Efficient Indexing of XML Data

Raghav Kaushik **Pradeep Shenoy** **Philip Bohannon** **Ehud Gudes**

Abstract

We explore efficient and easy-to-update indices on semistructured data for use in answering XML path queries, including full regular expression queries. In this paper, we present a parameterized family of approximate indices, offering a smooth tradeoff between index accuracy and size. Our family of indices is based on the notion of k-bisimilarity, a generalization of the well-known notion of graph bisimilarity. The parameter k controls the accuracy of the index and as k increases, the index converges to the bisimulation-based 1-index. We provide techniques that allow us to efficiently extract exact answers from our approximate indices. We present an assessment of path evaluation on large samples of real-life semistructured data. Our experiments show that our approximate indices are viable for processing path queries for exact answers, while using significantly less space than the full bisimulation-based index.

Graph bisimilarity has been well-researched in the verification community and has also been proposed for semistructured data indexing. However, there has been no known previous work on updating the bisimulation partition. We prove properties of bisimilarity and k-bisimilarity that we then use to develop efficient update algorithms for our approximate index structures and the previously proposed 1-index. Given the widespread use of the bisimilarity partition, these algorithms may be of use outside of data management. Further, preliminary results indicate that our update algorithms will work efficiently on real-world data.

1 Introduction

Recently, a variety of query languages have been proposed for XML or other semistructured (XML/SSD) data [2, 3, 9, 8, 31, 30, 11]. As noted by [10], these languages consist of *pattern language* and a *construction expression*. Fundamental to the pattern language is the *path expression*, which selects data based on the labels associated with an incoming (or outgoing) path from an element or data node. While simple path expressions consisting of a sequence of child relationships were popularized in object-oriented query languages, most path languages proposed for querying semistructured or XML data have significantly greater expressive power [26]. Two proposed XML/SSD query languages, Lorel [30] and UnQL [3], allow regular path expressions. Another path language for XML data with significant expressive power is the XPath W3C standard [7], which is used in the transformation language XSLT [6]. Regardless of the precise form, efficient evaluation of complex path expressions is a key factor in the timely processing of queries on XML/SSD data.

We focus on the problem of indexing XML/SSD databases for path queries. Since, like the DataGuide [15] and 1-index [23], we are indexing the tag structure of the data, our index structures can also serve as a discovered structural summary of schema-less data [13, 24, 15]. As noted in [15], such summaries can be used 1) as a schema to aid in user understanding, 2) to store a statistical summary of database content for query optimization, and 3) as an index for path-expressions on tag structure. While we focus on the third use, our contributions apply also to the first two. In particular, the index structures we propose can serve as "graph schemas" in the sense of [4].

1.1 Motivation

With the widespread adoption of XML as a standard for information representation on the web and as a language for data exchange, XML-enabled databases and XML repositories will be expected to store, index, update and search increasingly large volumes of XML data. When the database is large and subject to dynamic change, the index structures (or other structural summaries) must be efficient to create and reasonable in size, in addition to supporting efficient search and update.

Our primary aim is to generate configurable indexes in which the above properties can be tuned to take into account the complexity of the data, as well as the requirements of the user. Known indexing techniques have severe limitations in one or more of the above respects. In particular, the DataGuide [15] can, on highly cyclic databases, have an exponential size. The 1-index [23] is at most linear in the size of the database; however, when the size of this index approaches the size of the database it is unlikely to improve path evaluation time. Further, so far, no non-trivial update algorithm is known for the 1-index.

Other desirable qualities include limited size relative to the base data and the ability to respond efficiently to common query patterns. We seek to develop index structures and algorithms for building and maintaining path indices on large XML/SSD databases, which maximize path evaluation performance and otherwise meet the above goals without the size drawbacks of the DataGuide or 1-index.

1.2 Contributions

The first contribution of this paper is the introduction of a parameterized family of *approximate* index structures, the A(k)-indices, that, to a great extent, meet the above goals. Built on a principled theoretical framework, our index structures are smaller than the 1-index, and further, they allow the user control over index size while trading for worst-case performance. Also, a useful feature which distinguishes the A(k)-index from the 1-index is a more tightly bounded cost of index update. Our performance results show that, for several classes of simple path expressions, evaluation on the (smaller) A(k)-index is significantly faster than on the 1-index. And except for certain queries which traverse an extremely large number of long paths, the A(k)-index is competitive with the 1-index.

The second contribution of this paper is the design of efficient update algorithms for the A(k)-index and the bisimulation-based 1-index. Although the bisimulation partition, on which the 1-index is based, is well-studied in the verification community (see, for example, [12, 32, 33] and references), we are not aware of any published algorithm to efficiently update the partition following changes in the underlying data graph. Thus, our update algorithms may have applications outside the field of database management. We also present theoretical results that support the correctness of the above update algorithms, and also lead to more efficient computation of the simulation partition [22].

The third contribution of this paper is a preliminary performance study of regular path evaluation on the A(k)-index and the 1-index, focusing on large XML documents drawn from real-world data sets. Finally, we show that for a number of data sets, the similarity-based partition, though smaller than the bisimilarity partition, does not appear to be enough smaller to justify its much higher cost of creation, answering a question apparently left open by [23].

The rest of the paper is organized as follows. Preliminary definitions and assumptions are presented in Section 3. In Section 4, we introduce the A(k)-index. In Section 5, we discuss index construction algorithms. We describe query evaluation techniques for approximate indices in Section 6. In Section 7, we present performance results for path evaluation on the A(k)-index. Update algorithms for the A(k)-index (as well as, for the bisimulation partition and therefore the 1-index) are explained in Section 8. We discuss related work in Section 2 and conclude with a summary and discussion of future work in Section 9.

2 Related Work

Two previous proposals for indexing semistructured data for path expressions are reported in [15] and [23] In [15], the data graph is summarized by a graph called the DataGuide. In the *strong DataGuide*, for which algorithms are given in [15], each path in the data is represented by a single path in the DataGuide. A node in the DataGuide represents all the data nodes which are reachable by the paths reaching the index node. The list of these data nodes is maintained as the *extent* of the index node. The DataGuide is useful both for gaining insight into the overall structure of the database by using its graph structure (including statistics about paths, nodes, etc.), and for answering regular expression queries by using its extents. However the DataGuide approach has two problems: first, the DataGuide graph size (as noted by the authors of [15]) may be potentially exponential in the size of the original database, and second, the total size of the extents may also in the worst case be exponential. Thus, on large, complex databases it may not be possible to even compute the DataGuide; however, the A(k)-index is capable of efficient query processing on these databases even while remaining a fraction of the underlying database in size.

One may think of the strong DataGuide as a deterministic automaton which recognizes the data graph. [23] proposed a non-deterministic automaton representation of the DataGuide, called the 1-index, which may have repeated paths. The 1-index also has extents; however these extents do not overlap, and hence the size of the 1-index is at most linear in the size of the original database. The 1-index tries to group together nodes that have identical sets of paths from the root, and uses simulation [22] and bisimulation [28] as refinements of this grouping. However, neither [23] nor apparently previous work on simulation or bisimulation (for example [12, 32, 33]) addresses the problem of 1-index update, leaving only the alternative of recomputation, a problem we address. Further, in the case that the size of the bisimulation partition approaches the size of the underlying data graph, it may no longer improve on the underlying database for path expression evaluation. By contrast, our indices can be varied in size, and in our experiments provided some of their best performance for values of k at which they were significantly smaller than the 1-index.

[25] propose the Representative Objects as a structure that can be used for schema discovery as well as for path queries. In fact, as they mention, the Full Representative Objects(FROs) are implemented as DataGuides. They also propose an approximation of the FROs, the k-Representative Objects. This can be used to implement approximate DataGuides [16]. The A(k)-index is similar to the k-RO in that every path of length k present in the database is present in their "automaton summary". However, the A(k)-index (1) is non-deterministic and thus smaller (2) has efficient construction and update procedures and (3) can be used to provide precise answers to certain regular path queries such as those beginning with "_*". Other approximate versions of the DataGuide appear in [16], but except for a case that implements the k-RO, do not appear to preserve properties of the path structure in the database to allow efficient path evaluation.

The simulation and bisimulation techniques mentioned above come from the verification community, where they are used to create "compressed" versions of state space graphs, which preserve some properties and behavior of the state space. These compressed graphs could then be analyzed with respect to these properties hence used to prove statements about the entire state space. We note that the index graph as defined here, as well as the 1-index and DataGuide, are almost identical in structure to the *quotient graph* of [17, 5], and that such structures are commonly used for summaries of program automata. Further information about this body of work on graph summarization can be obtained by following pointers in the above papers and [12, 32, 33].

3 Foundations

In this section, we describe how we model XML data and path queries, introduce the concept of index graphs and explain some related terminology.

3.1 Data Model

We model XML data as a rooted, directed graph $G = (V_G, E_G, root, \Sigma_G, label)$ whose nodes (as opposed to edges to follow XML usage, see [14]) are labeled with a string-literal via the *label* function. Each element in an XML document is represented by a node n, with the element tag $label(n)$. Σ_G is the set of all literals for the graph G. Edges in E_G represent either *tree edges* which are element-subelement relationships, or *IDREF edges* which are created with the ID/IDREF construct of XML. Attribute values are also nodes in the graph. We do not distinguish tree edges from IDREF edges, nor do we distinguish attributes from subelements, though such distinctions may be easily made. Further, we do not preserve the document order of sibling nodes, and thus do not support pattern matching along this dimension, as provided in some query models [7, 26]. (We note that document ordering is typically ignored for data-exchange applications of XML [1].)

The *root* element corresponds to an artificial root of which the "root element" of the XML document appears as a child, similar to the "document" node which appears in the W3C Document Object Model [34]. The root node has a distinguished label, ROOT.

3.2 Path Expressions

A *path* in G is a sequence of nodes, $o_1 \ldots o_p$ such that an edge exists between node o_i and o_{i+1}, for $1 \leq i \leq p-1$. A *label path* is a sequence of labels $l_1 \ldots l_p$, and a node-path *matches* a label path if $label(o_i) = l_i$, for $1 \leq i \leq p$. A k-path is a label path with $p \leq k$. A *simple path expression* is a label path in which the first label is ROOT.

A *regular path expression*, R, is defined in the usual way in terms of sequencing (.), alternation (|) and repetition ($*$), as shown below. A path expression, P, associates the result of the expression with a variable, presumably to be used in a wider query.

$$
\begin{array}{l}
R ::= \Sigma \mid R.R \mid R|R \mid (R) \mid R^* \\
P ::= ROOT.R \;\; x
\end{array}
$$

We include the symbol _ as an abbreviation for $l_1|l_2|\ldots$ that covers each $l_i \in \Sigma_G$. Path expressions always begin at the root element of the database graph, and the initial ROOT label is implied if omitted. We refer to the regular language specified by P as $L(P)$. When a path expression is evaluated against a data graph G, it returns for x each node in V_G with an incoming label path in $L(P)$. In the current work we do not consider chained or branching expressions. Adapting the A(k)-index to these expressions is left as future work. For the rest of the paper, by the term *query* we mean path expression queries of the above form.

3.3 Index Graphs

The index structures we propose are similar in spirit to the DataGuide [15] and the 1-index [23]. Both these techniques try to put together the data nodes into groups that are in some sense *units* in the results of path queries. Once they group together the data nodes in a specific manner, they build an index by associating an index node for each group and connecting up these index nodes by edges appropriately. Thus, the index structure here, unlike conventional value indices and path indices, is a rooted directed graph by itself. The index nodes are also labeled. The group of data nodes associated with an index node is called its *extent*.

In more detail, the DataGuide treats the data graph as a nondeterministic automaton where the root node is the start state, and outputs a deterministic automaton that is equivalent to it. It does so by the usual construction. Thus, by traversing a label path $l_1.l_2 \ldots l_k$ in the DataGuide, we arrive at an index node whose extent is the result of this path expression query as evaluated on the data graph. In this process, a data node can be associated with the extent of more than one index node. Unfortunately, this can cause a space blowup.

In order to overcome this problem, the 1-index adopts a slightly different approach. It partitions the nodes in the data graph and associates an index node with each of these partitions. Since the 1-index deals strictly with partitions, the size is bounded by the size of the data graph. However, the resulting index structure need not be deterministic.

In this respect, our approach is similar to that of the 1-index. We partition the data nodes in some manner and associate index nodes with each partition. We call any such index structure an *index graph*. More formally, on a data graph $G = (V_G, E_G, root, label)$, an index graph, $I(G)$, is a mapping of nodes of the data graph to nodes of the index graph, as follows:

1. For a node $v \in V_G$, $A = I(v)$ is a node of the index graph. The set of data graph nodes which map to A, $\{v : I(v) = A\}$, the *extent* of A, is written as $ext^I(A)$.

2. If there is an edge from v to v' in the data graph, then there is an edge from $I(v)$ to $I(v')$ in the index graph (The converse may not be true.)

3. If *root* is the root of G, then $I(root)$ is the root of $I(G)$.

4. If all nodes in the extent of an index node have the same label, then this label is associated with the index node, otherwise the index node's label is undefined.

In the rest of the paper, we assume, for ease of exposition, that graphs have a single root with a distinguished label. Our results can be extended to the more general case without much difficulty.

Example 1 *Figure 1(a) is an example of an index graph obtained by partitioning the data nodes based on their labels. In the example, the data graph is denoted by G and the index graph is denoted by $I(G)$. The letters within the nodes denote the labels while the numbers to the sides denote the node numbers (or object ids). The numbers shown to the side of an index node denote the node numbers of the data nodes in its extent. Since all nodes in the extent of an index node have the same label, we associate this label with the index node itself. Node number 1 is the root of the data graph. Its corresponding index node is the root of the index graph.*

Notice that there is an edge between the index node containing data nodes 2 and 3 and the one containing 4 since one of them, 2, has an edge to 4 in the data graph.

For the rest of the paper, we consider only index graphs in which nodes with differing labels are in the extent of different nodes in the index graph, that is, $label(a) \neq label(b) \rightarrow I(a) \neq I(b)$. When we refer to the label of an index node, it refers to the common label of all data nodes in its extent. Thus, it is meaningful to talk about label paths in the index graph. We always use lower case letters to denote data graph nodes and capital letters to denote index graph nodes.

The *target set* of a path expression P on the data graph is the set of nodes in the graph that matches P. The *index result* of P is the union of the extents of nodes in the target set of P on the index graph.

Any index graph has the following properties.

1. Every label path in the data graph is represented in the index graph too. And hence,

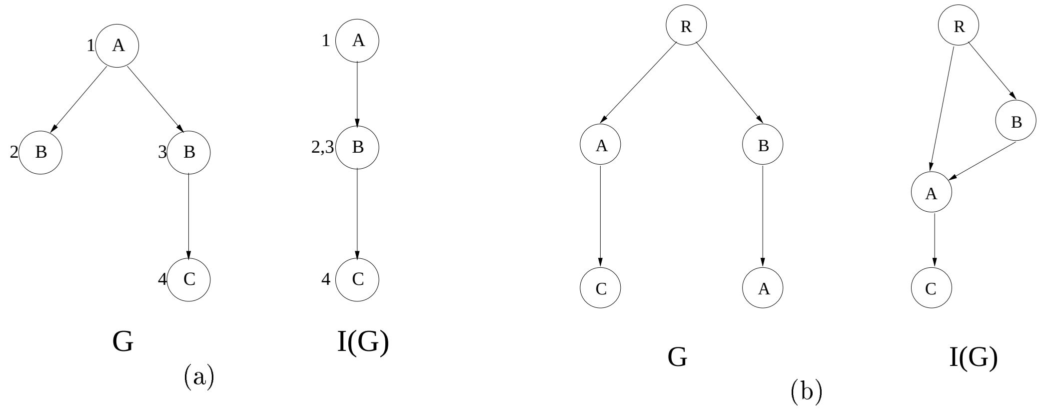


Figure 1: **Example index graph and index graph with spurious paths**

2. For any path expression P, the index result of P always contains the result of P as evaluated on the data graph.

Note that these properties are one-sided—the index graph may have paths not appearing in the data, and the index result for an expression may be a proper superset of the target set for that expression . We illustrate this with an example.

Example 2 *Consider a data graph and its index graph as shown in Figure 1(b). The index graph groups data nodes by label. The extents and the object ids are omitted for clarity. Notice that while the index graph shows an R.B.A.C path, this is absent in the data graph.*

We call an index graph *precise* with respect to a set of path expressions if for any path expression P in the set, the index result of P is identical to the target set of P. An index is *precise* if it is precise with respect to all such sets. The 1-index is a precise index graph.

We now define some additional terminology.

1. For any graph G (data or index), an edge from u to v is denoted $u \rightarrow v$.

2. The label of u is $label(u)$.

3. The *successors* of a node are defined as

$$succ(v) = \{u : v \rightarrow u\}$$

   The *predecessors* of v are defined symmetrically.

4. For a node set S,

$$Succ(S) = \bigcup_{v \in S} succ(v)$$

   $Pred(S)$ is similar.

4 The A(k)-index

We first briefly motivate the notion of bisimilarity and explain how it is used to build the 1-index. Then, we move on to explain the notion of k-bisimilarity. Note that the A(2)-index is not related to the 2-index of [23]. The latter is a specialized index structure for queries of the form $\_ * x, x\ R\ y$ where x and y are variables and R is an arbitrary regular expression, while the former is an approximation of the 1-index based on paths of length 2.

The definitions below are those given for a *backward* bisimulation; however, bisimulation is typically defined in the literature so that nodes are classified based on the *outgoing* paths (a *forward bisimulation*). We note that these definitions are symmetric, and simply use "bisimulation" to refer to the backward bisimulation in this paper, since this is the most directly useful partition for evaluating path expressions.

4.1 Bisimulation and k-Bisimulation

In order to ensure that an index graph is precise, the nodes must be partitioned such that whenever two nodes are grouped together, they have the *same set of in-coming paths* from the root.

If we obtain the coarsest such partitioning, the resulting index graph would have fewest nodes among all precise index graphs. But this problem is known to be $PSPACE$-complete. The solution is to go in for *refinements* of this ideal partitioning [23]. In order to obtain such a refinement, the notion of bisimilarity [28] is used.

The intuitive definition of bisimilarity is as follows. Two nodes u and v are allowed to be bisimilar only if they have the same label. In order to ensure that they have the same set of incoming paths from the root, we stipulate a stronger condition: for any parent u' of u, the paths that come into u from the root via u' must come into v from some parent v' who is bisimilar to u', and vice versa.

This latter condition, through its recursive application, ensures that bisimilarity implies equality of the set of incoming paths. Since the root has a unique label, this means the set of incoming paths from the root is also the same.

In order to define bisimilarity formally, we first define the notion of k-bisimilarity [21] which tries to ensure a weaker condition: if two nodes u and v are k-bisimilar, then the set of k-paths[1] into them (not necessarily from the root) is the same.

Definition 3 $\approx^k$ *(k-bisimulation): This is defined inductively.*

1. *A root is* 0*-bisimilar only to a root of the same label.*
2. *For two non-root nodes, u and v, $u \approx^0 v$ iff u and v have the same label.*
3. *Node $u \approx^k v$ iff $u \approx^{k-1} v$ and for every parent u' of u, there is a parent v' of v 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. As k increases, the partitioning induced by this equivalence relation keeps getting refined until at some point, it reaches a fixed point. The equivalence relation associated with this partitioning is called *bisimilarity* or *maximal bisimulation* denoted by $\approx^{max}$. This is equivalent to the k-bisimulation for the smallest value of k such that $\approx^k$ matches $\approx^{k+1}$ [21].

The index graph obtained from the bisimilarity partitioning is called $Bisim(G)$. This is one form of the 1-index as defined in [23].

Example 4 *Figure 2 shows a data graph and its index graphs based on 0-bisimilarity, 1-bisimilarity and 2-bisimilarity. In this case, the 2-bisimulation is actually the maximal bisimulation.*

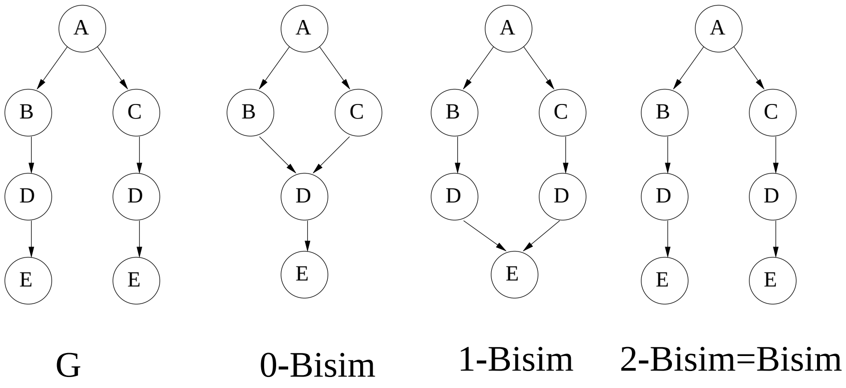


Figure 2: **Sequence of k-bisim index graphs tending to full bisim index**

The full bisimulation index is precise for *every possible* path expression, no matter how complex it is. We expect that complex path expressions will be rare in practice, thus it may be desirable to obtain index graphs with smaller sizes which though not precise in general, are precise for simpler, shorter queries. This motivates the search for approximate indexes where, depending on the query workload, size and path evaluation time for simpler expressions can be traded against the set of queries for which the index is precise.

4.2 The A(k)-index

The A(k)-index is simply the index graph constructed from the k-bisimulation partition, $\approx^k$. As Example 4 shows, the A(k)-index is a generalization of the full bisimulation index.

We now describe some properties of these index graphs.

Property 1 *If nodes u and v are k-bisimilar, then the set of label-paths of length $\leq k$ into them is the same.*

This property follows directly from the definition of the A(k)-index.

Property 2 *The set of k-(label) paths into any A(k)-index node is the set of k-paths into any node in its extent.*

Property 3 *The A(k)-index is precise for any simple path expression of length less than k.*

This follows from Property 1 and the fact that all simple path expressions begin with the distinguished label ROOT.

Property 4 *The $(k+1)$-bisimulation is either equal to or is a refinement of the k-bisimulation.*

From this property, one can see that by varying k, we obtain a smooth A(k)-indices that increase in size, converge to the 1-index, and that each subsequent index is derived by partitioning some of the nodes in the previous index.

The final property of the A(k)-index that we present here relates to the maximum cost of keeping the index graph updated after an update to the underlying data graph. Essentially, the effect on the index of any update in the data graph is limited locally to a "neighborhood" of distance k.

[1]i.e., paths of length $\leq k$, where length is measured in terms of number of edges

Property 5 *Suppose an edge is added or deleted between u and v. Let nodes x and y be at distance $> k$(in terms of edges) from v. Then, the update does not affect the k-bisimilarity relationship between x and y.*

This limit to the effect of an update is unlike bisimulation where the whole graph can be affected. We will describe this in more detail in section 8.

5 Index Graph Construction

Here, we describe our index construction algorithm for A(k)-index, after introducing the important notion of stability.

5.1 Stability

There is an alternative definition of bisimilarity based on the property of *stability*.

Definition 5 *Given two sets of data graph nodes A and B, A is said to be stable with respect to B if either A is a subset of $Succ(B)$ or A and $Succ(B)$ are disjoint.*

Thus, if A and B are nodes in an index graph, A is stable with respect to B, and there is an edge from B to A, then every data graph node in the extent of A has a parent in the extent of B. This property is crucial for preciseness of an index graph. In the above, when talking about stability of index nodes, we actually refer to the stability of their extents.

We call a partitioning of the nodes *stable* if, for every pair of partitions p_1 and p_2, p_1 is stable with respect to p_2 (again, when talking about the stability of partitions, we actually mean stability of the corresponding node sets). The coarsest partitioning of the nodes of a graph that is stable is the maximal bisimulation. Here, the partitioning is constrained not to group together nodes of different labels. In other words, any stable partitioning of the data nodes is either equal to or is a refinement of the maximal bisimulation.

In a similar vein, for the k-bisimulation, we have the following. Let us call a partitioning P_1 of the nodes stable with respect to partitioning P_2 of the (same) nodes if every partition in P_1 is stable with respect to any partition in P_2.

The $(k+1)$-bisimulation is the coarsest refinement of the k-bisimulation that is stable with respect to it. Let us again observe here that if the k-bisimulation equals the $(k+1)$-bisimulation, then the k-bisimulation is stable with respect to itself, a requirement for the bisimulation.

5.2 Computation of A(k)-index

The key observation here is that given the k-bisimulation, the $(k+1)$-bisimulation can be obtained by making a copy of the k-bisimulation and splitting partitions in this copy until stability with respect to the partitions of the k-bisimulation. The procedure is sketched in Figure 3. The data structures used to implement this are based on the ones used in [27].

The algorithm maintains a partition of the data nodes, $\mathcal{Q}$. In each iteration, the algorithm stabilizes $\mathcal{Q}$ with respect to a copy of itself(line 9 implements stabilization of a partition w.r.t another). This process is repeated k times. Thus, if the initial partitioning is one by label, i.e is the A(0)-index partitioning, then the algorithm outputs the partitioning corresponding to A(k)-index. In the figure, $\mathcal{X}$ is set to a copy of the $\mathcal{Q}$ partitioning.

This approach takes time $O(km)$ where m is the number of edges in the data graph. The memory it takes is $O(m)$. In our experiments on two real-world data sets, the k value at which the A(k)-index equals the full bisimulation is small (we will see the details in section 7).

```
procedure compute_k_bisim(graph G)
begin
Q is initialized by partitioning the data nodes by label
X is also initialized in the same way
for i = 1 to k do
   //stabilize Q w.r.t X
   foreach partition X in X do
      compute Succ(X)
      //split w.r.t Succ(X)
      foreach partition Q in Q do
         replace Q by Q ∩ Succ(X) and Q − Succ(X)
   if there was no split then
      break
   X is set to a copy of Q
end
```

Figure 3: **A(k)-index computation**

The full bisimulation can also be computed in this manner, but its worst case complexity can be as high as $O(mn)$ where n is the number of nodes in the data graph. Paige and Tarjan in [27] propose an algorithm(henceforth referred to as PT) where the worst case complexity is $O(m \log n)$.

Their algorithm maintains two partitions $\mathcal{X}$ and $\mathcal{Q}$ with the following invariants.

1. $\mathcal{Q}$ is a refinement of $\mathcal{X}$
2. $\mathcal{Q}$ is stable w.r.t $\mathcal{X}$

The details are shown in Figure 4. A *compound_X_block* is an X partition such that it is properly refined in $\mathcal{Q}$. In each iteration, some compound_X_block X is chosen. One of its component Q partitions is chosen as splitter and $\mathcal{Q}$ is stabilized w.r.t this partition. The compound X partition is replaced by its two parts, the splitter and the rest. This process is repeated till $\mathcal{X}$ and $\mathcal{Q}$ become the same. Notice that by maintaining the above invariants, this termination condition ensures that the resulting $\mathcal{Q}$ partitioning is stable. In fact, it is the maximal bisimulation partitioning.

```
procedure compute_bisim(graph G)
begin
Initialize Q by partitioning the data nodes by label
Initialize X to a single partition with all data nodes
Initialize compound_X_blocks appropriately
while there is a compound_X_block do
   pick partition X ∈ X which is compound
   pick partition Q ∈ Q contained in X and smaller than half its size
   replace X in X by Q and X − Q
   stabilize Q w.r.t Q and X − Q
end
```

Figure 4: **Bisimulation computation**

The main gain in this algorithm comes by picking a small Q partition to split $\mathcal{Q}$. By maintaining proper data structures, the above can be implemented in $O(m \log n)$.

Let us note that the above optimization cannot be extended easily for the purposes of building the A(k)-index. Hence, we did not directly adapt PT to compute A(k)-index.

6 Path Evaluation with Approximate Index Graphs

In this section, we discuss the evaluation of path expressions with the A(k)-index. We begin by introducing our (simple) regular path language.

6.1 Path Expression Evaluation

Path expression evaluation is modeled on the straightforward approach to evaluating regular path queries on a graph, as follows:

1. an NFA, A, is created according to the regular expression $P = ROOT.R$.
2. A is then *run* on the index graph, i.e., the index graph is traversed, while making corresponding state transitions in the automaton for matching (node-label, transition-label) pairs
3. A table is kept to avoid repeatedly visiting a data node in the graph in the same state of the automaton, thus avoiding cycles.
4. When a node in the index graph is reached while the corresponding automaton state is an *accept* state, the index node's *extent* is added to the final result.

6.2 Validation

When a path expression for which a given A(k)-index is not precise is evaluated as above, the index result contains all correct answers, but may also contain some false positives. Let M the portion of the index result set for a path expression P within which false positives may occur. We term M the *maybe set* for this evaluation. We deal with the possible false positives by *validating* the nodes of M against the original data graph. This validation is handled by a reverse execution of the automaton beginning with each node in M, and is made more efficient by reusing information when validating several nodes. Details of this technique are discussed in Section 7.

To determine which nodes need to be validated, we use Property 1 which states that two data nodes in the same extent have identical sets of incoming label-paths of length $\leq k$. Thus, even if the A(k)-index is not precise for an automaton P, it is accurate for portions of the result set which are accepted along paths of length $\leq k$ from the root. Thus, when our query automaton accepts an index node N along such a path, all data nodes in $ext[N]$ are directly marked as validated.

We now prove a theorem that significantly expands the cases in which index node extents can be directly marked valid. Consider a k-path $p = l_1, l_2, \ldots, l_j, j \leq k$ and an A(k)-index node N, such that p is the *suffix* of one or more paths to N.

Theorem 6: *If all paths $p'.p$ that exist in the database are in $L(P)$ for a path expression P, then all data-graph nodes in the $ext[N]$ are in the target set of P.*

(The proof of Theorem 6 appears in the full version of the paper.) To see how this result allows more index nodes to be marked as valid, recall that all data graphs have a unique root with label ROOT. Second, for data graphs derived from XML documents, that root node will have a single successor node (other than, possibly, XML processing instructions). Say that the label of the successor to the root node is DOC. In this case, since all successful queries will have to initially match ROOT.DOC, Theorem 6 shows that the extents of nodes matched on paths of length $\leq k+1$ from the root may be directly marked as valid.

This technique can be further generalized with knowledge of constraints on the database. One such constraint might be that the only paths to label B pass through $ROOT.A$. For example, all paths to label

GPA might be through $ROOT$.STUDENTS. In this case, Theorem 6 indicates that when marking nodes of type B that are the answer to a query, as valid, only the length of the accepting path *following the STUDENTS tag* need be considered, and if this length is $\leq k$, the B node can be marked as valid.

While our current implementation does not take advantage of general constraints, we do make use of the subcase where the path expression is of the form $\_*.R$, a common construct. That is, if a node N in an index graph is recognized by an automaton due to some simple path $p.p'$, such that p is matched by the $\_*$ portion of the path and p' by the remainder, and the length of p' is $\leq k$, then the data nodes in the extent of N are directly marked as validated.

The result of these observations is that many simple queries can be evaluated accurately on the A(k)-index, and this fact is demonstrated on queries on real-world data in the next section.

7 Performance

In this section, we evaluate the performance of the A(k)-index when used to evaluate path expressions on large XML datasets.

7.1 Data

The experiments described in this section use XML data drawn from two web sites supporting querying and browsing of that data. The first source we used was the Internet Movie Database (IMDB) [18], and the second is the Open Directory Project (ODP) [29]. We now briefly describe these data sets and the subsets we extracted for our experiments.

IMDB

The IMDB database is available as a collection of ASCII files, and we selected it since it was identified in [15] as a highly cyclic database likely to stress indexing algorithms. The portion of the database is organized around *movie* elements and elements for classes of people who appear in movie credits–e.g., *actor*, *director*, *composer*, etc. The data for each element is as one might expect. Cyclicity is arises since each movie element has pointers to individuals who worked on that movie, and each element representing an individual has pointers to the movies on which s/he worked.

To create a subset of this database, we chose a small subset of movies and all the people associated with these movies. We then repeated the following steps: 1) sample all movies associated with these people and 2) add these movies and their associated people to the database. Once the desired database size was reached, dangling pointers were removed.

ODP

The second source of semistructured data that we used was the Open Directory Project [29] data. This data consists of an internet directory similar to the directory provided by Yahoo, and has a hierarchical classification of topics. We extracted subsets of this data by choosing a set of top-level topics, in this case "Shopping", "Regional", "Society",and "Home" forming the "SHSR" data set. We then selected only topics from the original data set which matched one or more of these topics somewhere in their body. We manipulated the original RDF format so that inter-topic references would appear as IDREF edges, and selectively replaced <*Topic*> tags with the word which served as the topics title. That is, <Topic Name="home"> would become <Home NodeType="Topic">. We note that in a complete query processing system embodying such indices, transformations like this can be made via simple mappings during index creation and analogous transformations to path expressions. Further, we note that topics which appear in a path expression but not the index can be replaced with the tag "Topic" and the final result set filtered on the original topic. (We do not discuss further this filtering.) We varied the complexity of the index graph by only applying the above transformation to a subset of those tags that appeared most frequently in the data, e.g., the top 200 frequent tags.

7.2 Index Sizes

Figure 5 shows the sizes of the simulation, bisimulation indices and the A(k)-index on a sample IMDB database of size 190,000 nodes. The figure shows the number of nodes in the index graphs on the y-axis against the k-parameter of the A(k)-index. The indices are well within 10% of the database size. However, note that the simulation index is not much smaller than the bisimulation index size. This was consistently the case across a variety of sample databases, and for this reason we do not consider simulation-based indices in detail in the paper.

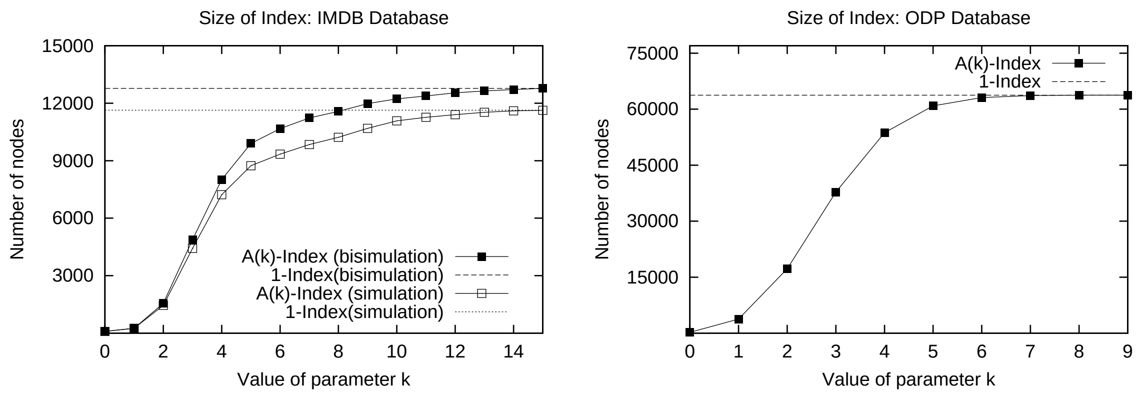


Figure 5: $A(k)$**-index sizes**

There are two important features about this figure: Firstly the A(k)-index is equal to the bisimulation for a relatively small value of $k = 15$. . Secondly, the A(k)-index sizes increase smoothly from $k = 0$ to k_{max}.

The second graph (in Figure 5) shows the A(k)-index sizes on a subset of ODP data generated using the top frequent tags of the SHSR dataset. The size of the database is 143,000 nodes built on 41,445 topics. The variation in the sizes of the indices is similar to the case of the IMDB data. However, note that the final bisimulation is a significantly higher fraction of the database – nearly 50%. Thus the space-saving properties of the A(k)-index are significantly more important for this dataset.

7.3 Query Evaluation Cost Model

We use the abstract cost of *number of nodes visited* in the index graph for answering a query. In the case of the approximate indices, we include the number of graph nodes visited for validating our approximate index result. Note that the number of nodes visited is a reasonably accurate measure of CPU time of query execution.

We assume an in-memory model–i.e., disk-access cost is not considered. This is for two reasons: Firstly, we wish to compare the performance of a family of increasingly accurate and larger indices. The number of nodes visited is an effective way of comparing the size-cost tradeoffs of these indices. Secondly, there is no clear consensus for a storage model for semistructured data. To consider disk-access cost we need to address complex and partly orthogonal issues regarding data storage that are beyond the scope of this paper.

7.4 Path Evaluation Strategies

While this work focuses on indexing rather than optimization (see, for example, [19]), there are times when the straightforward automaton evaluation strategy generates poor results. Thus, we introduce two alternate mechanisms to ensure that individual queries are evaluated with reasonable plans. In particular, an initial $\_*$ construct is commonly used to find a pattern anywhere in the graph, yet with the above execution scheme the entire graph will be explored with this pattern. To this end, we allow the nodes of an index or data graph to be indexed by label (the "L-index" of [20]), and use this index to find the first symbols appearing in R.

The second evaluation option we allow also makes use of an L-index and simulates "backward evaluation" of path expressions. We use the L-index to find the nodes bearing the final label(s) mentioned in R. Then R is evaluated in a reverse manner to determine whether these data nodes match the automaton. The intuition here is that the "end" of the expression may be significantly more selective than the earlier parts, and thus processing in this manner will win. Of course, there are many other feasible execution plans, but their evaluation is beyond the scope of this paper (and we are not aware of prior work addressing cost-based optimization of regular path expressions). We note that, in most cases, *plans and/or additional indices may be applied equally well on the A(k)-index, 1-index and Data Guide* (when it is smaller than the data graph), giving us some some reason to expect that performance results with these limited plans will be indicative of those obtained with the benefit of other plans, statistical estimation, additional indexes, etc.

Note that the backward evaluation strategy and validation of "maybe" nodes are very related. The main difference is the number of nodes which must be checked against the automaton (in fact, backward evaluation is almost exactly the same as using the A(0)-index, except that we count the cost of the L-index as 0). A key observation for efficient backward processing is to reuse information as the reverse automaton is run on different nodes. In particular, these nodes may *share paths or partial paths* to the root.

To implement this, during the executions of the reverse automaton on the data graph, we keep track of all paths on the data graph that led to accept states, and for each node on these paths, the state of the automaton during its visit to this node. In subsequent validation calls for different nodes, if we reach one of the nodes on these paths, in the appropriate automaton state, then we know the validation leads to a "yes" answer, and can terminate the current automaton execution. A similar technique, with minor modifications, can be used to keep track of paths that lead to failure as well.

As discussed in Section 6, we compare the best evaluation mechanism for the A(k)-index with the best mechanisms for the 1-index and graph, respectively. (We assume a perfect cost estimator, simply running all the plans and choose the one with the lowest cost.) An interesting result is that backward evaluation of a query starting from a label-index result is cheaper than forward evaluation for almost all queries, over almost all indices and the data graph (except of course for the rare case in which the query ends in a "_"). We hypothesize that this is due to 1) the fact that all nodes have paths to the root and 2) the fact that some nodes have a large number of outgoing edges (all actors, all topics, etc), but no nodes have a large fan-in, which helps backward evaluation avoid excessive work.

7.5 Path Expression Evaluation Results

We divide path expressions into two classes for the purposes of evaluating index performance, short, and long. Our queries are linear path expressions randomly generated by performing random walks on the A(1)-index for a given data set. "Don't care" symbols (_) were added by choosing a value n between 1 and 3, and adding (_)|(_._)| . . . |(_. . . . ._) so that the last term has n_'s. This construct is chosen to simulate an expression which ignores a few tags at certain levels in the query, perhaps when the structure of the document is imperfectly known(note that on cyclic data, _$*$ accomplishes this very poorly). Further, we add an initial _$*$ to some queries. The intention is that short queries will be answered precisely by the A(k)-index (for most values of k), while the long queries will require validation.

Short queries

In Figure 6 we show the cumulative cost of executing short queries (some with an initial '_*' some without) on the A(k)-index for increasing values of k. Two lines represent the query cost on the data graph and 1-index for comparison. Figure 6(a) shows the result for 35 random queries on the IMDB, and Figure 6(a) shows for 35 random queries on the SHSR dataset from the ODP. As we can see from these charts, for values of k between 3 and 5 where the A(k)-index is substantially smaller than the 1-index, these queries can be evaluated significantly faster on the A(k)-index than on either the data graph or the 1-index.

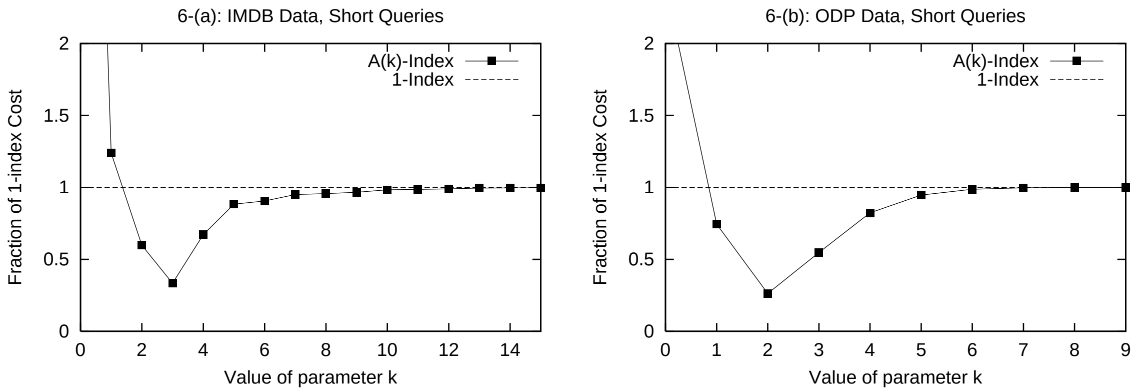


Figure 6: **Short Query Performance: IMDB+DMOZ data**

Long queries

Figure 7 shows the result of experiments which follow longer randomly-generated paths as described above. These queries are run only on the IMDB, since relatively few long paths exist in the ODP dataset, and our random walk produced very few such queries. The first graph has queries without any '_*' while the second graph shows results on queries containing an initial '_*'. (Note that, as mentioned above, almost all expressions are evaluated with a backward strategy and with the use of a label index.) The third graph is for execution costs on queries containing an '_*' in the body of the query. The final graph shows the relative cost of evaluation and validation for different values of k for the initial data set (no '_*'). In all graphs, the cost of the A(k)-index, represented as a fraction of the bisimulation index cost, is plotted against the parameter k.

We make this segregation in order to be able to clearly delineate the issues involved in processing longer queries on a highly cyclic database such as the IMDB. In such a database, most pairs of nodes have a directed path between them. As a result, a query containing '_$*$' is likely to touch most of the database (while adding dubious expressive power to the query). However, in an approximate index (since it has false paths), this leads to a large number of false nodes to be returned in the answer – as a result the validation cost is high.

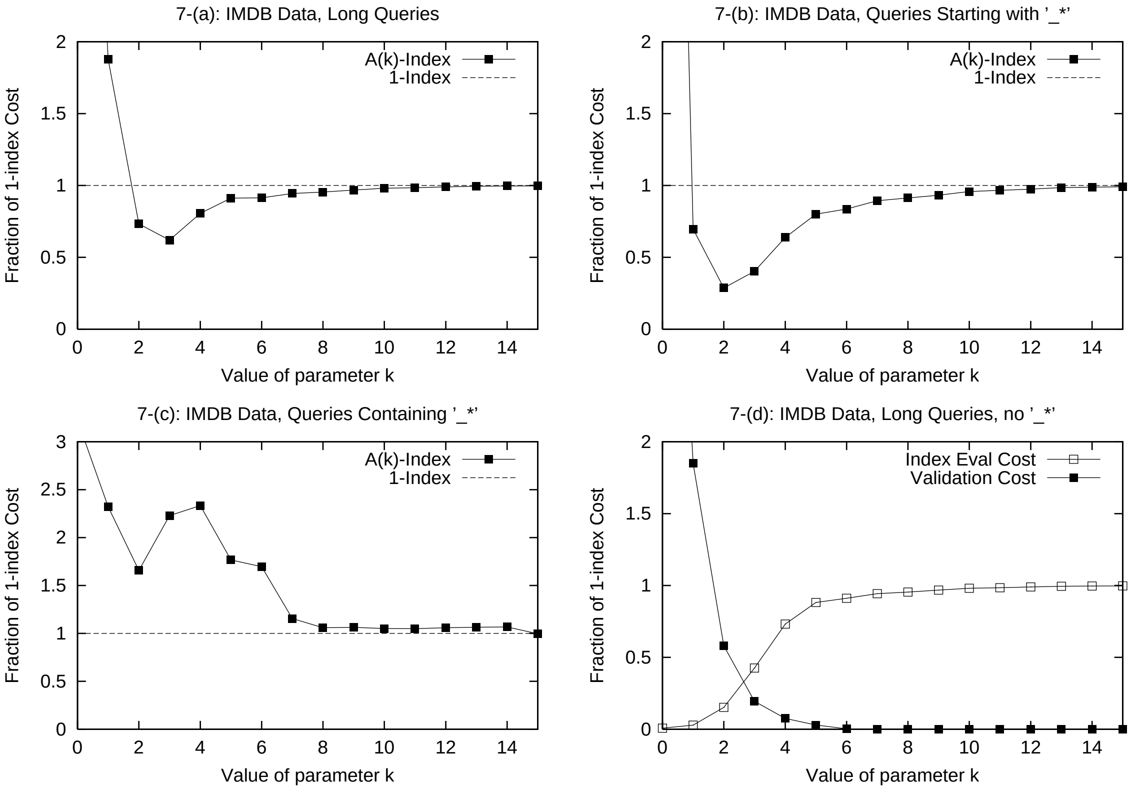


Figure 7: **Query Performance: IMDB data**

The information from this sequence of experiments is summarized below: (1) On regexp queries that have no '_*' (figure 7-(a)), the A(k)-index is significantly better, e.g., for $k = 3, 4$. The performance is upto 50% better than the 1-index while at the same time being close to half its size. (2) On queries containing an initial '_*', we do even better, where the same A(k)-indexes offer upto 75% better performance than the 1-index. (3) When queries *contain* an '_*', our performance is worse than the 1-index performance. This is due to the reasons described above.

Discussion

The overall picture we get from our performance study is that the A(k)-index, although much smaller than the 1-index, is nonetheless quite competitive in query evaluation for exact answers, and improves upon the 1-index significantly for a large class of interesting queries. In the case of the combination of data and queries being such that accept paths are short, or in the case of an approximate answer being acceptable, the A(k)-index is a clear winner. The presence of '_*' in a query however, in combination with the approximate nature of our index, can makes our execution cost higher. Thus the A(k)-index is clearly advantageous when most user path expressions are short, or when a query optimizer can select which path index to use. When there is a mix of user queries, the A(k)-index may still be the best choice, since it will win for most expressions and be competitive for most other queries.

8 Update Algorithms

Despite a significant body of work, especially on bisimulation, to our knowledge, the problem of update or incremental maintenance of the bisimulation or simulation relationship has not been addressed. However, the obvious and relatively expensive update algorithm of re-running simulation or bisimulation is not suitable for database applications which admit updates. In this section, we develop an update algorithm for the bisimulation which is significantly more efficient than the trivial algorithm. We also develop an update algorithm for the A(k)-index which can be extended to an alternative algorithm to update the 1-index. The update we handle is edge insertion and deletion, though we expect it will generalize with relative ease to larger changes. While the discussion below is focussed on edge addition, the algorithms can be extended almost directly to cover edge deletion as well. We ignore details of the XML from which the graph was extracted, such as the distinction between IDREF and tree edges and the possibility that an entire subtree might be added or deleted.

Note that a small change in a graph can trigger large changes in an index graph. For example, consider a data graph G, which is actually a tree, as shown in Figure 8 in which the sub-trees S_1 and S_2 under the two nodes labeled B are identical. The bisimulation does not group nodes in S_1 and S_2 since the two B-labeled nodes are not bisimilar, and hence neither are corresponding descendants. The 1-index is labeled as $Bisim(G)$. Suppose an edge is added between nodes 2(labeled C) and 6(labeled B) (shown with the dotted edge) and G changes to G'. Now, nodes 6 and 7 become bisimilar and so, the two sub-trees get merged in the new bisimulation based 1-index, which is labeled as $Bisim(G')$. Thus, the size of the index graph reduces by about 50%, if the two sub-trees S_1 and S_2 are large.

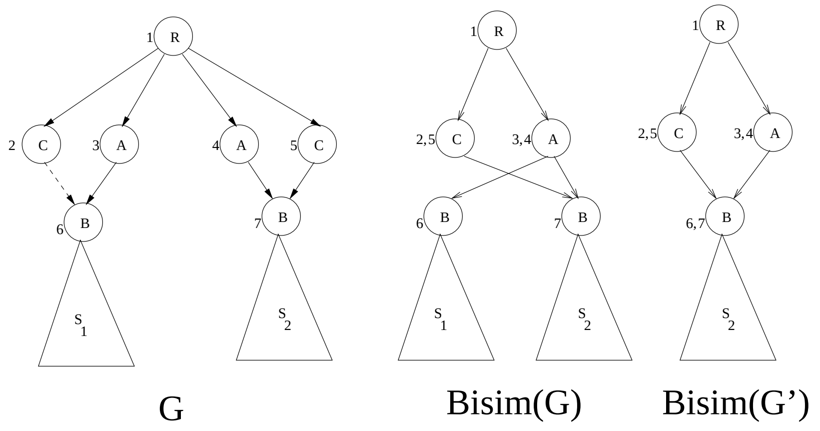


Figure 8: **Example for bad update**

As noted earlier, one advantage of k-approximate indexes is that this propagation of updates is limited to a distance of at most k from the point where the update is made. In the example above, index nodes more than k from the B-labeled nodes would have already been merged.

Our main goal is to develop algorithms in which the cost of the update is proportional to the *size* of the change in the index graph. This can be measured in terms of the number of index nodes and edges added or deleted and the number of data nodes whose extents have changed.

8.1 Propagate Algorithm for Bisimulation based 1-index

Let data graph G be updated by adding an edge from node u to node v. Let the modified data graph be G'. Let the bisimulation based 1-index on G be $Bisim(G)$ and on G' be $Bisim(G')$. The update algorithm we propose begins by taking $Bisim(G)$ and outputs some refinement of $Bisim(G')$, $Bisim^{ref}(G')$. The corresponding binary relations on the data nodes are denoted by $\approx_{G'}^{max}$ and $\approx^{ref}$. Notice that for query processing, any refinement of the bisimulation partitioning yields a precise index. Thus, $Bisim^{ref}(G')$ by itself is precise. The actual bisimulation partitioning can then be computed by using a special property of the bisimulation relation, stated in Theorem 7.

Theorem 7: *Let G' be a data graph and $\approx^{ref}$ be any refinement of $\approx_{G'}^{max}$. Then, $Bisim(Bisim^{ref}(G')) = Bisim(G')$. Here, graph equality means isomorphism.*

That is to say, to compute the bisimulation of a graph, we can take an arbitrary refinement, treat the refinement as a graph and compute the bisimulation on this refinement instead. Of course, the extents need to be "blown-up". If the index graph is much smaller than the data graph (as has proven to be for most of our experimental data), then this approach can be expected to be much less costly than re-running the bisimulation algorithm on G'.

The proof of Theorem 7 is omitted due to lack of space. However, the intuition of the proof is that two similar nodes can be "fused" and treated as a single node for the purpose of computation of the similarity partition without affecting correctness. Any refinement, $Bisim^{ref}(G)$ can be seen as resulting from a series of such "fuse" operations.

Further, if the refinement, $Bisim'(G')$, is not much larger than $Bisim(G')$, a *lazy* approach can be taken to index minimization since the refinement maintains all the properties necessary for correct functioning of the index graph. We will revisit this point later.

8.1.1 Computation of a Refinement

The key observation used by this phase of the *propagate* algorithm is that any stable partitioning of the data nodes is a refinement of the bisimulation partitioning.

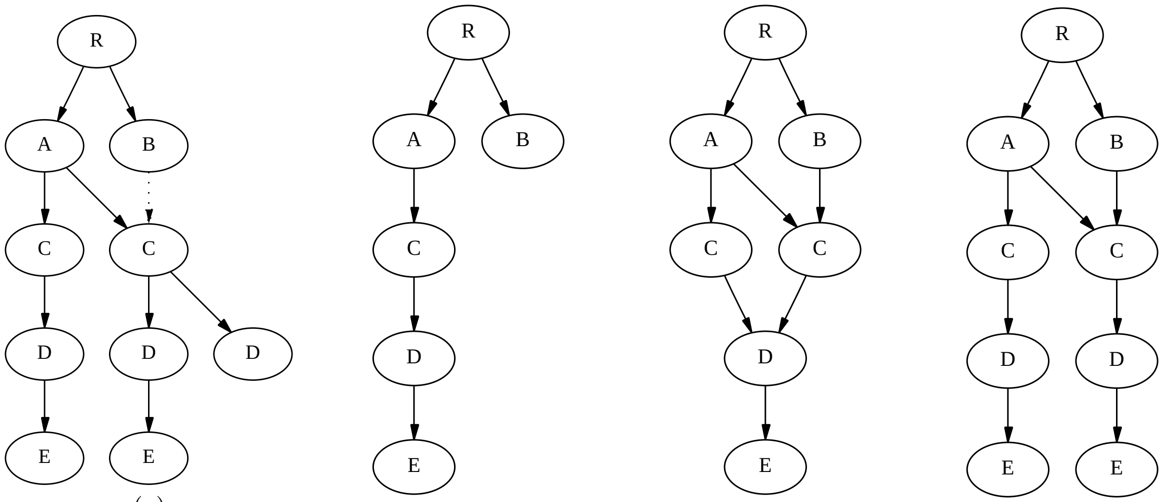
Now, the original bisimulation partitioning is stable. By adding an edge from u to v, the index node containing v in its extent, $I(v)$, may no longer be stable with respect to $I(u)$. To resolve this, *propagate* first checks if there is already an edge from $I(u)$ to $I(v)$. If so, the stability condition is ensured and *propagate* returns. If not, it pulls v out of the extent of $I(v)$ and creates a new index node I' with only v in its extent.

Now, the index nodes containing children of v may not be stable with respect to the rest of the index nodes. If this is indeed the case, the children of v are pulled out of the index nodes containing them and put into new index nodes. Here, if two children of v were earlier bisimilar, *propagate* tries to put them into the same index node if possible. It then proceeds to the grand-children of v and so on. This procedure terminates when the current partitioning is stable.

We illustrate this with an example.

Example 8 *Consider the data graph G shown in Figure 9(a). Suppose an edge is added as shown by the dotted line in the figure. Let the source node be src and the destination node be dst. The old bisimulation based 1-index is shown in Figure 9(b). For clarity, the extents are not shown. The insertion of this edge causes the index node labeled C to become unstable, since only one of the C labeled nodes in G has a parent labeled B.*

Thus, the node labeled C splits and we obtain an intermediate index graph as shown in Figure 9(c). This split means that the index node with label D is now unstable, and so needs to be split. During the split of the index node labeled D, the two children of dst can be kept together since they are still bisimilar. Propagate *groups together data nodes by extent in this manner whenever possible. The splits propagate in this manner*



(a)
Data Graph

(b)
Original 1-index

(c)
In the middle of *propagate*

(d)
Final 1-index

Figure 9: **Execution of *propagate* on an example**

till the leaf level is reached. Thus, the final index output by propagate looks as shown in Figure 9(d). In general, if the graph is strongly connected, the termination condition is stability.

Note that in this example, the final 1-index obtained after propagate *happens to be the actual 1-index of the updated data graph. This is not true in general.*

To implement this, we use a variant of the Paige-Tarjan(PT) algorithm for bisimulation computation [27], which we reviewed in Section 5. The algorithm is shown in Figure 10, where the update is the addition of an edge between data nodes u and v (in the direction u to v).

PT handles how the nodes get repartitioned. However, it does not deal with edge changes. In our algorithm, we take edges also into account. The detailed algorithm involving edges is omitted for lack of space.

The algorithm works with partitions $\mathcal{X}$ and $\mathcal{Q}$ of the data nodes, as described in section 5. However, these are initialized differently in the update algorithm, as shown in the figure. The initialization maintains the invariants needed for PT, which we described in Section 5. Thus, when we run PT with this initialization, we get a partitioning of the data nodes that is stable, which is what we desire.

procedure propagate(u,v) //Edge added from u to v
begin
1. add an edge from u to v in the data graph
2. **if** there is already an edge between $I[u]$ and $I[v]$ **then**
3.   return;
4. Initialize $\mathcal{X}$ to be the old partitioning of nodes
5. Create a new index node I'
6. Put v in the extent of I'
7. Add an edge from $I[u]$ to I'
8. Initialize $\mathcal{Q}$ to be this partitioning of nodes
9. Run PT

end

Figure 10: **Propagate**

Theorem 9: *The partition created by the procedure* propagate *is stable, and hence is a refinement of the actual bisimulation index.*

Proof: If we can show that the initialization of $\mathcal{X}$ and $\mathcal{Q}$ maintains the invariant that all $\mathcal{Q}$-partitions are stable with respect to all $\mathcal{X}$ partitions, then the claim follows by the correctness of the original algorithm. By construction, the index nodes in the index graph are mutually stable. Once the update is performed, the index node containing v may not be stable w.r.t. the index node containing u. By placing v in a partition by itself, we guarantee that the $\mathcal{Q}$ partitions are stable w.r.t. the $\mathcal{X}$ partitions. □

8.1.2 Computation of the Actual Final 1-index

To complete the update, we run a new bisimulation computation, treating the nodes of the partition computed by refine as the nodes of a data graph. Finally, these "nodes" are replaced by their extents to create an index graph based on the generated partition. Theorem 7 guarantees that the resulting partition is, in fact, the maximal bisimulation of the updated data graph.

But in our experiments, as we will see in Section 7, the size of the refined bisimulation after the propagate procedure is executed, changes by very small amounts.

Thus, in our implementation, we adopt a **lazy update scheme**, where the recomputation of the accurate bisimulation is done periodically. Note that by doing this, we do not compromise the accuracy of the bisimulation index since any refinement of the bisimulation is also a precise index. This approach also in some sense amortizes over all edge additions and deletions when the change due each individual update is small, which is usually the case as we show below.

8.1.3 Performance of Lazy Update

We now describe the performance of our lazy-update algorithm for the bisimulation. The database used is the IMDB data subset described previously, and the updates consist of a sequence of IDREF edge additions. The edges were chosen to be meaningful edges: in fact we randomly selected a subset of the already present IDREF edges in the data, and deleted them. This resulting graph was used as the initial data graph, and the deleted edges were added one at a time.

The experiment consists of (a) adding an edge to the data graph (b) using the $refine$ procedure to compute the refinement of the updated index (c) using this as the index graph, and continuing to add edges. The goal is to check the cumulative effect of lazy updates on the index size.

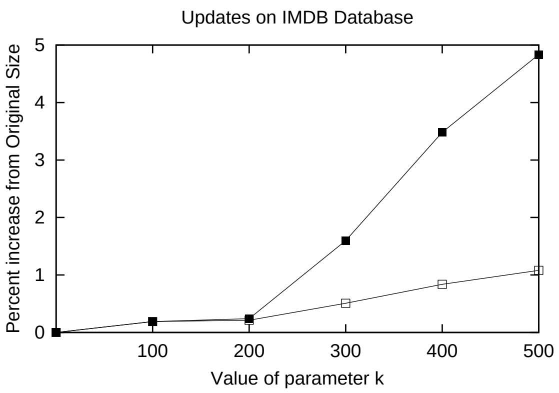


Figure 11: **Lazy Updates: IMDB data**

Figure 11 shows the results of this experiment. The x-axis is the number of IDREF edges added. The two curves are, respectively, the size of the refinement computed by repeatedly calling *propagate* to handle the IDREF additions, and the size of the accurate bisimulation index after the same IDREF edges. The y-axis is the number of nodes in the two indices, averaged over 10 different random trials. As the graph shows, even a sequence of 500 edge additions makes the refinement larger than the exact index by about 5%. Note that the exact index itself is within 10% of the database size, and further, the refinement can still be used for query processing. Hence a lazy update algorithm is eminently applicable in this scenario.

We note that the ODP data also showed a very similar behavior. The index sizes are a much larger fraction of the database in this case – however, a sequence of 1000 updates still changed the size by less than 5%.

8.2 A(k)-index Update Algorithm

The update algorithm we propose for the A(k)-index is accurate, i.e., it accurately computes the modified index. This algorithm can be extended to obtain an accurate update algorithm for the 1-index as well.

As we noted earlier, the addition of an edge between data nodes u and v can only change the k-bisimilarity classes of nodes that are distance at most $k-1$ from v. For instance, if k is 0, then there is no change in the index nodes. At most, an edge can be added from $I(u)$ to $I(v)$. Similarly, if k is 1, then the only node that needs to be relocated in the index grouping is v and if k is 2, then relocation needs to be done for v and its children.

Let us specifically consider how A(1)-index gets modified. As mentioned above, the only node that needs to be relocated is v. By the addition of an edge from u, the set of labels of the parents of v could have changed. For example, if v earlier had edges from nodes with labels l_1, l_2, l_3 and u has label l_4, then the set of labels in v's parents has changed to $\{l_1, l_2, l_3, l_4\}$. Thus, if there is an index node I' such that the set of labels in the parents of every data node in $ext(I')$ is $\{l_1, l_2, l_3, l_4\}$, then v needs to be put in $ext(I')$.

In general, to decide the relocation of a node x for A(i)-index, we need to determine the set of $i-1$-*bisimilarity equivalence classes* of the parents of x.

To implement the algorithm, we maintain an update data structure which is essentially a hierarchy of indices, from the 0-index upto the k-index. This structure indicates the *tree of splits* describing which index node split at which stage during the index construction. We call this structure the *Russian doll* update structure. However, the full index nodes are not stored:

- The index nodes are only shells, i.e., they do not contain the graph nodes that belong in their extent. They do not contain edges either. Thus, these shells are very small in size.
- The indices are linked up in a logical fashion: each index node in A(i)-index points to the unique index-node in A($i-1$)-index that was split to create this node[2] and to the A($i+1$)-index nodes into which this splits.
- Only the A(k)-index nodes have extents containing the graph nodes corresponding to the A(k)-index partitioning, and the index edges pointing to other index nodes.

We refer to these edges crossing between indices as *vertical edges*, and the edges between nodes of a single index as *horizontal edges*.

[2]Note that this is true because an i-index is a *refinement* of the $(i-1)$-index

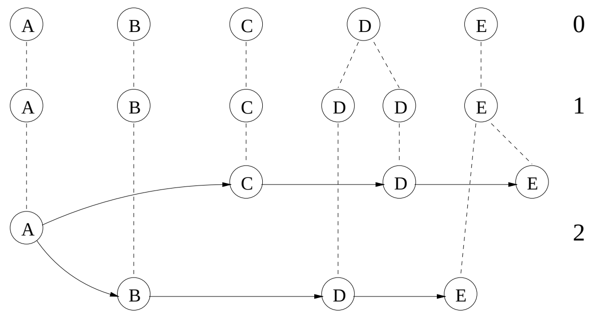


Figure 12: **Example for Russian Doll Structure**

Example 10 *The Russian doll structure for the data graph G shown in Figure 2 is shown in Figure 12 for A(2)-index. The solid lines are the horizontal edges, while the dashed ones are the vertical edges. Although the vertical edges are bi-directed, they are shown undirected for clarity.*

At level 0, there is one doll node for every label in G, corresponding to A(0)-index. At level 1, the A(0)-index node with label D splits, and this is represented in the Russian doll as shown. Note that no extents or graph edges are stored except at level 2.

The update algorithm works by first computing the correct A(1)-index grouping of nodes. It then proceeds to compute the A(2)-index partitioning, and so on for $k-1$ iterations. It also maintains a set of nodes that need to be relocated in the current iteration i, say $curr_i$. Each node in this set is relocated by first computing the set of $i-1$-bisimilarity equivalence classes of its parents. It is then relocated to an appropriate A(i)-index node I' determined by this set. This mapping from the set of $i-1$-bisimilarity equivalence classes of the parents to the A(i)-index index node is maintained in a hash table. $curr_{i+1}$ is determined by $curr_i$ union all children of $curr_i$ whose extents changed in iteration i.

Once the regrouping of nodes in A(k)-index is determined, the edges to be added and deleted are computed. The algorithm is sketched in Figure 8.2. We omit the details owing to lack of space.

```
procedure update(u,v) //Edge added from u to v
begin
add an edge from u to v in the data graph
if there is an edge from I(u) to I(v) then
   return
curr-list = {v}
for  i = 1 to k − 1 do
   foreach  w in curr-list do
      Parents_w = set of A((i − 1))-index nodes containing the graph parents of w
      Dest_w = HashTableLookup(i − 1, Parents_w)
      if  Dest_w == NULL do
         create New_{i,w}, the new i-index node for w
         update hash table and Russian doll appropriately
         set Dest_w to New_{i,w}
      relocate w to Dest_w
      mark that the extent of w has changed
   curr-list = curr-list ∪ children of nodes whose extents changed
update edges appropriately
end
```

Figure 13: **k-index update algorithm**

It is possible for large k that the space taken up by the Russian doll structure is high. But, we can compress it based on a simple observation: if an index node does not split during a particular iteration, we do not have to create a new node for its (vertical) child. It turns out that the benefit yielded by this compression is high. But, maintaining a compressed Russian doll with updates is hard. Here, we can be lazy and periodically compress the Russian doll.

Since the 1-index is nothing but the A(k)-index for sufficiently large k, this algorithm can be extended to accurately update the 1-index. However, for large ks, the cost of the doll update algorithm can be high. This is typically the case for the full bisimulation based 1-index. Thus, the *propagate* algorithm can be cheaper for the full 1-index.

9 Conclusion

The A(k)-index is a clean generalization of the 1-index. By varying k, this family of indices offers a smooth tradeoff between the index size and accuracy. Owing to their smaller size, and for appropriate values of k, they perform much better than the 1-index for interesting classes of queries as indicated by our performance study, while remaining competitive for most other queries. They also have the advantage of worst case guarantees on the impact of updates. We expect that their use will extend beyond indexing to areas such as schema extraction and query optimization—for instance, in maintaining statistics. We also expect our techniques to generalize to handle more complex path conditions such as selection and branching. This is part of our future work.

Any practical indexing scheme must be updatable, and we also handle this aspect of the path indexing problem for schema-less XML and semi-structured data. The update algorithm we propose for the 1-index, although lazy, works well for the test cases we ran it on. Since this algorithm can equally well be used to update the bisimulation partition, we expect our update algorithms will have applications outside the field of databases. We also propose update algorithms for the A(k)-index based on the "Russian doll" structure. We expect that this Russian doll structure has many applications beyond update such as in query processing, since it can potentially maintain a combined index equivalent to several A(k)-index, allowing different portions of the index to be used as appropriate for a particular query. Finally, we would like to make our index structures more *adaptable* to specific query workloads, so that complexity is not maintained in the index structure which is never utilized in its application.

References

[1] S. (Serge) Abiteboul, Peter Buneman, and Dan Suciu. *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[2] Serge Abiteboul. Querying semi-structured data. In *ICDT*, volume 6, pages 1–18, 1997.

[3] P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, May 2000. Electronic edition.

[4] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186 of *Lecture Notes in Computer Science*, pages 336–350, Delphi, Greece, 8–10 January 1997. Springer.

[5] D. Bustan and O. Grumberg. Simulation based minimization. In *Proceedings 17th ICAD*, Pittsburgh, PA, jun 2000.

[6] J. Clark. Extensible Stylesheet Language (XSL), Version 1.0. World Wide Web Consortium Recommendation. `http://www.w3.org/TR/xslt`, Nov 16, 1999.

[7] J. Clark and S. DeRose. Xml path language (xpath) 1.0. w3c recommendation. World Wide Web Consortium, www.w3.org, nov 1999.

[8] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion. In *SIGMOD'98*, 1998.

[9] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proceedings of the Eighth World-Wide Web Conference*, 1999.

[10] M. Fernandez, J. Sim, e Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom. XML query languages: Experiences and exemplars. `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`, 1999.

[11] Mary Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. Declarative specification of Web sites with Strudel. *VLDB Journal: Very Large Data Bases*, 9(1):38–55, May 2000. Electronic edition.

[12] K. Fisler and M. Vardi. Bisimulation and model checking. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Sep 1999.

[13] A. Gionis, M. Garofalakis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *Proceedings of ACM-SIGMOD 2000 International Conference on Management of Data*, 2000.

[14] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Workshop on the Web and Databases (WebDB '99)*, pages 25–30, 1999.

[15] 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.

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

[17] David Lee and Mihalis 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.

[18] The Internet Movie Database Ltd. Internet movie database. `ftp://www.imdb.com`.

[19] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases*, September 1999.

[20] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Dataguides: Enabling query formulation and optimization in semistructured databases. Technical report, Stanford University, January 1998.

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

[22] Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, London, UK, September 1971. William Kaufmann.

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

[24] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *SIGMOD98*, pages 295–306, 1998.

[25] 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, Washington - Brussels - Tokyo, April 1997. IEEE.

[26] F. Neven and T. Schwentick. Expresive and efficient pattern languages for tree-structured data. In *Proceedings of the Nineteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 145–155, Dallas, Texas, 2000.

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

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

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

[30] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases (DOOD '95)*, number 1013 in LNCS, pages 319–344. Springer, 1995.

[31] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). WWW The Query Language Workshop (http://www.w3.org/TandS/QL/QL98/pp/xql.html), 1998.

[32] Markus Roggenbach and Mila Majster-Cederbaum. Towards a unified view of bisimulation: a comparative study. *Theoretical Computer Science*, 238(1–2):81–130, May 2000.

[33] C. Stirling. The joys of bisimulation. *Lecture Notes in Computer Science*, 1450:142–157, 1998.

[34] W3C. Document object model(DOM). `http://www.w3.org/DOM/`.

A Proofs

This Appendix is not to be considered part of the paper, and contains certain proofs *strictly* for the convenience of the interested reviewer.

Theorem 11: *Let p be a label $l_1, l_2, \ldots, l_j, j \leq k$, P be a regular path expression, and N a node in an $A(k)$-index built on graph G. If all paths $p'.p$ that exist in G are in $L(P)$ then all data-graph nodes in $ext[N]$ are in the target set of P.*

Proof: Assume node N and path p which is a suffix of p', an incoming path to N. Let $o \in ext[N]$ and let o' be a node in the extent of the index node labeled l_1 on this path which is connected by a path p to n. Since the length of p is $\leq k$, we know such a node and element o' exist. Since o' is connected to the root, there is a path in the data graph from the root to o the suffix of which matches p, and by the claim, this path is in $L(P)$. □

Theorem 12: *Let G be a data graph and $\approx^{ref}$ be any refinement of $\approx_G^{max}$. Then, $Bisim(Bisim^{ref}(G)) = Bisim(G)$. Here, graph equality means isomorphism.*

Proof: The graph $Bisim(Bisim^{ref}(G))$ induces a partitioning of the nodes of G if we "blow up" the extents to the graph nodes. It is easy to see that the index graph obtained by this partitioning is the same as $Bisim(Bisim^{ref}(G))$. Thus, all we have to show is that this partitioning is the same as the $Bisim(G)$ partitioning.

For this purpose, we make the following claim. *Claim:* Let u and v be two nodes in G. Let $Bisim^{ref}(u)$ and $Bisim^{ref}(v)$ be the two nodes in $Bisim^{ref}(G)$ that have u and v respectively in their extents. Then,

$$u \approx_G^{max} v \text{ iff } Bisim^{ref}(u) \approx_{Bisim^{ref}(G)}^{max} Bisim^{ref}(v)$$

$\Rightarrow$: Let u and v be bisimilar in G. We show that $Bisim^{ref}(u)$ and $Bisim^{ref}(v)$ are bisimilar in $Bisim^{ref}(G)$ by producing a binary relation $\approx'_{ref}$ on the nodes of $Bisim^{ref}(G)$ that is a bisimulation. *equiv'* is defined as follows:

$$a \approx_G^{max} b \Rightarrow Bisim^{ref}(a) \approx'_{ref} Bisim^{ref}(b)$$

Note that since $Bisim^{ref}(G)$ is defined from a refinement of the actual bisimulation, if $Bisim^{ref}(a) \approx'_{ref} Bisim^{ref}(b)$, then each node in the extent of $Bisim^{ref}(a)$ is bisimilar to every node in the extent of $Bisim^{ref}(b)$.

To show that *equiv'* is a bisimulation, consider some parent $Bisim^{ref}(u')$ of $Bisim^{ref}(u)$, where u' is a parent of some u_1 in the extent of $Bisim^{ref}(u)$. Now, u_1 is bisimilar to v and so, there is some parent v' of v such that u' is bisimilar to v'. Thus, $Bisim^{ref}(v')$ is a parent of $Bisim^{ref}(v)$ and is $\approx'_{ref}$ to $Bisim^{ref}(u')$. Now consider any parent $Bisim^{ref}(v'')$ of $Bisim^{ref}(v)$. This part is similar to the above and can be shown in the same way.

$\Leftarrow$: We proceed as in the previous case by defining a binary relation $\approx'$ on the nodes of G that we claim to be a bisimulation.

$$a \approx' b \text{ iff } Bisim^{ref}(a) \approx^{max}_{Bisim^{ref}(G)} Bisim^{ref}(b)$$

Consider some parent u' of u. Hence, $Bisim^{ref}(u')$ is a parent of $Bisim^{ref}(u)$. Since $Bisim^{ref}(u)$ is bisimilar to $Bisim^{ref}(v)$, there is some parent $Bisim^{ref}(v'')$ of $Bisim^{ref}(v)$ (where v'' is a parent of some v_1 in the extent of $Bisim^{ref}(v)$) that is bisimilar to $Bisim^{ref}(u')$. Now, since v and v_1 are bisimilar, there is some parent v' of v that is bisimilar to v''. Hence, by the first part of the proof, $Bisim^{ref}(v'')$ is bisimilar to $Bisim^{ref}(v')$. Hence, since bisimilarity is a transitive relation, $Bisim^{ref}(u')$ is bisimilar to $Bisim^{ref}(v')$. Hence, $u' \approx' v'$. Similarly, for any parent v''' of v, we can produce some parent of u that is related to it by $\approx'$. Thus, $\approx'$ is a bisimulation.$\Box$

This Appendix is not to be considered part of the paper, and contains certain proofs *strictly* for the convenience of the interested reviewer.