous to the approach used in relational systems. In a relational setting, indexes that cover all attributes of all tables are rarely built. Instead, indexes are built on specially chosen subsets of certain tables. Note that the result is an index that does not cover all queries, but that is expected to be very efficient for the queries it does cover. Inspired by this observation, we propose a scheme that allows us to explore and exploit a tradeoff between the size of a covering index for branching path expressions and the size of the class of queries that the index covers.

These indexes can range all the way from tiny indexes that are so focussed that they almost amount to cached answers to specific queries, to the full F&B-Index (which as we have said covers all path expressions), and many points in between. When viewed in this manner, these covering indexes also take on some of the flavor of summary tables in OLAP or multidimensional workloads — in these environments, the more highly aggregated the summary table, the smaller the class of queries it can service, but the higher the performance it can deliver.

Of course, in the context of branching path expressions over the labeled graph data model, the techniques used to specify and build these indexes are entirely different from the techniques used to define and build covering relational indexes or multidimensional summary tables. The main contribution of this paper is to propose these techniques and to evaluate them experimentally.

Our covering index technology applies to a number of different scenarios. Each of the scenarios is a different context in which branching path expressions are useful. The three scenarios we chose to explore are (1) data stored in a native XML format, with native XML query processing; (2) data stored in an RDBMS, with the understanding that the data originated as XML data but was "shredded" into a relational schema so that an RDBMS can be used as the query processor; (3) data stored in an RDBMS, with the understanding that the data originated in the RDBMS but an application is posing branching path expressions over an XML view defined on this relational data.

For each alternative, we compare the performance of branching path expression queries with and without our covering index. Depending on the specific query and index chosen, in all scenarios we obtain significant performance improvements, and observe that using the covering index can be an order of magnitude faster than using the base data.

The rest of the paper is organized as follows. Background material and an illustrative example are provided in Section 2. In Section 3, we review the notion of forward and backward index and show that it is a covering index for all branching path expression queries, and

prove that it is the smallest such index. In Section 4, we introduce our index definition scheme and provide an algorithm to construct an index according to a definition. Section 5 evaluates the performance of these covering indexes. We summarize related work in Section 6 and conclude in Section 7.

## 2. XML, BRANCHING PATH EXPRESSIONS, AND BISIMILARITY

In this section we review some concepts and definitions that will be useful throughout the paper.

### 2.1 The Labeled Graph Data Model

We model XML or other semi-structured data as a directed, node-labeled tree with an extra set of special edges called *idref* edges. More formally, consider a directed graph $G = (V_G, E_T, E_{Ref}, root, \Sigma_G, nodelabel, oid, value)$. $V_G$ is the node set. $E_T$ denotes the set of tree edges. The graph induced by $E_T$ on $V_G$ defines the underlying spanning tree. Each edge in $E_T$ indicates an object-subobject or object-value relationship. When we talk about parent-child and ancestor-descendant relationships, we refer to the tree edges. $E_{Ref}$ is the set of *idref* edges each of which indicates an *idref* relationship. "Simple" nodes in $V_G$ have no outgoing edges and are given a value via the *value* function. Each node in $V_G$ is labeled with a string-literal from $\Sigma_G$ via the *nodelabel* function and with a unique identifier via the *oid* function, with simple objects given the distinguished label, VALUE. There is a single *root* element with the distinguished label, ROOT.

Figure 1 shows a portion of a hypothetical "metro-guide", represented as a data graph. The solid edges represent the tree edges. The numeric identifiers in nodes represent *oid*'s. Non-tree edges (shown dashed) may be implemented with the ID/IDREF construct or XLink [5] syntax. The nodes labeled feature and star are attributes of their parent elements and indicate respectively whether a museum has a featured exhibit and whether a hotel is starred. This guide could be a large XML document, the output of publishing a relational database, or the result of decomposing an XML document into relations for the purpose of storage and querying. Attributes like "name", "address", etc are suppressed from the data graph.

### 2.2 Branching Path Expressions

A *label-path* is a sequence of labels $l_1 \ldots l_p$ ($p \geq 1$), separated by forward separators $/, //, \Rightarrow$ or by backward separators $\backslash, \backslash\backslash, \Leftarrow$. A *node-path* in $G$ is a sequence of nodes, $n_1 \ldots n_p$, again separated by $/, //, \Rightarrow$ or $\backslash, \backslash\backslash, \Leftarrow$ such that, for $1 \leq i \leq p - 1$, if $n_i$ and $n_{i+1}$ are separated by a

1. $/$, then $n_i$ is the parent of $n_{i+1}$
2. $//$, then $n_i$ is an ancestor of $n_{i+1}$
3. $\Rightarrow$, then $n_i$ points to $n_{i+1}$ through an *idref* edge
4. $\backslash$, then $n_i$ is a child of $n_{i+1}$
5. $\backslash\backslash$, then $n_i$ is a descendant of $n_{i+1}$
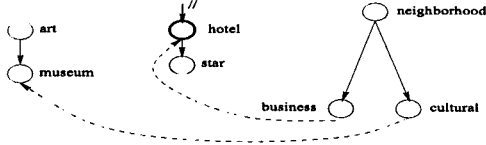6. $\Leftarrow$, then $n_i$ is pointed to by $n_{i+1}$ through an *idref* edge

134

**Figure 1: An example graph-structured database**

A node-path $n_1 \ldots n_p$ *matches* a label path $l_1 \ldots l_p$ if the corresponding separators are the same and $label(n_i) = l_i$, for $1 \leq i \leq p$. Label paths and node paths where the separators are restricted to be the forward separators are called *forward label paths* and *forward node paths* respectively. We can similarly define *backward label paths* and *backward node paths*. Label paths that involve both forward and backward separators are called *mixed paths*. For example, in Figure 1, the path ROOT/metro/neighborhoods/neighborhood/business⇒hotel is a forward label path, and the node path 1/2/5/9/24 ⇒ 23 matches it. The label path ROOT/metro/business/hotels /hotel⇐business\neighborhood is a mixed label path and the node path 1/2/4/7/23 ⇐ 24\9 matches it.

We now provide examples of branching path expressions, followed by a definition. The path expression ROOT/metro/neighborhoods/neighborhood[/business⇒hotel]/cultural⇒museum finds all museums that have a hotel in the same neighborhood. This is done by specifying a *primary path* ROOT/metro/neighborhoods/neighborhood/cultural⇒museum to museum nodes and applying a condition on (intermediate) nodes labeled neighborhood asserting that they have the path /business⇒hotel coming out of them. More formally, we define branching path expressions by the following grammar sketch.

| | | |
|---|---|---|
| bpathexpr | → | fwdlabelpath [orexpr] fwdsep bpathexpr |
| | | \| fwdlabelpath |
| orexpr | → | andexpr 'or' orexpr |
| | | \| andexpr |
| andexpr | → | bpathexpr2 'and' andexpr |
| | | \| notbpathexpr2 'and' andexpr |
| | | \| bpathexpr2 |
| | | \| notbpathexpr2 |
| notbpathexpr2 | → | 'not' bpathexpr2 |
| bpathexpr2 | → | labelpath2 [orexpr] bpathexpr2 |
| | | \| labelpath2 |
| labelpath2 | → | fwdsep labelpath |
| | | \| backsep labelpath |
| fwdlabelpath | → | forward label paths |
| labelpath | → | label paths |
| fwdsep | → | / \| // \| ⇒ |
| backsep | → | \ \| \\ \| ⇐ |

In an XML context, the above definition of branching path expressions forms a subset of the XPath [3] stan-

dard, except for one additional feature, which is that we allow backward traversal of *idref* edges. Let us note here that we ignore order and value based selections, which will be addressed by future work.

As mentioned above, we can think of these branching path expressions as a basic forward label path (call it the *primary path*) with boolean path conditions on intermediate labels. The *primary path* corresponding to a branching path expression is the path that remains when all parts between brackets '[' and ']' are removed (including the brackets themselves). The other parts of the branching path expression act as path constraints on the primary path.

Evaluating a branching path expression on a graph as defined above amounts to matching the primary path and satisfying the intermediate boolean path conditions. A path condition is evaluated on a data node and takes the form of another branching path expression which is evaluated recursively. These constraints can be connected by the usual logical operators 'and', 'or' and 'not'. The return set corresponds to all nodes that match the last tag in the primary path. For example, in the above query that finds all museums having a hotel in the same neighborhood, the node with *oid* 16 is returned as part of the result, since (a) the node path 1/2/5/9/25 ⇒ 16 matches the primary path, (b) 16 is the *oid* of the node that matches the last tag, museum, in the primary path and (c) the node with *oid* 9, which matches the tag neighborhood in the primary path, satisfies the path condition imposed on it. The full answer set for this query is 15, 16, 17, 18, 19.

Some other examples of branching path expression queries are:

1. ROOT/metro/neighborhoods/neighborhood[/business⇒hotel and not /business⇒cinema-hall]/cultural ⇒museum asks for all museums that have a hotel in the same neighborhood, but no cinema-hall.

2. //hotel[star][⇐ business\neighborhood[/cultural⇒ museum[\art]]] finds all star hotels that have an art museum in the same neighborhood. Note that this query can written in a different way, similar to the query above.

These branching path expression queries can be thought

135

**Figure 2: Graph view of hotel query**

of in a graph format. For example, the graph in Fig 2 represents the "hotel" query above. The dark hotel node indicates that it is the return node. The // label on the edge coming in to the hotel node indicates the // separator in the query. Again, solid edges represent tree edges and dashed edges represent *idref* edges. It is straightforward to similarly visually represent boolean connectives. We note that the underlying undirected graph corresponding to a query graph is always a tree.

## 2.3 Index Graphs

In this paper we are concerned with index graphs. An *index graph* for data graph $G$ is a graph $I(G)$ where we associate an *extent* with each node in $I$. If $A$ is a node in the index graph $I(G)$, then $ext^I(A)$, the extent of $A$, is a subset of $V_G$. We add the constraint that the extents of two index nodes should never overlap. The *index graph result* of executing a branching path expression $P$ on $I(G)$ is the union of the extents of the index nodes that result from evaluating $P$ on $I(G)$. An index graph $I(G)$ *covers* a branching path expression query $P$ if the index result of $P$ is accurate, i.e., it is the same as the result on $G$.

The 1-Index [11] is an example of an index graph and covers in-coming path queries (i.e. forward label path queries). In fact, any partition of the data nodes defines an index graph where (1) we associate an index node with every equivalence class, (2) define the index node's extent to be the equivalence class that formed it and (3) add an edge from index node $A$ to index node $B$ if there is an edge from some data node in $ext(A)$ to some data node in $ext(B)$. Henceforth, whenever we refer to an index graph obtained from a partition of the data nodes, we mean the above construction. Thus, even a simple grouping of the data nodes by label defines an index graph.

We now introduce terminology about partitions of data nodes. A partition $P_1$ of the data nodes is a *refinement* of another partition $P_2$ if the following condition holds: whenever two nodes are in the same equivalence class in $P_1$, they are in the same equivalence class in $P_2$ as well. If $P_1$ is a refinement of $P_2$, then $P_2$ is *coarser* than $P_1$. We also talk about one index graph being a *refinement* of another — this refers to the corresponding partitions (and makes sense only if the set of data nodes indexed in both is the same).

## 2.4 Bisimilarity

We briefly introduce the notion of bisimilarity [14] since it is central to the rest of the paper. This notion was first used in the context of semi-structured data while introducing the 1-Index [11]. The intuition behind the 1-Index is to try and group together nodes if they have the same set of incoming paths, but achieving exactly this ideal grouping is PSPACE-complete [20]. The

solution of [11] is to use instead a grouping like bisimilarity, which *refines* the ideal grouping; that is, it splits some of the groups in the ideal grouping. We modify the definition slightly to distinguish between tree and *idref* edges.

A symmetric, binary relation $\approx$ on $V_G$ is called a *bisimulation* if, for any two data nodes $u$ and $v$ with $u \approx v$, we have that (a) $u$ and $v$ have the same label, (b) if $par_u$ is the parent of $u$ and $par_v$ is the parent of $v$, then $par_u \approx par_v$ and (c) if $u'$ points to $u$ through an *idref* edge, then there is a $v'$ that points to $v$ through an *idref* such that $u' \approx v'$, and vice-versa. Two nodes $u$ and $v$ in $G$ are said to be *bisimilar*, denoted by $u \approx^b v$, if there is some bisimulation $\approx$ such that $u \approx v$.

The partition of $V_G$ induced by $\approx^b$ can be used to obtain an index graph. This index graph is referred to as $Bisim(G)$ or simply "the 1-Index" in this paper[1]. Thus, there is a worst case guarantee on the index size, since the 1-Index can never be bigger than the data graph. Further, it can be computed in time $O(m \lg n)$ where $n$ is the number of nodes and $m$ is the number of edges in the data graph, using an algorithm proposed by Paige and Tarjan [13].

## 3. PROPERTIES OF THE FORWARD AND BACKWARD INDEX

In [2], the authors note that by using the notion of *inverse edges*, we get structural summaries (of schema-less data) that capture information about both in-coming and out-going paths. More precisely, let us consider the following process for an edge-labeled data graph (a similar process can be applied to node-labeled graphs too). While the discussion below does not distinguish between tree and *idref* edges, the definitions and properties we talk about can be easily tweaked to accommodate the same. We omit these details for lack of space.

1. For every (edge) label $l$, add a new label $l^{-1}$.

2. For every edge $e$ labelled $l$ from node $u$ to node $v$, add an (inverse) edge $e^{-1}$ with label $l^{-1}$ from $v$ to $u$.

3. Compute the 1-Index (or DataGuide) on this modified graph.

The above is a structural summary that captures information about paths both entering and leaving nodes in the data graph. With the 1-Index used in step 3 above, we obtain a partition of the data nodes which can be used to define an index graph. Let us call this the Forward and Backward-Index(F&B-Index).

In this section, we prove some new theorems about the F&B-Index that are important in the context of families of covering indexes. To do so, we first give an alternative definition of the F&B-Index based on the notion of the *stability* of one set of graph nodes with respect to another. For a set of nodes, $A$, let $Succ(A)$ denote the set of successors of the nodes in $A$, i.e., the set $\{v|$ there

---

[1]The authors of [11] also consider the use of the *similarity* relationship [10] for the 1-Index. We do not consider this alternative due to inefficient construction algorithms for the similarity relation – see [11] for details.

is a node $u \in A$ with an edge in $G$ from $u$ to $v$}. We can define the predecessor set of $A$, $Pred(A)$ analogously.

DEFINITION 1. *Given two sets of data graph nodes $A$ and $B$, $A$ is said to be succ-stable with respect to $B$ if either $A$ is a subset of $Succ(B)$ or $A$ and $Succ(B)$ are disjoint.*

It is possible to similarly define the notion of *pred-stability*. We call a partition of the nodes *succ-stable* if, for every pair of individual partitions $p_1$ and $p_2$, $p_1$ is succ-stable with respect to $p_2$ (when talking about the stability of partitions, we actually mean stability of the corresponding sets of nodes).

A succ-stable partition has the property that if we build an index graph from it, then whenever there is an edge from index node $A$ to index node $B$, there is an edge from every data node in $ext(A)$ to some node in $ext(B)$. Let us call a *label grouping* a partition of the data nodes that corresponds to their node-labels (i.e. two nodes are in the same equivalence class if they have the same label). The 1-Index is then the coarsest partition of the data nodes that is (1) a refinement of the label grouping and (2) is succ-stable. We can think of the 1-Index computation as consisting of two parts: (1) initialization by label grouping and (2) splitting the label grouping till we obtain a succ-stable refinement.

Now consider the following procedure over data graph $G$. We work with a current partition of the data nodes which is initialized to the label grouping.

1. Reverse all edges in $G$.
2. Compute the bisimilarity partition (with the current partition as the initialization).
3. Set the current partition to what is output by the previous step.
4. Reverse edges in $G$ again, obtaining the original $G$.
5. Compute the bisimilarity partition (again initializing the computation with the current partition).
6. Set the current partition to what is output by the previous step.
7. Repeat the above steps till the current partition does not change.

It is not hard to see that the F&B-Index is the index graph obtained by using this final partition. From this point of view, the idea behind the F&B-Index is to obtain a partition of the data nodes that is both succ-stable and pred-stable. One way to do this is by first ensuring pred-stability (by reversing $G$'s edges and computing the bisimilarity partition), then computing a succ-stable refinement of this pred-stable partition, and continuing thus till the current partition does not change.

The F&B-Index for the data graph shown in Figure 1 merges the two hotel nodes with *oids* 22 and 23. Other than that, in this case, the F&B-Index is the same as the data graph.

We have the following theorem that shows the importance of this index. The proof follows from the fact that the F&B-Index is both a pred-stable and succ-stable partition and is omitted for lack of space.

THEOREM 1. *The F&B-Index over a data graph $G$ covers all branching path expressions over $G$.*

Thus, for the data in Figure 1, the fact that two of the hotel nodes are merged together means that no branching path expression query can distinguish between the two. Conversely, using a simple diagonalization argument, we have the following theorem. The proof is again omitted for lack of space.

THEOREM 2. *For data graph $G$, any index graph that covers all branching path expressions over $G$ must be a refinement of the F&B-Index.*

COROLLARY 3.: *For data graph $G$, the F&B-Index is the smallest index graph that covers all branching path expressions over $G$.*

As a result, whenever two nodes are not in the same extent of the F&B-Index, there is *some* branching path query that distinguishes between the two.

## 3.1 Size of the F&B-Index

Unfortunately, the F&B-Index is often big. For example, on the XMark [1] XML benchmark document of size 10MB, the data has about 181000 nodes, while the F&B-Index has about 164000 nodes. Similarly, for a subset of the Open Directory Project [15] data where the data has about 143000 nodes, the F&B-Index has about 93000 nodes. This size problem with the F&B-Index leads us to look for a way to cut down the size of this index. Since it is the smallest index that handles all branching path expressions, the only way out is to compromise on the class of queries to be covered. This motivates the index definition scheme we propose in the next section.

## 4. COVERING INDEX DEFINITION SCHEME

As we saw above, the F&B-Index can be big and it is important to be able to cut down the index size. Since branching path expressions can be numerous and complicated, our index definition scheme is designed towards eliminating branching path expressions that are deemed less important, so that we arrive at an index that is much smaller and can handle the remaining branching path expressions more efficiently. We use four different approaches toward this goal based on the following intuitions.

1. There are many tags in data that are of lesser interest and so need not be indexed.
2. Branching path expressions (in the manner defined by us) give more importance to tree edges over *idref* edges. In particular, // matches only tree edges and there is no equivalent for *idref* edges. It may be desirable to reflect this in the index.
3. Not all structure is interesting. Our intuition is that queries on long paths are rare. Instead, short paths are more common. Thus, it may be useful to exploit local similarity to cut down the index size.
4. Restricting the "tree-depth" of the branching path expressions for which the index is accurate might help. We will explain this in more detail later.
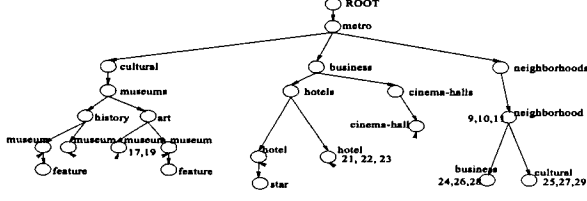
**Figure 3: F&B-Index on tree edges**

## 4.1 Tags to be indexed

It is often the case that there are tags in the data that are never queried using branching path expressions and thus need not be indexed. We do so by altering the data graph so that all nodes that have tags that are not to be indexed are labeled with a unique label, other. In addition, if a node labeled other does not appear in the tree path to any node that is being indexed, it can be assumed to be absent from the data graph for purposes of indexing. This simple technique can have a lot of effect in practice. For example, in the XMark data, there are text tags such as bold and emph that appear in the description of categories and items that are being auctioned. It may be worthwhile to build an index that ignores these tags. The F&B-Index on the tree edges of the XMark file of size 100MB (which has about 1.43 million nodes) has about 436000 nodes. But on ignoring the text nodes, the number of nodes in the index drops to about 18000.

## 4.2 Tree Edges Vs Idref Edges

The XPath [3] standard for path expressions gives a higher priority to tree edges. In particular, // matches only tree edges and there is no equivalent ancestor operation for *idref* edges. On the other hand, all occurrences of *idref*s in path expressions have to be explicit. This is also reflected in the way we have defined our branching path expressions. The effect on the index size of *idref*s can be tremendous. For example, on the XMark data of size 100MB, the F&B-Index on just the tree edges has 18000 nodes (ignoring text nodes) while the size is about 1.35 million when all *idref*s are incorporated (again ignoring text nodes). Thus, it is desirable to have some way of giving priority to tree edges. We specify the set of *idref* edges to be indexed as part of the index definition. We do so by specifying the source and target labels of the *idref* edges we wish to retain.

Figure 3 shows the F&B-Index constructed on only the tree edges of the data graph in Figure 1 (again we only show extents that have more than one data node). As we can see, all three nodes labeled neighborhood are merged together in this index, showing that they cannot be distinguished by any branching path expression query on the tree edges alone. However, this index cannot necessarily be used to cover a branching path expression query that refers to an *idref* edge.

## 4.3 Exploiting Local Similarity

As pointed out in [17], an important approach in con-

trolling index sizes lies in exploiting local similarity. Our intuition is that most queries refer to short paths and seldom ask for long paths. As a result, it may not be desirable to split the index partition along long paths. For example, in the data graph in Figure 1, it may not be desirable to split nodes labeled neighborhood based on whether they contain a museum that has a featured exhibit. This can be achieved by looking at paths of length up to 2 (here, length refers to the number of edges). This process is reminiscent of summary tables for OLAP workloads, where picking a larger number of attributes to aggregate yields a smaller summary that is accurate for queries covered by it while picking a smaller number of attributes to aggregate makes the summary more generic.

We have the notion of $k$-bisimilarity (defined in [10]) which groups nodes based on paths of length up to $k$. We reproduce it below (again modified slightly to distinguish between tree and *idref* edges).

DEFINITION 4.: $\approx^k$ *(k-bisimilarity): This is defined inductively.*

1. *For any two nodes, $u$ and $v$, $u \approx^0 v$ iff $u$ and $v$ have the same label.*
2. *Node $u \approx^k v$ iff $u \approx^{k-1} v$, $par_u \approx^{k-1} par_v$ where $par_u$ and $par_v$ are respectively the parents of $u$ and $v$, and for every $u'$ that points to $u$ through an idref edge, there is a $v'$ that points to $v$ through an idref edge such that $u' \approx^{k-1} v'$, and vice versa.*

By a simple induction, we can see that this definition ensures the weaker condition it sets out to achieve. Note that $k$-bisimilarity defines an equivalence relation on the nodes of a graph. We call this the *k-bisimulation*. This partition is used in [17] for in-coming path queries, to define an index graph, the $A(k)$-index, where $k$ is a parameter. Thus, an $A(2)$-index is accurate for the query //neighborhood/cultural⇒museum, but not necessarily for the query //neighborhood/cultural⇒museum/featured.

In our context, it should be possible to specify, say, that in the forward direction, we only want local similarity for paths of length at most 1 and in the backward direction, we want "global" similarity. This way, we get a covering index where, for example, the condition "museum with a featured exhibit", or the condition "hotel that is a star hotel" can be imposed since these only involve paths of length 1, whereas the condition "neighborhood with a museum that has a featured exhibit" cannot be imposed since it involves a longer path. We argue that by thus restricting the query-able path length, we can still index a large and interesting subset of branching path expression queries.

## 4.4 Restricting Tree Depth

In Section 3, we defined the F&B-Index as a transitive closure of the following sequence of computations:

1. Reverse all edges in $G$.
2. Compute the bisimilarity partition (with the current partition as the initialization).
3. Reverse edges in $G$ again, obtaining the original $G$.
4. Compute the bisimilarity partition (again initializing the computation with the current partition).
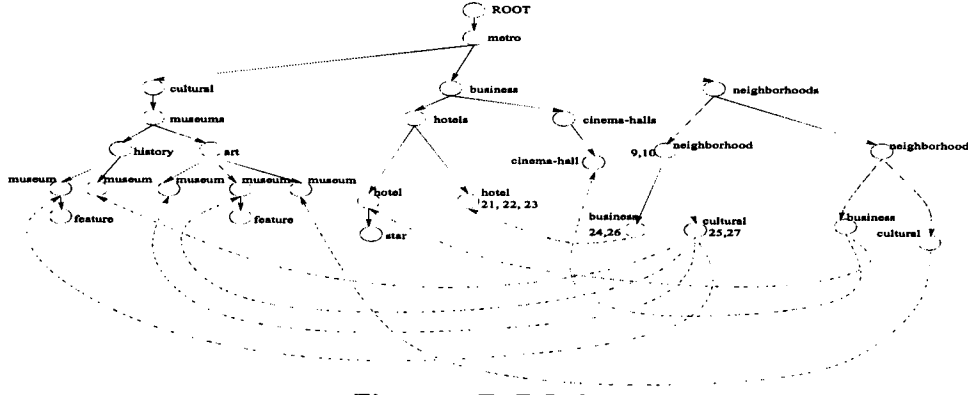
**Figure 4: F+B-Index**

Consider the partition (and hence the index graph) defined by one iteration of these operations. Let us call this index graph the *F+B-Index*. The index graph obtained after two iterations above would be called *F+B+F+B-Index* and so on.

We now define the notion of tree-depth of a node in a query. Consider the query //museums/history/museum[/featured and ⇐ cultural\neighborhood [/cultural⇒ museum[\ art]]] that asks for history museums that have a featured exhibit and also have an art museum in the same neighborhood. The query is shown in Figure 5 in graph format. The numbers to the side indicate the tree-depths of the nodes. The idea is that all nodes on the primary path and having a path to some node in the primary path have tree-depth 0. All nodes that do not have tree-depth 0 and have a path from some node in the primary path have tree-depth 1, nodes that do not have tree-depth 1 and have a path to some node of tree-depth 1 have tree-depth 2, nodes that do not have tree-depth 2 and have a path from some node of tree-depth 2 have tree-depth 3, and so on. Intuitively, odd tree-depths correspond to out-going path conditions, while even tree-depths correspond to in-coming path conditions. Nodes that have an edge to or from a node of higher tree-depth are called *branching points*. The nodes in the example query graph that are branching points are indicated in bold. The return node is shaded, and in this case, also happens to be a branching point. Note that tree-depth is different from the nesting level of a node in the query text. In particular, the neighborhood node has tree-depth 0 although it is nested in the query text. In general, the same query can be written in more than one way. Tree-depths of nodes do not depend on how the query is written (whereas nesting levels do). The tree-depth of a query is the maximum tree-depth of its nodes.

The F+B-Index is accurate for the subset of branching path expressions over $G$ that have tree-depth at most 1.
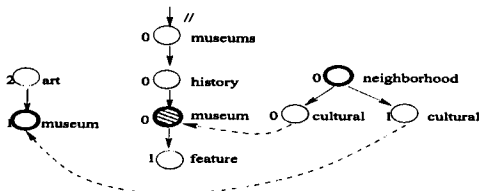
The F+B+F+B-Index is accurate for all branching path expressions that have tree-depth at most 3 and so on (to restrict to tree-depth 2, we must use a "B+F+B-Index"). It is our intuition that it is rare to get meaningful branching path queries with large tree-depths. Hence, we wish to be able to restrict the tree-depth of queries being indexed by specifying the maximum tree-depth we want to index. So, we make this part of the index definition.

Figure 4 shows the F+B index for the data in Figure 1. Note that nodes 9 and 10, labeled neighborhood are together in this index, unlike in the F&B-Index. However, the other node labeled neighborhood, 11, is separate, unlike the index on the tree edges alone. Since this index is smaller than the F&B-Index, it cannot handle all branching path expression queries. In particular, a query that can be answered by the F&B-Index but not by the F+B-index is //neighborhood[/cultural⇒museum[/featured][\history]], which asks for all neighborhood nodes that have a history museum that has a featured exhibit. This cannot be answered by the F+B-index since it cannot distinguish between the two neighborhood nodes 9 and 10, whereas node 10 is in the answer set and node 9 is not. This inaccuracy arises because the term history is at a tree depth of 2, whereas the F+B-index is only (necessarily) accurate for queries with tree-depth at most 1.

## 4.5 Putting it together

An index definition consists of the following parts:

1. A set of tags to be indexed. Call this set $T$.
2. For each of the forward and backward directions:

   (a) Set of *idref* edges to be indexed (call them $ref_{fwd}$ and $ref_{back}$)
   (b) A parameter $k$ indicating the extent of local similarity desired (call them $k_{fwd}$ and $k_{back}$)

3. The number of iterations in the F&B-Index computation to be performed. Call this $td$, the tree depth.

The parameters $k_{fwd}, k_{back}$ and $td$ can be set to be $\infty$, referring to a transitive closure computation. The index obtained for a given index definition $S$ is called the $BPCI(S)$ (for branching path expression covering index). The algorithm for computing the $BPCI(S)$ is shown below in Figure 6. The $k$-bisimulation partition



**Figure 5: Example for tree depth**

can be computed by the algorithm given in [17], and it can be trivially extended to handle the $k = \infty$ case.

**procedure** compute_partition($G$,$S$)
$G \rightarrow$ data graph, $S \rightarrow$ index definition
**begin**
1.  Convert all tags in $G$ not in $T$ into special tag other
2.  Remove any occurrence of a node labeled other if it is not on some tree path from the root to any node with a label that is to be indexed
3.  Let $\mathcal{P}$ be a list of sets of nodes
    //representing a partition of the nodes of $G$
4.  $\mathcal{P} \leftarrow$ label-grouping of $G$
5.  **for** $i = 1$ to $td$ **do**
    //forward direction
6.  Retain $idref$ edges in $ref_{fwd}$
7.  Reverse all edges in $G$
8.  Compute the $k_{fwd}$-bisimulation on $G$ initializing the computation with $\mathcal{P}$
9.  $\mathcal{P} \leftarrow$ partition of nodes of $G$ corresponding to the above $k_{fwd}$-bisimulation
    //backward direction
10. Restore $G$
11. Retain $idref$ edges in $ref_{back}$
12. Compute the $k_{back}$-bisimulation on $G$ initializing the computation with $\mathcal{P}$
13. $\mathcal{P} \leftarrow$ partition of nodes of $G$ corresponding to the above $k_{back}$-bisimulation
**end**

**procedure** compute_index($G$, $S$)
**begin**
1.  compute_partition($G$,$S$)
2.  **foreach** equiv. class in $\mathcal{P}$ **do**
3.    create an index node $I$
4.    $ext[I] =$ data nodes in the equiv. class
5.  **foreach** edge from $u$ to $v$ in $G$ **do**
6.    $I[u] =$ index node containing $u$
7.    $I[v] =$ index node containing $v$
8.    **if** there is no edge from $I[u]$ to $I[v]$ **then**
9.      add an edge from $I[u]$ to $I[v]$
**end**

**Figure 6:** *BPCI(S)* **computation**

The subset of branching path expressions for which an arbitrary covering index is accurate depends, of course, on the index definition. The following are some example index definitions and the indexes they generate.

1.  The F&B-Index can be obtained by indexing all tags and all *idref* edges, with $k_{fwd} = k_{back} = td = \infty$.
2.  The F+B-index can be generated by indexing all tags and all *idref* edges, with $k_{fwd} = k_{back} = \infty$ and $td = 1$.
3.  The 1-Index can be generated by indexing all tags and all *idref* edges, with $k_{fwd} = 0, k_{back} = \infty$ and $td = 0$.
4.  The A($k$)-index can be generated by indexing all tags and all *idref* edges, with $k_{fwd} = 0, k_{back} = k$ and $td = 0$.

Consider the following definition on the data in Figure 1.

1.  index tags ROOT, metro, cinema-hall, neighborhoods, neighborhood, business
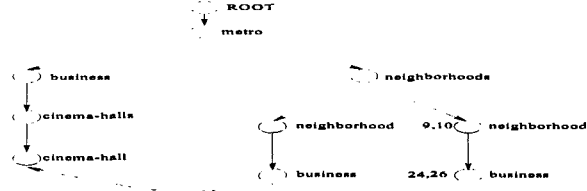


**Figure 7: Index for given definition**

2.  pick $k_{fwd} = k_{back} = \infty$
3.  tree depth $= \infty$.

The index generated is shown in Figure 7. This is almost a precomputation of the query
    //neighborhood[/business⇒cinema-hall],
which asks for all neighborhoods that have a cinema hall. As seen above, we can also generate the full F&B-Index by this workload scheme. Thus, a wide variety of indexes can be defined by our scheme.

## 4.6 Index Selection

We now discuss the issue of how to arrive at a reasonable index definition that covers a set of queries. Note that this is analogous to the problem of choosing an appropriate covering index for a given query workload. Consider the following queries on the data in Figure 1.

1.  Hotels that have a museum in the same neighborhood. This could be written as a branching path expression query in the following way: //neighborhood[/cultural⇒museum]/business⇒hotel Note that the primay path length is 3 and the tree-depth is 1.
2.  Starred hotels: //hotel[/star]. The primary path length is 1, and the tree-depth is 1.
3.  Neighborhoods with an art museum: //neighborhood[/cultural⇒museum[\art]] Here, the primary path length is 1 and the tree-depth is 2.

The following constraints hold for any index that covers the above queries. First of all, the tags involved in this query must be indexed. Since the maximum tree-depth is 2, the index tree-depth must be atleast 2. Since the maximum path length for path conditions is 2 (//neighborhood[/cultural⇒museum]), $k_{fwd} \geq 2$. Similarly, the constraint on $k_{back}$ is that $k_{back} \geq 2$. Given these constraints, one straight-forward thing to do is to pick the minimum values needed to cover the queries. Call this index $I_{min}$. However, there is a tradeoff — by picking larger values for the parameters, we get a more generic index that can potentially cover more queries, although the performance of this index for these queries may be worse than that of $I_{min}$. This example serves to illustrate the issues involved in choosing an index definition to cover a set of branching path expression queries. It is possible to give simple heuristics (for example, output $I_{min}$) to create a covering index. However, a good choice of index definition depends heavily on the data and the queries, and will be addressed in future work.

## 4.7 Testing if an Index covers a Query

We now discuss how to test whether a given index covers a branching path expression query. Intuitively, the conditions that must be satisfied are:

140

**procedure** cover($I$, $Q$)
$I$ ← index
$Q$ ← branching path expression
**begin**
1. convert $Q$ into a query graph $Q_G$
2. check if all tags in $Q$ are indexed in $I$
3. check if the tree-depth of $Q \leq td$, the tree-depth of $I$
4. check if all paths in $Q_G$ with even tree-depth have length $\leq k_{back}$; if $k_{back} < \infty$, no separator in these paths should be //
5. check if all paths in $Q_G$ with odd tree-depth have length $\leq k_{fwd}$; if $k_{fwd} < \infty$, no separator in these paths should be //
**end**

**Figure 8: Checking if $BPCI(S)$ covers a query**
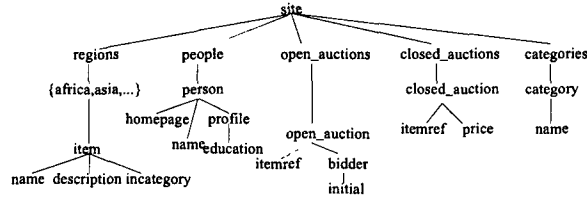


**Figure 9: XMark schema**

1. The tags and *idref*s should match — all tags and *idref*s referenced in the query must be indexed.
2. The tree depth should match — the tree-depth of the index should be at least as large as the tree-depth of the query.
3. If the index uses local similarity, then all relevant path lengths must be bounded by the extent of local similarity captured in the index.

In more detail, let $Q_G$ be the query graph. A path in $Q_G$ has tree-depth $i$ if all nodes in the path, with the possible exception of either end, have tree-depth $i$ (not all paths necesarily have a tree-depth). The procedure shown in Figure 8 tests whether a given index covers a given branching path expression query. This procedure can be implemented using a depth first search of the query graph and takes time linear in the query size.

## 5. PERFORMANCE

In this section we explore the performance of the covering indexes introduced in the previous section. We first investigate the efficacy of the index definition scheme in controlling the size of the covering index, and then go on to a performance study over queries that are covered by the index. All our experiments are over the 100MB XMark XML benchmark [1] data set. The XMark data models an auction site. The element relationships relevant to us are reproduced for convenience in Figure 9. The tag names are largely self-explanatory. The tag itemref is an *idref* value pointing to item nodes. Similarly, the tag incategory is an *idref* pointing to category nodes.

## 5.1 Range of Indexes

The goal of this subsection is (a) to establish that there is a wide variety of covering indexes spanning a whole range of sizes, and (b) to explore the effectiveness of each index definition parameter we have specified in controlling the index size.

| Defn No. | Index | No. of Nodes |
|---|---|---|
| Defn 0 | F&B-Index | 1.35 million |
| Defn 1 | F&B-Index on tree edges | 436602 |
| Defn 2 | F&B-Index on tree edges w/o text nodes | 18336 |
| Defn 3 | F+B-index w/o text nodes | 1.29 million |
| Defn 4 | F+B-index where we (a) ignore text tags (b) ignore *idref* edges in the fwd direction (b) ignore *idref*s pointing to person tags in the back direction | 467070 |
| Defn 5 | Defn 4 + ignore *idref*s pointing to open_auction tags in the back direction | 40938 |
| Defn 6 | Defn 5 + $k_{fwd} = 1$ | 10705 |
| Defn 7 | Defn 6, except that tree-depth = 3 | 32716 |
| Defn 8 | Index only tags and *idref* edges present in query Q | 17 |

**Table 1: Index Definitions and Sizes**

Since the number of possible index definitions is high, we restrict ourselves to a few representative examples. The definitions are shown in Table 1 (query Q will be introduced later) along with the number of nodes. The data has about 1.43 million nodes. The full F&B-Index on the whole data graph has about 1.35 million nodes, which indicates (by minimality of the F&B-Index as per Theorem 2) that almost all nodes in the data can be distinguished from one another by some branching path expression query. The F&B-Index on just the tree edges is itself quite large with approximately 430,000 nodes. The numbers in Table 1 indicate the wide range of sizes achievable by our covering indexes on this data set.

We now examine the importance of each of the parameters in an index definition in isolation.

1. *Ignoring Tags*: As explained in Section 4.1, a lot of the "splits" in the F&B-Index are caused by the text markup elements bold, emph, and so forth. The size of the F&B-Index on the tree edges ignoring these tags (Defn 2) is about 18000 nodes, as shown in Table 1, which is about 4% of the size of the F&B-Index on all tree edges (Defn 1).
2. *Picking idref edges to index*: The F+B-index on ignoring the text tags (Defn 3) has about 1.29 million nodes. We found that among all the *idref* edges, the ones pointing to person elements (call them *personref*s) caused the largest number of splits. Keeping everything else the same, if we ignore all *idref* edges in the forward direction and index all *idref* edges except the *personref*s in the backward direction (Defn 4), the covering index we obtain has about 470000 nodes, which is about 36% the size of the F+B-index (Defn 3). If we also ignore the *idref* edges pointing to open_auction elements in the backward direction (Defn 5), we obtain an index with 40000 nodes.
3. *Local Similarity*: In the above index (with 40000 nodes), if we set $k_{fwd} = 1$ keeping everything else the same (Defn 6), we get an index with only about 11000 nodes. We note that even this index, small

| Query name | Query |
|------------|-------|
| Q1 | find the number of persons whose education information is known |
| Q2 | find the number of persons with a homepage |
| Q3 | find "hot" items, i.e. items that have a bidder(same as Q above) |
| Q4 | for all items being auctioned, but w/o a bidder, find their category |
| Q5 | find all categories where items have been sold only from North America |

**Table 2: Test Queries**

as it is, can index a large number of branching path expressions. In particular, it allows conditions like item[/featured], person[/homepage] and so on to be imposed and allows primary paths that can be arbitrarily long and pass through any of the tree edges and any *idref* edge pointing to an item or category node.

4. *Tree Depth*: While the above index has tree depth of 1, the same index where the tree depth is set to 3 (Defn 7) has about 33000 nodes, showing that this parameter can play an important role in controlling the size of the index. However, in our other experiments (not reported here), we realize that among the above parameters, this has the least impact on the index size.

Finally, let us examine an example of how we can get a highly specific index for a given query. Let the query Q under consideration be

//open_auction[/bidder]/itemref⇒item/id,

which finds all items that are being auctioned and have at least one bidder. By indexing only the tags (and attributes) site, item, open_auctions, open_auction, bidder, id, itemref, setting $k_{fwd} = 1, k_{back} = \infty$ and tree depth $= 1$ (Defn 8), we get an index with only 17 nodes. This index covers the above query.

## 5.2 Performance on Queries

In this subsection, we wish to demonstrate that (a) by appropriately choosing a covering index, we can speed up the performance of a set of branching path expression queries considerably and (b) we can tradeoff performance with scope: that is, we can pick a more general index that would also cover this set of branching path expressions, but at a lower performance benefit.

We evaluate the performance of our indexes on a set of five queries, shown in Table 2. We pick three representative members from the above set of indexes discussed corresponding to definitions 5, 6 and 8 in Table 1 (with the number of nodes being respectively 40938, 10705 and 17). Let us call these respectively $I_{all}, I_{almost-all}$ and $I_{specific}$. $I_{all}$ covers the whole set of queries. $I_{almost-all}$ covers all queries except Q1. Q1 when written as a branching path expression over the XMark data reads as: count(//person[/profile/education]). Index $I_{almost-all}$ does not cover this since $k_{fwd}$ is set to 1 while building it. Queries Q2, Q3, Q4 and Q5, on the other hand only involve conditions on paths of length 1 (as can be seen from the XMark schema shown in Figure 9). $I_{specific}$

only covers query Q3. While larger index structures inherently cover more path expressions, we see that control can be exercised over *which* sets of path expressions are covered.

We report results for three scenarios:

1. *RELSTORE:* The XML data is stored in a relational system using the *shared* relational decomposition strategy proposed in [18].
2. *NSTORE:* The XML data is stored using a native storage engine, based on a simple breadth-first clustering of objects on pages (we picked the breadth first approach since it almost always performed better for our queries than a depth-first approach). We store node pointers in both directions allowing traversal in either direction.
3. *RELPUBLISH:* The relational decomposition of the data is given and the queries are over an XML view of the data. Here, we pick a different decomposition for the data than the one proposed in [18]. The main difference is that there is one item and incategory table for each continent, whereas in *RELSTORE*, we place all item nodes in a single table and do the same for incategory nodes.

We used DB2 (version 7.1) as the relational engine. The buffer pool size for all experiments with the relational engine was set at 32MB. The experiments were run on a Linux workstation with 256MB of RAM. The native storage engine was implemented using a disk simulator based on the disk model proposed in [19]. In order to get an "apples to apples" comparison, when using a relational engine for the data, we stored the index also using the relational engine, and used a native storage for the index when we used native storage for the data. The index was stored using a fixed relational decomposition in both scenarios RELSTORE and RELPUBLISH. We picked response time as our performance metric. We ran experiments under both cold and warm buffer pool conditions. Since the trends for both cases are similar, we report results for only the warm buffer pool case. For the relational storage cases, we built the traditional value indexes useful for this set of queries.

Figure 10 shows the performance of the three covering indexes under RELSTORE. The X-axis represents the queries, while the Y-axis shows the speedup measured in terms of the ratio between the response time for the data using only traditional relational indexes and the response time using our covering index. Figure 11 and Figure 12 show the performance for NSTORE and
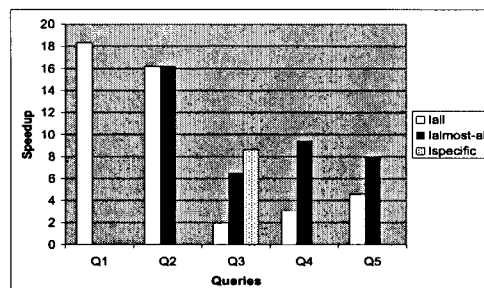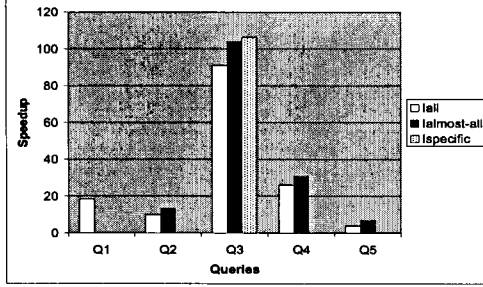


**Figure 10: Speedups for RELSTORE**

142

**Figure 11: Speedups for NSTORE**



**Figure 12: Speedups for RELPUBLISH**

RELPUBLISH respectively. We note the following interesting trends from the above graphs:

1. No matter how the data is stored, there is a considerable benefit to be obtained by using an appropriate covering index. For example, using an appropriate covering index in RELSTORE speeds up performance by factors of up to about 18.

2. In general, the speedup depends on the granularity of the index. For queries Q2, Q3, Q4 and Q5, the speedup obtained using index $I_{almost-all}$ is higher than that obtained using $I_{all}$ since $I_{all}$ is more generic. Similarly, for query Q3, the speedup obtained by using $I_{specific}$ is higher since it is tailored towards Q3. This illustrates the tradeoff involved and indicates a new role for a database administrator in the context of XML — choosing the tags and *idref* edges to be included in a covering index.

3. For a specific query, the speedup using a given index depends on the level of granularity of the index *with respect to this query*. For example, the number of person and homepage nodes in indexes $I_{all}$ and $I_{almost-all}$ is the same although $I_{all}$ overall has more nodes than $I_{almost-all}$. As a result, the speedup using either of them is the same for query Q2. This is not the case for NSTORE since the object clustering is "global"— thus, even though the parts of the index that need to be accessed are the same, they are clustered differently. Similarly, although $I_{almost-all}$ is much more generic than $I_{specific}$, there is no significant difference in the speedup for query Q3 between the two — this indicates that although $I_{almost-all}$ has many more nodes than $I_{specific}$, for the relevant parts of the data, this gap is not significant.

4. Since queries Q1 and Q2 are count queries, evaluating them on a covering index means that the extents of the index nodes need not be accessed if we store the extent sizes in the index nodes. As a result, the speedup obtained for these queries is higher than that obtained for others. This is not the case when native storage is used. One possible reason for this is that the per-query overhead in the native storage implementation is higher and dominates the cost of a simple queries like Q1 and Q2.

5. The speedups for RELSTORE and RELPUBLISH are different. This is to be expected. In both cases, the indexes are stored using a fixed decomposition.
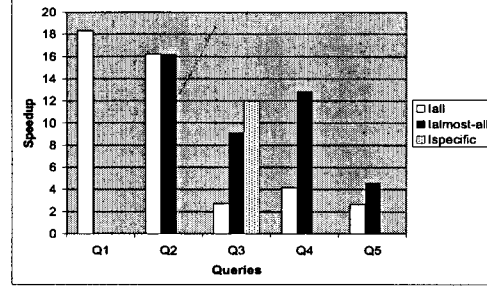
Thus, for queries Q1 and Q2, there is no difference between the two scenarios since the person table is stored the same way in each case. However, for queries Q3 and Q4, the decomposition of RELSTORE is more favorable. This results in a lesser speedup for these queries. For Q5, the decomposition of the data in RELPUBLISH is more favorable. Thus, the speedup is lower there.

6. Speedups in the native storage scenario are higher than the relational counterparts. Our implementation of the native storage is not fine-tuned for performance. As a result, the cache-hit rates could be low and this could affect the data more than the index, which is much smaller. We did not investigate this further since it is not our intention to compare native storage with relational storage. Rather, our goal is to examine the effect of a covering index in each case.

## 6. RELATED WORK

There has been a considerable amount of work on indexing for semi-structured/XML data [4, 6, 7, 9, 11, 12, 17]. Almost all of this work has concentrated on indexing simple path expressions. In [16], the authors propose a storage/indexing strategy in which data nodes are partitioned into relational tables by the extent of the DataGuide into which they fall. Based on this partition, search can be pruned by the DataGuide, and only particular tables searched. This data structure can be used to answer branching path expression queries. However, since the DataGuide is accurate only for incoming path queries, in order to test out-going path conditions, joins have to be performed over the data. In addition, they present their results on tree data. Our work generates indexes that *cover* branching path expressions — so, the base data need not be touched. In addition, we handle *idref* edges as well. To our knowledge, ours is the first implemented and experimentally evaluated set of covering indexes for branching path expressions.

The notion of F&B-Index is introduced in [2] in the context of structural summaries. A contribution of our paper is to show that the F&B-Index is useful as a basic construct in the context of covering indexes for branching path expressions, and to propose techniques to control the size of the covering index. In [12, 17], the notion of local similarity is exploited in constructing summary/index structures for semi-structured data/XML. However, the focus there is on simple path expressions.

Finally, we note that the index graph as defined here,

143

as well as the 1-index and DataGuide, are similar in structure to the *quotient graph* of [8], and that such structures are commonly used for summaries of program automata.

## 7. CONCLUSIONS AND FUTURE WORK

Branching path expressions are an important idiom in XML query languages, and based on our experience in the work reported in this paper, it appears that covering indexes are a promising approach to their efficient evaluation.

We demonstrate that the F&B-Index for a given graph can be used as a covering index for the set of all branching path queries that can be expressed over that graph. However, our experiments indicate that this index is often too big to be useful in speeding query evaluation. Accordingly, we introduce mechanisms through which one can specify a wide range of covering indexes that range from extremely focussed and tiny indexes for very targeted classes of branching path expression queries to the fully generic F&B-Index.

We experimentally evaluated these covering indexes in the context of native XML storage and two variants of relational storage for XML data. In all cases, we observe that the speedup due to covering indexes can be significant. Our study clearly demonstrates some of the tradeoffs involved in picking a covering index. In particular, a covering index that is more generic can handle a larger class of branching path expressions, while an index that is more specific handles fewer branching path expressions but is more accurate for the queries it does cover.

A number of interesting open problems remain that we hope to explore in future work. For example,

- *Index Selection*: The problem of picking an optimal set of covering indexes in order to handle a given query workload naturally arises in this context.

- *Integration with Value Indexes*: Value based conditions are crucial in querying any kind of data and so it is important to integrate structure indexes like our covering indexes with traditional value indexes.

- *Updates and Bulk Loading Algorithms*: Clearly, before these indexes can be deployed we will need efficient index building and updating algorithms.

- *Hierarchies of Covering Indexes*: This item is inspired by the analogy with summary tables for multidimensional workloads. There, one typically defines a hierarchy of summary tables, where higher level summaries are most efficiently computed from lower level tables. It would be interesting to explore whether or not analogous ideas will be useful in our context.

## 8. REFERENCES

[1] Xmark: The xml benchmark project. http://monetdb.cwi.nl/xml/index.html.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[3] J. Clark and S. DeRose. XML path language (XPath) 1.0. W3C recommendation. World Wide Web Consortium, http://www.w3.org/TR/xpath, Nov. 1999.

[4] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of VLDB*, 2001.

[5] S. DeRose, E. Maler, and D. Orchard. The XLink standard. World Wide Web Consortium, http://www.w3.org/TR/xquery, Nov. 1999.

[6] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, 1997.

[7] R. Goldman and J. Widom. Approximate DataGuides. In *Proc. of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 436–445, January 1999.

[8] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 264–274, Victoria, British Columbia, Canada, 4–6 May 1992.

[9] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB*, 2001.

[10] R. Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[11] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT: 7th International Conference on Database Theory*, 1999.

[12] S. Nestorov, J. Ullman, J. Weiner, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 79–90. IEEE, April 1997.

[13] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[14] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conf.*, LNCS 104, pages 167–183. Springer-Verlag, Karlsruhe, March 1981.

[15] Open Directory Project. DMOZ open directory project. http://www.dmoz.org.

[16] F. Rizzolo and A. Mendelzon. Indexing XML data with ToXin. In *Proc. of WebDB 2001*, 2001.

[17] R.Kaushik, P.Shenoy, P.Bohannon, and E.Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proceedings of ICDE*, 2002.

[18] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 302–314, 1999.

[19] E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Joint International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '98/Performance '98)*, pages 182–191, Madison, WI, June 1998.

[20] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proceedings of STOC*, pages 1–9, 1973.