Index Structures for Path Expressions

Extended Abstract

Tova Milo
Tel Aviv University
milo@math.tau.ac.il

Dan Suciu
AT&T Labs
suciu@research.att.com

1 Introduction

In recent years there has been an increased interest in managing data which does not conform to traditional data models, like the relational or object oriented model. The reasons for this non-conformance are diverse. One one hand, data may not conform to such models at the physical level: it may be stored in data exchange formats, fetched from the Internet, or stored as structured files. One the other hand, it may not conform at the logical level: data may have missing attributes, some attributes may be of different types in different data items, there may be heterogeneous collections, or the data may be simply specified by a schema which is too complex or changes too often to be described easily as a traditional schema. The term *semistructured* data has been used to refer to such data. The data model proposed for this kind of data consists of an edge-labeled graph, in which nodes correspond to objects and edges to attributes or values. Figure 1 illustrates a semistructured database providing information about a city.

Relational databases are traditionally queried with associative queries, retrieving tuples based on the value of some attributes. To answer such queries efficiently, database management systems support indexes for translating attribute values into tuple ids (e.g. B-trees or hash tables). In object-oriented databases, *path queries* replace the simpler associative queries. Several data structures have been proposed for answering path queries efficiently: e.g., access support relations [14] and path indexes [4].

In the case of semistructured data, queries are even more complex, because they may contain *generalized* path expressions [1, 7, 8, 16]. The additional flexibility is needed in order to traverse data whose structure is irregular, or partially unknown to the user. For example the following query retrieves all restaurants serving *lasagna* for dinner:

select x
from $(*.Restaurant)\ x\ (Menu.*.Dinner.*.Lasagna)\ y$

Starting at the root of the database DB, the query searches for paths satisfying the regular expression $*.Restaurant$ and, from the retrieved nodes, searches for another regular expression, $Menu.*.Dinner.*.Lasagna$.

How are such queries evaluated ? A naive evaluation that scans the whole database traversing all possible paths and selects those that match the patterns in the query is obviously very expensive. As in the case of relational and OO databases, we would like to use some indexes to speed up the evaluation of such queries. Index structures developed for traditional data models rely on some pre-defined database schema: e.g. relational databases index on a specific attribute of a specific relation, while object-oriented databases index on a specific path [4, 14] in the object-oriented schema (e.g. *document.section.title*). Hence, these index structures are not applicable to semistructured data, because the schema is missing, unavailable, or only partially known. At the other extreme, full text indexing systems take an opposite approach. Given no knowledge on the structure of information, they index *all* the data. But this is still of limited use for semistructured data, where some (perhaps very partial) knowledge on the structure may be available and exploited in queries: e.g. the query above insists that a *Dinner* item appears inside a a *Menu*.

Recent work has addressed the problem of efficiently evaluating path expressions on semistructured databases [2, 19, 18, 11]. But they focused mainly on deriving and using schema information to rewrite queries and guide the search. The issue of indexing was almost ignored. An exception are the *dataguides* of [11] which record information on the existing paths in a database, using this as an index. However, the scope of dataguides is restricted to queries with a single regular expression: they are not adequate for more complex queries, having several regular expressions and variables, like the one above.

In this paper we propose a novel, general index structure for semistructured databases, called T-index. It improves over the previous approaches in several ways. First, T-indexes are flexible in that they allow us to trade space for generality. The class of paths associated with a given T-index is specified by a *path template*. For example, we can build a T-index to evaluate paths described by the template $\boxed{P}\ x\ \boxed{P}\ y$: here $\boxed{P}$ can be replaced by any regular expression (P stands for "path expression"). The query above is of this form. An alternative template would be ($*.Restaurant$) $x\ \boxed{P}\ y$, in which the first regular expression is fixed to $*.Retaurant$: the corresponding T-index takes less space while being less general. Second, we show that every T-index can be efficiently constructed. Dataguides [11] required a powerset construct over the underlying database, which in the worst case can be of exponential cost: by contrast, T-indexes rely on the computation of a simulation or a bisimulation relation, for which efficient algorithms exists. Third, we offer guarantees for the size of a T-index. For example the size of a T-index associated to a single regular expressions is at most linear in that of the database, (again, we contrast this to dataguides which, in the worst case, are exponential), and often, as our experiments show, it is much less. Third, we show that T-indexes turn out to be elegant generalizations of index structures considered previously in various contexts: dataguides for semistructured data, Pat trees for full text indexes [12, 21], and Access Support Relations for OODBs [14].

A T-index starts by grouping database objects into equivalence classes containing objects that are indistinguishable w.r.t to a class of paths defined by a path template as described above. Computing this equivalence relation may be expensive (PSPACE complete), so we consider finer equivalence classes defined by bisimulation or simulation, which are efficiently computable. Next, a T-index is built from these equivalence classes, by constructing a non-deterministic automaton whose states represent the equivalence classes and whose transitions correspond to edges between objects in those classes.

While each T-index is designed for a particular class of queries (given by one template), it can be *used* to answer queries of more general forms. We address the problem of deciding whether a given query with generalized path expressions can be rewritten to take advantage of a given T-index. In its full generality, this problem is a generalization of the query rewriting problem [15] to the case of queries with generalized path expressions and, to the best of our knowledge, is still open. Here we have a more modest goal: we show that a certain restriction of this query rewriting problem is decidable, and, moreover, it is in PTIME for a specific class of queries, which is of interest in practice. Even in this restricted form, our result has an interesting Corollary: the fact that containment of regular expressions consisting of concatenations of constants and wildcards is decidable in PTIME. This comes at a surprise, because the associated deterministic automaton in this case is still exponential in the size of the regular expression.

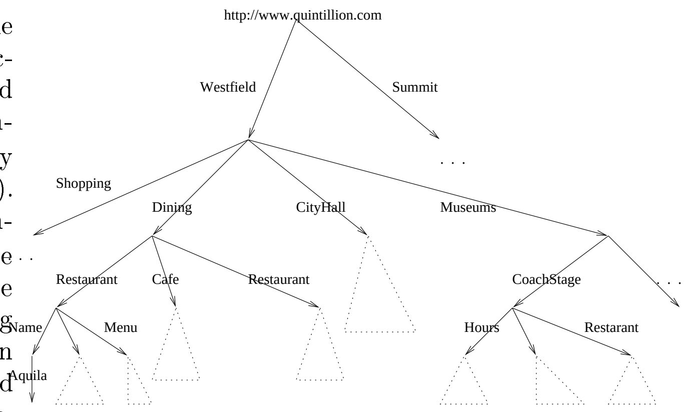

Figure 1: Example of a semistructured database with informations on small towns in New Jersey.

Organization: In section 2 we review the data model and query language for semi-structured data and introduce the notion of *path templates*. To explain how T-indexes are built for such templates, we first consider in Sections 3 and 4 two specific templates and their corresponding indexes, called 1 and 2-index resp. While presenting these two cases we illustrate the details of our techniques, then we carry them over in Section 5 to general case of T-indexes. We conclude in section 6.

2 Review: Data Model and Query Languages

We start by reviewing the basic framework on databases and queries.

The data model: All models proposed for semistructured data consists of a labeled graph, in which nodes correspond to the objects in the database, and edges to their attributes. Unlike the relational or object oriented data models, the labeled graph model carries both data and schema information, making it easy to represent irregular data and treat data coming from different sources in a uniform manner[2].

Definition 2.1 *We assume an infinite set $\mathcal{D}$ of* data values *and an infinite set $\mathcal{N}$ of* Nodes. *A* data graph *$DB = (V, E, R)$ is a labeled rooted graph, where $V \subset \mathcal{N}$ is a finite set of nodes; $E \subseteq V \times \mathcal{D} \times V$ is a set of labeled edges, and $R \subseteq V$ is a set of root* *nodes. W.l.o.g. we assume that all the nodes in V are reachable from some root in R. We will often refer to such a data graph as a* database.

Path expressions: Following [16, 6], we use formulas to describe properties of the labels of the edges of data graphs. We assume a set of base predicates $p_1, p_2 ...$, over the domain of values $\mathcal{D}$, and denote with $\mathcal{F}$ the set of formulas obtained by taking boolean combinations of such predicates. We assume that satisfiability of formulas in $\mathcal{F}$ is decidable.

A *regular path expression*, or *path expression* in short, P, is a regular expression over formulas in $\mathcal{F}$. That is, $P ::= \epsilon \mid f \mid (P|P) \mid (P.P) \mid P*$. We denote with $L(P)$ the regular language defined by P, and with $W(P)$ the set of all words $w = a_1 \dots a_n$ over values in $\mathcal{D}$, s.t. there exists a word $w' = f_1 \dots f_n \in L(P)$ and $f_i(a_i)$ holds for all $i = 1 \dots n$ (i.e. the set of word obtained by replacing each formula by some value that satisfies it). Using the traditional techniques for regular language it is easy to see that the languages defined by path expressions are closed under intersection and that the emptiness problem for $W(P)$ is decidable.

Given a data graph DB and a path $p = v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} v_2 \dots v_{n-1} \xrightarrow{a_n} v_n$ in DB, we say that p *matches* the path expression P iff the word $a_1 \dots a_n$ is in $W(P)$.

For brevity, we will use in the sequel the following shorthands. The path expression $\lambda x.(x = d)$, where d is a constant, is written as d; $\lambda x.True$ is written as $\_$; and $\_*$ is written as $*$. For example $*.Restaurant.*.Name.Fridays$ is a regular path expression.

Queries A *query path* is an expression of the form $P_1\ x_1\ P_2\ x_2 \dots P_n\ x_n$ where the x_i's are distinct variable names, and the P_i's are path expressions. Given a graph database $DB = (V, E, R)$, we say that the nodes $v_0, v_1, \dots, v_n$ *satisfy* a query path $P_1\ x_1\ P_2\ x_2 \dots P_n\ x_n$ if $v_0 \in R$ (is a root) and for all $v_{i-1}, v_i, i = 1 \dots n$, there exist a path from v_{i-1} to v_i that matches P_i. A *query* has the form:

select $x_{i_1}, x_{i_2}, \dots, x_{i_k}$
from $P_1\ x_1\ P_2\ x_2 \dots P_n\ x_n$

where $1 \le i_1 < i_2 < \dots < i_k \le n$. That is, a query consists of a query path and a set of head variables. The query in Section 1 has this form. We will often refer to the query by giving only the query path, and implicitly assume all its variables to be head variables. The *answer* of a query is the projection on the indexes $i_1, \dots, i_k$ of all tuples $(v_0, v_1, \dots, v_n)$ that satisfy the query path.

Path Templates: Relational databases create a separate index for each relation, attribute pair. Object oriented databases associate separate path indexes for each path in the object-oriented schema. Hence, an index can answer only a certain class of queries, for which it was designed. The index structures we describe here are also designed for given classes of queries. Such a class is specified by a *query template*. Formally, a *query template* t has the form $T_1\ x_1\ T_2\ x_2 \dots T_n\ x_n$ where each T_i is either a regular path expression, or one of the following two place-holders: $\boxed{P}$ and $\boxed{F}$. A concrete query path q is obtained from a query template t by instantiating each of the $\boxed{P}$ place holders by some concrete path expressions, and each of the $\boxed{F}$ place holders by some concrete formulas. The query path thus obtained is called an *instantiation* of the query template t. The set of all such instantiations is denoted $inst(t)$.

For example, consider the query template $(*.Restaurant)\ x_1\ \boxed{P}\ x_2\ Name\ x_3\ \boxed{F}\ x_4$. The following three query paths are possible instantiations:

$$q_1 = (*.Restaurant)\ x_1 * x_2\ Name\ x_3 Fridays\ x_4$$
$$q_2 = (*.Restaurant)\ x_1 * x_2\ name\ x_3\ \_\ x_4$$
$$q_3 = (*.Restaurant)\ x_1\ (\epsilon \mid \_)\ x_2\ Name\ x_3 Fridays\ x_4$$

Given a query template t, our goal is to construct an index structure that will enable an efficient evaluation of queries in $inst(t)$. (In fact, as we shall see later, it will also assist in answering several variants of such queries). The templates are used to guide the indexing mechanism to concentrate on the more interesting (or frequently queried) parts of the data graph. For example, if we know that the database contains a restaurants directory and that most common queries refer to the restaurant and its name, we may use a query template such as the one above to guide the indexing process. As another example, assume we know nothings about the database, but assume that users never ask for more than k objects on a path. Then we may take $t = \boxed{P}.x_1.\boxed{P}.x_2 \dots \boxed{P}x_k$, and build the corresponding index.

Before explaining how indexes are constructed for general templates, we give some intuition about the indexing process using two concrete templates $t_1 = \boxed{P}\ x_1$ and $t_2 = *\ x_1 \boxed{P}\ x_2$. The first is targeted to queries searching for nodes reachable from the root by some arbitrary path expression, (i.e. queries of the form select x from $P\ x$, where P is any path expression). The second is targeted for queries searching for pairs of objects connected by some path matching an arbitrary path expression. (i.e. queries of the form select x, y from $*\ x\ P\ y$). We call the index constructed to handle the first case a *1-index* and the one for the second case a *2-index*. While presenting these two cases we will illustrate the details of our techniques. Then we will carry them over to the general case, called *T-index* (Template index). We do not address the issue of index maintenance here and consider it only briefly in section 6.

3 1-Indexes

The 1-index assists, given some path expression P, in finding all objects reachable from the root by a matching path. Putting this in terms of query templates, it assists in computing query paths $q \in inst(\boxed{P}\ x)$. The index consists of a concies description of all possible paths in DB and, for each such path, of the objects reachable by the path. Queries can then be evaluated over this compact representation, rather than on the original database.

A First attempt: A naive way (which we will soon refine) to capture information about the paths in a data graph DB is to proceed as follows. For each node v in DB, let $L_v(DB)$, or L_v in short, when DB is understood, be the set of words on paths from some root node to v:

$$L_v(DB) \overset{\text{def}}{=} \{w \mid \begin{array}{l} w = a_1 \ldots a_n \text{and there exists a} \\ \text{path } v_0 \overset{a_1}{\to} \ldots \overset{a_n}{\to} v \\ \text{in } DB \text{ with } v_0 \text{ being a root node}\} \end{array}$$

Next, define the *language equivalence relation*, $v \equiv u$ on nodes in DB to be:

$$v \equiv u \Longleftrightarrow L_v = L_u$$

We denote with $[v]$ the equivalence class of v in DB. Clearly, there are no more equivalence classes than nodes in DB. The language equivalence is important because two nodes v, u in DB can be distinguished[1] by a query path in $inst(\boxed{P}\ x)$ iff $u \not\equiv v$.

A naive index can be constructed as follows: it consists of the collection of all equivalence classes $s_1, s_2, \ldots$, each accompanied by (1) an automaton/regular expression describing the corresponding language, and (2) the set of nodes in the equivalence class. We call this set the *extent* of s_i, and denote it by $\mathsf{extent}(s_i)$. Given the naive index, a query path of the form $P\ x$ can be can be evaluated by iterating over all the classes s_i, and for each class testing if the language of that class has a nonempty intersection with $W(P)$. The answer of the query is the union of all the extents $\mathsf{extent}(s_i)$ for which this intersection is not empty.

This naive approach is inefficient, for two reasons.

[1]By *distinguished* we mean that one node belongs to the query's answer while the other does not.

- Construction Cost: the construction of the index is very expensive since computing the equivalence classes for a given data graph is a $PSPACE$ complete problem [22].
- Index Size: the automaton/regular expressions associated with different equivalence classes have overlapping parts which are stored redundantly. This also results in inefficient query evaluation, since we have to intersect $W(P)$ with each regular language.

We next address these problems. To tackle the construction cost we introduce the notion of approximation. We call an equivalence relation $\approx$ an *approximation* if it has the property:

$$v \approx u \Longrightarrow v \equiv u \qquad (1)$$

As we shall see, any approximation is fine for constructing 1-indexes, as soon as it is efficiently computable: we illustrate below two examples of approximations. The basic idea to tackle the index size was introduced in [19], and consists in a more concise representation for the languages of $s_1, s_2, \ldots$, based on finite state automata. A novelty here over [19] is the use of a *non deterministic* automaton to get a more compact structure.

Approximations We discuss here two choices for approximations $\approx$ of $\equiv$: *bisimulation*, $\approx_b$, and *simulation*, $\approx_s$. Both are discussed extensively in the literature [17, 20, 13]. The idea that these can be used to approximate the language equivalence dates back to the modeling of reactive systems and process algebras [13]. For completeness, we revise their definitions in the Appendix.

Both $\approx_b$ and $\approx_s$ are approximations, i.e. satisfy Equation 1. In fact we have: $v \approx_b u \Longrightarrow v \approx_s u \Longrightarrow v \equiv u$. The implications are strict: this is illustrated in Figure 4, where $x \equiv y \equiv z$, $x \not\approx_s y \approx_s z$, and $x \not\approx_b y \not\approx_b z$. Moreover, $\approx_b$ is easiest to compute ($O(m \log n)$), followed by $\approx_s$ ($O(mn)$), then by $\equiv$ (PSPACE).

In constructing our indexes we will use either a bisimulation or a simulation. The reader may wonder how much we loose in practice by using an approximation instead of $\equiv$. The answer is: not much. In fact, for tree data graphs the three coincide. We prove a slightly more general statement. Let us say that a database DB has *unique incoming labels* if for any node x, whenever a, b are labels of two distinct edges entering x, then $a \neq b$. In particular, tree databases have unique incoming labels. We prove in the Appendix:

Proposition 3.1 *If DB is a graph database with unique incoming labels, then $\equiv$, $\approx_s$, and $\approx_b$ coincide.*

1-Indexes We can now define 1-indexes. Given a database DB and an approximation equivalence relation $\approx$ (i.e. satisfying equation (1)), we construct a rooted labeled graph $I(DB)$ as follows. Its nodes will be the equivalence classes $s_1, s_2, \ldots$, i.e. each s_i is some equivalence class (w.r.t $\approx$) $[v]$, for some node v in DB. $I(DB)$ has an edge $s_i \xrightarrow{a} s_j$ iff DB contains an edge $v \xrightarrow{a} v'$ for some $v \in s_i, v' \in s_j$. Finally, the roots are the equivalence classes of DB's roots, i.e. all the $[v]$ where v is a root of DB. Thus, the regular languages which previously had to be stored explicitly for each equivalence class s_i are now implicitly given as $L_{s_i}(I(DB))$.

We call $I(DB)$ the *1-index* of DB, and when DB is clear from the context we omit it and simply use I. We store an 1-index as follows. First we associate an oid s to each node in I, and store I's graph structure in a standard fashion. Second, we record for each node s the nodes in DB belonging to that equivalence class, which we denote $\mathsf{extent}(s)$. That is, if s is an oid for $[v]$, then $\mathsf{extent}(s) = [v]$. The space for I incurs two costs: the space for the graph I, and that for the extents. The graph is at most as large as the data graph DB, but we will argue that in practice it may be much less. The extents are exactly the total number of nodes in DB: this may be acceptable for 1-indexes, but will become too costly for more complex indexes, discussed later. We describe in the Appendix techniques for reducing the total size of all extents.

Evaluating Query Paths with 1-Indexes We describe now how to evaluate a query path $P\ x$. Rather than evaluating it on the data graph DB we evaluate it on the index graph $I(DB)$. Let $\{s_1, s_2, \ldots, s_k\}$ be the set of nodes in $I(DB)$ that satisfy the query path. Then the answer of the query on DB is $\mathsf{extent}(s_1) \cup \mathsf{extent}(s_2) \cup \ldots \cup \mathsf{extent}(s_k)$. The correctness of this algorithms follows from the following proposition, whose proof is in the Appendix:

Proposition 3.2 *Let $\approx$ be an approximation (i.e. satisfies Equation (1)) on DB. Then, for any node v in DB, $L_v(DB) = L_{[v]}(I(DB))$.*

The complexity of evaluating a query $q = P\ x$ on any graph is proportional to the size of the graph. In fact it is polynomial in the size of the graph, the query path, and the complexity of computing the truth value of unary formulas in $\mathcal{F}$. Since the index is likely to be smaller than the database DB, evaluating the query on the index rather than on the database yields better performance. Note that nodes in the index graph may have many outgoing edges. This is because an equivalence class may contain many nodes, and the outgoing edges of the class node is the union

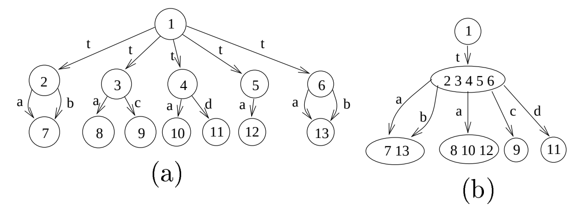

Figure 2: A data graph (a) and its 1-index (b)

of all their outgoing edges. To make the computation faster, these edges can be further indexed (e.g. by hashing or using B-tree on the labels) so that the selection of edges with specific labels is faster.

Example 3.3 Figure 2 (a) illustrates a fragment of a database with tuples of a irregular structure (we dropped the values of the attributes). Its 1-index is shown in Figure 2 (b). When evaluating a query $q = t.a\ x$ we follow the two $t.a$ paths (rather than the 5 in the original database), and take the union of their extents: $\{7, 13\} \cup \{8, 10, 12\}$. If attribute values are added, then the current leaves will have outgoing edges representing them. Typically there are many possible values (hence outgoing edges) for an attribute, so we will index these outgoing edges (e.g. using B-tree). When searching for a specific value, e.g. $t.a.7$, we will follow the two $t.a$ paths, then in each of them use the corresponding B-tree to identify the outgoing 7 edge.

The Size of a 1-Index The storage of a 1-index consists of the graph I and the sum of all extents. As explained above, both of them are bounded in size by the size of the database (up to a constant factor). Since query paths are now computed on the index graph I rather than on DB, the smaller I is, relative to DB, the better the improvement in performace. On the experimental side we tested the technique on a variety of databases, obtaining very encouranging results, showing that in common scenarios I is significantly smaller than DB. A brief discussion of the experiments is given in the Appendix, where we also describe three simple implementation techniques to further reduce the the storage size for both the graph I and the associated extents. On the theoretical side we identify here two parameters which may cause the size of I to approach its upper bound. These are: (1) a large number of distinct labels in DB, and (2) the existence of very long acyclic paths. We prove here that, by imposing limits on these parameters, the upper bound on the size of I is independent on that of DB. Technically this is one of the hardest results in this paper, and we believe it is valuable in focusing future research aimed at reducing the index size.

Formally, for a database DB and number k, we say that DB is "k-short" if there are no simple[2] paths of length $> k$. For example trees of depth $\leq k$ are k-short. Some important instances of semistructured databases are in practice k-short, for some small k. Namely many web sites have the following structure: they start as a tree of depth d, then add *back links*, which always point back to some ancestor of the current page, and a *navigation bar*, consisting of p links to p distinguished pages in the web site: importantly, every page having a navigation bar refers to the same set of p distinguished pages. It is easy to see that such a database is $d + p(d-1)$ short. In practice, both d and p are very small, even if the web site itself is large.

Theorem 3.4 *Let DB be a k-short database having at most p distinct labels, and let $\approx$ be any approximation which is at least as coarse as a bisimulation[3]. Then the size of I is bounded by some number depending only on k and p, and is independent on the size of DB.*

The proof is sketched in the Appendix.

Connection to Related Work: Data Guides In [19] and [11], the authors proposed for the first time a method for extracting all the possible path information from a given database DB, and describe it as a concise labeled graph called a *dataguide*. In their approach they insist that each path in the data be represented at most once in the dataguide: this implies that the dataguide, when viewed as a finite state automata, is *deterministic*. In fact, a dataguide G for DB is any deterministic automaton which generates the same words as DB. Here, and in the following discussion, both DB and G are viewed as automata by taking their roots as initial states and all their nodes as final states. However, [11] observes that not any dataguide is appropriate for answering queries, because in general there exists no clear correspondence between states in G and sets of nodes in DB (our extents). They therefore consider only dataguides having certain properties, which they call *strong* data guides. For any DB there exists exactly one strong dataguide G, namely the standard powerset automaton construct on DB. The correspondence between nodes in G and nodes in DB is now explicit, since each node in G *is* a set of nodes in DB: this relationship is similar to our extents. However, unlike in our 1-indexes, the extents of a strong dataguide may overlap. Hence, the storage size for dataguides is larger than that for 1-indexes for two reasons: (1) the size of the dataguide graph may be as large as exponential in that of the database, while the 1-index is at most linear, and (2) the total size of all extents in a dataguide may be as large as exponential in that of the database, due to overlaps, while for 1-indexes it is again linear in the size of DB. We believe that one of the main contributions of our work is to identify that, by relaxing the determinism requirement imposed on dataguides, the 1-indexes can be constructed and stored more efficiently, while at the same time achieve a similar performance. We pinpoint the relationship between dataguides and 1-indexes in the following proposition. (Proof omitted.)

Proposition 3.5 *Let $\approx$ be any approximation relation on the nodes of a database DB (i.e. $\approx$ satisfies Equation (1)), and let I be the 1-index constructed on DB using $\approx$. Then the deterministic automaton built from I by the standard powerset construction coincides with the strong dataguide.*

Thus, 1-indexes are non-deterministic alternatives to dataguides. Moreover, the two coincide on tree databases, (because in this case I, when viewed as an automaton, is deterministic.)

4 2-Indexes

In this section we describe index structures for answering queries of the form select x, y from $* \; x \; P \; y$, where the P can be any regular path expression. The template representing these queries is $* \; x_1 \boxed{P} \; x_2$. We again use language equivalence to form equivalence classes of nodes. But here we are interested in pairs of nodes (matching x_1 and x_2), so we will consider the language between pairs of nodes. Formally, define

$$L_{(v,u)}(DB) \overset{\text{def}}{=} \{w \mid \; w = a_1 \ldots a_n, \text{and there exists a path } v \overset{a_1}{\to} \ldots \overset{a_n}{\to} u \text{ in } DB\}$$

We write $L_{(v,u)}$ when DB is clear from the context. Now, define two pairs to be equivalent, $(v,u) \equiv (v',u')$, iff $L_{(v,u)} = L_{(v',u')}$, and let $[(v,u)]$ denote the equivalence class of (v,u). As before, computing $\equiv$ is prohibitively expensive, so we consider (efficiently computable) approximations, $\approx$, satisfying:

$$(y,u) \approx (v',u') \Longrightarrow (v,u) \equiv (v',u') \qquad (2)$$

As for the case of 1-index, it is possible to define efficient approximations $\approx$ using variants of the simulation or bisimulation relations. (Details are omitted for lack of space). Then, we define the 2-index $I^2(DB)$ of DB to be the following rooted graph.

[2]A simple path is a path which does not go through the same node twice.

[3]That is $u \approx_b v \Longrightarrow u \approx v$.

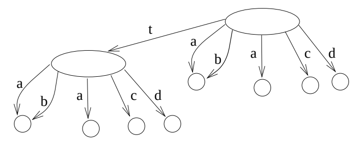

Figure 3: A 2-index for the data graph

Its nodes are equivalence classes (w.r.t $\approx$), $[(v,u)]$; the roots are all the equivalence classes of the form $[(x,x)]$; finally, there is an edge $s \xrightarrow{a} s'$ iff there exist v, u, u' s.t. $(v,u) \in s$, $(v,u') \in s'$, and DB contains an edge $u \xrightarrow{a} u'$. Besides the graph I^2 itself, we also store, for each state s, the extent of s, consisting of all pairs (v,u) in the equivalence class s.

Proposition 3.2 now becomes: $L_{(v,u)}(DB) = L_{[(v,u)]}(I^2(DB))$. Node that the $L_{(v,u)}(DB)$ on the left represent the paths between v and u in the database DB, while the $L_{[(v,u)]}(I^2(DB))$ on the right represents the paths, in the 2-index $I^2(DB)$, between some *root* of the index and $[(v,u)]$.

Query evaluation with 2-indexes proceeds similarly to that with 1-indexes, with small modification: To compute select x, y from $* \ x \ P \ y$, we compute the query path $P \ y$ on I^2 and take the union of the extents. Note that this saves the $*$ search: rather than searching for P from all the nodes in DB, in the index it suffices to look for P paths staring at the roots. These are often fewer than nodes in DB: For example, in acyclic databases, I^2 has a single root, because[4] $(u,u) \equiv (v,v)$ for every nodes $u, v \in DB$. Figure 3 shows the 2-Index (without extents) for the database in Figure 2 (a). It has a single root: the top node. The query select x, y from $* \ x \ a \ y$ is evaluated by traversing the outgoing a edges of that root.

As for 1-indexes, the storage of a 2-index consists of two parts: the graph and the extents. Both are now (at worst) quadratic in the size of DB. Again, while this guarantees that querying the index will not take more than querying the database, we would like to keep the index as small as possible. Our experiments (described breifly in the Appendix) indicate that in practice the index size is by far smaller than this upper bound, thus providing a significant improvement in query evaluation. A number of implementation techniques for further reducing the size of 2-indexes are also available, but they are beyond the scope of this paper, and are only mentioned briefly in the Appendix. On the theoretical side, Theorem 3.4 can be extended to 2-indexes for obtaining upper bounds on the size of the graph of I^2 which are independent on the size of DB: we omit this for lack of space.

[4]This remains true if we replace $\equiv$ with $\approx_b$ or $\approx_s$.

Connection to Related Work: Patricia Trees We conclude this section by explaining briefly the relationship to full-text indexing mechanisms and in particular to Pat trees [12, 21]. Its purpose is to assist in computing regular expressions over large text files. A Pat tree is a Patricia tree [9, 12] constructed over all the possible suffixes of a text (viewing the text as infinitely long), as follows. The root node will have one outgoing edge for each character in the file. Each of its children, say that corresponding to the letter k, will have one child for each character following that letter, e.g. the children may correspond to $ka, kb, kc, \ldots$ These nodes in turn will have one child for each continuation of that group of two characters, etc. If a node has only one child, that child is deleted, and the node is annotated with the number of descendents being omitted. The leaves of the tree point back into the data, to the beginning of the corresponding strings.

There exists a close relationship between Pat trees and 2-indexes, if we view a file consisting of a sequence of characters $a_1, a_2, \ldots$, as a graph database DB having a single long chain: $v_1 \xrightarrow{a_1} v_2 \xrightarrow{a_1} \ldots$ Here the 2-index for DB is a tree (note that the discussion above implies that the 2-index has a single root). The Pat tree can be obtained from the 2-index by performing some of the optimizations presented in the Appendix (namely (1) keeping only the x values in the extents, (2) skipping nodes and pointing back to the data whenever the descendents form a long chain, and (3) keeping extents only in leaf nodes).

5 T-Indexes

The 1-index and 2-index represent all the paths in the database (or all the paths from the root, in the case of 1-index) hence if the paths structure is very irregular, the index may become too large and hence inefficient. More performance improvement can be obtained if we restrict the class of queries which the index supports. This general principal has been applied successfully to relational and object oriented databases, where indexes are specific for one attribute, or for one fixed path. To illustrate with an example from semistructured data, consider the repository of cities in Figure 1. Assume that a high percentage of the query mix has the form select x_2 from $*.Restaurant \ x_1 \ R \ x_2$, where R is some arbitrary path expression: that is, the query conforms to the template $*.Restaurant \ x_1 \ \boxed{P} \ x_2$. Rather than indexing all the paths, it is more convenient to index only those having a *Restaurant* incoming edge. Another example is the case where most of the information in the database has a fixed, pre-defined structure, and only certain components are irregular. For example, consider the relation *Restaurants*(*Name*, *Phone*, *Menu*): *Name* and *Phone* have a fixed structure while the *Menu* attribute has a complex structure that differs from one restaurant to the other. We want to use standard optimization and indexing techniques for the structured parts, and focus our novel indexing mechanisms to the *Menu* part, where the standard ones do not apply.

We show here how the principles underlying the 1- and 2-indexes can be extended to more flexible index structures, capturing the above, and generalizing relational indexes, object-oriented path indexes, as well as 1- and 2- indexes. For the remainder of this section we consider a query template $t = T_1\ x_1\ T_2\ x_2 \ldots T_n\ x_n$, where each of the T_i's is either a path expression or a place holder $\boxed{P}$ or $\boxed{F}$. We build an index structure, called a *T-index*, to assist in answering queries $q \in inst(t)$. Before going into the definition of the index, we would like to point out that T-index both generalize and specialize 1 and 2-indexes, in certain ways. The generalization comes from the fact that both 1 and 2-indexes are particular cases of T-indexes (see below). But T-indexes also specialize 1 and 2-indexes, because of the following intuition. Suppose we built a T-index for a template t, and then want to evaluate a query $Q =$select x from $P\ x$. We can always use a 1-index to evaluate Q, but we can use the T-index only if the path expression P is in some sense "compatible" with the $T_1.T_2 \ldots T_n$ path in t: thus T-indexes reduce the class of path expressions they can evaluate. We will discuss below how to test whether a given query can be evaluated using a T-index.

Definitions In the case of 1 and 2-indexes we defined the language equivalence to be the equivalence relation on nodes, (resp. on pairs of nodes) in DB. We want to proceed similarly for arbitrary templates t. The difference is that here a query binds the variables $x_1, \ldots, x_n$ in some order, hence it makes sense to talk about identifying tuples of nodes corresponding to subsets of these n variables. We make a choice, and impose the evaluation strategy where the variables $x_1, \ldots, x_n$ are searched and bound in this order. This leads to the definition below. First, some notations: given a tuple $(v_1, \ldots, v_i)$ of nodes in DB, we use $\hat{L}_j$, $j = 1, i$, to denote the language $L_{(v_{j-1}, v_j)}$ for $j \geq 2$, and to denote the language L_{v_1} for $j = 1$.

Definition 5.1 *Let $t = T_1\ x_1 \ldots T_n\ x_n$ be a path template. Let $\$, S_1, \ldots, S_n$ be new data values not in $\mathcal{D}$. ($\mathcal{D}$ is the domain of data values from Definition 2.1.) For any i-tuple $(v_1, \ldots, v_i)$ of nodes in DB, $i = 1 \ldots n$, we define $T_{(v_1,\ldots,v_i)}(DB)$ to be the following regular language over the alphabet $\mathcal{D} \cup \{\$, S_1, \ldots, S_n\}$: $T_{(v_1,\ldots,v_i)}(DB) \stackrel{\text{def}}{=} R_1.\$.R_2.\$ \ldots R_i$, where the R_j's , $j = 1 \ldots i$ are the regular expression below:*

- *If $T_j = \boxed{P}$ (path template), then $R_j \stackrel{\text{def}}{=} \hat{L}_j$.*
- *If $T_j = \boxed{F}$ (formula template), then $R_j \stackrel{\text{def}}{=} \hat{L}_j \cap \mathcal{D}$. That is, R_j is the set of labels on all edges from v_{j-1} to v_j (where v_0 is a root).*
- *If $T_j = P_j$ (constant path expression): if $\hat{L}_j \cap W(P_j) \neq \emptyset$ then $R_j \stackrel{\text{def}}{=} S_j$, otherwise $R_j \stackrel{\text{def}}{=} \emptyset$.*

Finally, for two i-tuples $(v_1, \ldots, v_i)$ and $(u_1, \ldots, u_i)$ we define the language-equivalence relation, $(u_1, \ldots, v_i) \equiv (u_1, \ldots, u_i)$, iff $T_{(v_1,\ldots,v_i)}(DB) = T_{(u_1,\ldots,u_i)}(DB)$. The equivalence class of $(v_1, \ldots, v_i)$ is denoted $[(v_1, \ldots, v_i)]$

As before two tuples $(v_1 \ldots v_n)$, $(u_1 \ldots u_n)$ in DB can be distinguished by a query path $P_1\ x_1 \ldots P_n\ x_n$ in $inst(t)$ iff $(v_1, \ldots, v_n) \not\equiv (u_1, \ldots, u_n)$. The goal of the the \$ and the new S_i symbols is to pinpoint the range of each of the path term in the query, (and in particular those that match the constant path expressions in the template), and thus determine the assignments of nodes to the query variables. This issue will be further clarified below.

Here again, computing $\equiv$ is expensive, so we consider approximations, $\approx$, satisfying:

$$(v_1, \ldots, v_i) \approx (u_1, \ldots, u_i) \Longrightarrow (v_1, \ldots, v_i) \equiv (u_1, \ldots, u_i) \quad (3)$$

and that can be computed efficiently. As for the case of 1 and 2-index, it is possible to define efficient approximations using variants of the traditional simulation and bisimulation relations. (Details omitted).

Given such an approximation $\approx$, the T-index $I^t(DB)$ for t is the following rooted graph:
Nodes - The nodes include all the equivalence classes (w.r.t $\approx$) $[(v_1, \ldots, v_i)], i = 1, n$. Also, for each such class we introduce an additional new node which we denote $[(v_1, \ldots, v_i)]^\$$.
Edges - We have edges labeled by \$ from each node $[(v_1 \ldots v_{i-1}, v_i)]^\$$, $1 \leq i < n$, to $[(v_1 \ldots v_{i-1}, v_i, v_i)]$. Additionally, each T_i in the template $t = T_1\ x_1 \ldots T_n\ x_n$ introduces some edges, depending on its structure:

1. If $T_i = \boxed{P}$, then for each edge $v_i \xrightarrow{a} v_i'$ is in DB, I^t has an edge $[(v_1 \ldots v_{i-1}, v_i)] \xrightarrow{a} [(v_1 \ldots v_{i-1}, v_i')]$. Additionally, each $[(u_1 \ldots u_i)]$ has an edge to $[(u_1 \ldots u_i)]^\$$ labeled by a special ϵ symbol.
2. If $T_i = \boxed{F}$, then for each edge $v_i \xrightarrow{a} v_i'$ is in DB, I^t has an edge $[(v_1 \ldots v_{i-1}, v_i)] \xrightarrow{a} [(v_1 \ldots v_{i-1}, v_i')]^\$$.

3. If $T_i = P_i$, (i.e. a path expression), then for each node $[(v_1 \ldots v_{i-1}, v_i)]$ and every v_i' s.t. $L_{(v_i,v_i')} \cap W(P_i) \neq \emptyset$, I^t contains an edge $[(v_1 \ldots v_{i-1}, v_i)] \xrightarrow{S_i} [(v_1 \ldots v_{i-1}, v_i')]^\$$, where S_i is a new symbol.

Root nodes - The roots are all the nodes $[(v)]$ where v is a root of DB.
Terminal nodes - Unlike graph databases and 1 and 2-indexes, here we distinguish *terminal nodes*: these are all nodes of the form $[(v_1, \ldots, v_n)]^\$$.

Finally, we remove all nodes not reachable from a root or not having an outgoing path to a terminal node, and associate with each terminal node $[(v_1, \ldots, v_n)]^\$$ the extent containing all tuples in $[(v_1, \ldots, v_n)]$.[5]

Example 5.2 Consider the template $t = (*.Restaurant.*.Menu)\ x\ \boxed{P}\ y$. The equivalence classes are the following. For single nodes, u, there are exactly two classes $[(u)]$: the first, s_1, contains all nodes u reachable from a root via a path matching $*.Restaurant.*.Menu$, and the second, s_2, contains all the other nodes. Considering pairs next, the equivalence classes are now sets of pairs (u, v) for which $u \in s_1$ and which have the same language $L_{(u,v)}$; in addition there are similar equivalence classes for pairs (u, v) with $u \in s_2$.

I^t has one transition $s_1 \xrightarrow{S_1} s_1^\$$, continued with $s_1^\$ \xrightarrow{\$} [(u, u)]$, for all $u \in s_1$, has arbitrary transitions $[(u, v)] \xrightarrow{a} [(u, v')]$ for all edges $v \xrightarrow{a} v'$ in DB and all $u \in s_1$, and finally has transitions $[(u, v)] \xrightarrow{\epsilon} [(u, v)]^\$$, ending in a terminal state. Note that s_2 has no outgoing edges, hence all nodes $[u], [(u, v)]$ with $u \in s_2$ are removed from the graph. The resulting T-index looks like a 2-index that considers only the data reachable by a $*.Restaurant.*.Menu$ path.

Observe that every path from a root to a terminal node traverses exactly $n - 1$ \$-edges. We define $L_{[(u_1,\ldots,u_i)]}$ to be the language describing paths from the root to $[(u_1, \ldots, u_i)]$, with the slight modification that the ϵ symbols are interpreted as the epsilon moves (i.e. they are omitted from the strings).

Evaluating Query Paths with T-Indexes In the simplest scenario the query matches the template completely, i.e. $q = R_1\ x_1\ \ldots\ R_n\ x_n \in inst(t)$. First, let $P_q \stackrel{\text{def}}{=} P_1'.\$\ldots\$.P_n'$, where:

$$P_i' \stackrel{\text{def}}{=} \begin{cases} S_i & \text{when } T_i \text{ is a constant} \\ P_i \cap (\neg\$)* & \text{when } T_i \text{ is } \boxed{P} \text{ or } \boxed{F} \end{cases}$$

[5]As in the case of 1 and 2-indexes, when nodes in I^t have many outgoing edges, we can further index their labels.

Then evaluate the query path $P_q\ x$ on I^t, interpreting the ϵ edges as epsilon moves. Since P_q has exactly $n - 1$ \$-signs, all the retrieved nodes are of the form $[(v_1, \ldots, v_n)]^\$$. The answer to the query is the union of the extents of the retrieved nodes. The following guarantees the correctness of this algorithm.

Proposition 5.3 .
(1) Let $\approx$ be an approximation (i.e. satisfies Equation (3)) on DB. Then, for every $i = 1 \ldots n$ and every i-tuple $(v_1 \ldots v_i)$, we have $T_{(v_1,\ldots,v_i)}(DB) = L_{[(v_1,\ldots,v_i)]^\$}(I^t(DB))$.
(2) a tuple $(v_1, \ldots, v_n)$ satisfies a query q iff $W(P_q) \cap T_{(v_1,\ldots,v_n)}(DB) \neq \emptyset$, (where P_q is as defined above).

Evaluating More Complex Queries Sometimes we can use a T-index to evaluate queries $q \notin inst(t)$. We illustrate first with two examples.

Example 5.4 Let t and q be:

$$\begin{aligned} t &= \boxed{P}\ x\ ((B.A)*)\ y\ C\ z \\ q &= ((A.B)*).A\ y\ C\ z \end{aligned}$$

Obviously $q \notin inst(t)$, but we can still use I^t as follows. First instantiate t to $p = A\ x\ ((B.A)*)\ y\ C\ z \in inst(t)$ (we have instantiated $\boxed{P}$ with A). Then q can be expressed as a projection from p, namely as select y, z from p, because $A.(B.A)* = (A.B)*.A$.

Example 5.5 Let t and q be:

$$\begin{aligned} t &= A\ x\ B\ y\ C\ z \\ q &= A\ x\ B\ y\ (C.D)\ u\ E\ v \end{aligned}$$

Again $q \notin inst(t)$. Here t has a single instance, $p = A\ x\ B\ y\ C\ z$. We can use it to compute a prefix of q, namely the variables x and y, then continue to compute u,v with a search in the data graph. That is, we rewrite q as: select (x, y, u, v) from $p, y\ (C.D)\ u\ E\ v$. A subtle point here is that the unused "tail" of p, namely $C\ z$ is not harmful (it is implied by $y\ (C.D)\ u$). In effect we have replaced some prefix of q with an instance of t: we call this *prefix replacement*.

The general problem of deciding whether a path query q can be rewritten in terms of one or more T-indexes generalizes the *query rewriting problem* [15] to regular path expressions. We do not attempt to solve the rewriting problem for regular expressions: this is still open. Instead we identify restrictions under which the rewriting can be done efficiently.

Formally, given a template t and query path q with variables $x_1, \ldots, x_n$, we define a *prefix replacement* of q w.r.t. t to consists of (1) an instance

$p \in \mathit{inst}(t)$ (with proper variable renamings), and (2) a postfix q' of the query path q, such that the query select $(x_1, \ldots, x_n)$ from p, q' is equivalent to q. Checking whether a query path q admits a prefix replacement is PSPACE-hard. Indeed, given two arbitrary regular expressions R, R', they are equivalent iff the query path $q = R\ x$ has a prefix replacement w.r.t. the template $t = R'\ y$: equivalence of regular expressions is PSPACE-complete [22]. In the full version of the paper we prove the converse too: that checking whether there exists a prefix replacement (and finding one, when it exists) is in PSPACE. The proof consists in a careful reduction of the prefix replacement problem to two problems: (1) testing equivalence of regular path expressions (which is known to be in PSPACE), and (2) finding, for a regular expression R and number n, all n-tuples of regular languages $R_1, \ldots, R_n$ for which $R = R_1.R_2 \ldots R_n$: we prove that this problem is in PSPACE too.

Finally, we consider a particular case of templates and queries which we believe to be more frequent in practice. Define a regular path expression to be *simple* if it consists of a concatenation of (1) constants from $\mathcal{D}$, (2) _, and (3) $*$. For example $*.A.*.B.\_.\_.C$ is a simple regular path expression. Similarly, define a template to be *simple* if all its constant regular expressions (if it has any) are simple. We prove in the full version of the paper that checking/finding a prefix replacement for a simple query w.r.t. a simple template is in PTIME. At the core of this result lies a Lemma stating that containment of simple path expressions can be tested in PTIME. This may come at a surprise, since the deterministic automata associated to a simple regular path expression may have exponentially many states (proof omitted), hence the traditional containment test of regular languages would be much more expensive. Summerizing:

Proposition 5.6 *Given a template t and a query path Q, the problem whether there exists a prefix replacement of Q w.r.t. t is PSPACE complete. When both Q and t are simple, then the problem is in PTIME.*

Connection to Related Work T-indexes are flexible structures which can be fine-tuned to trade-off space for generality. They capture 1- and 2-indexes, by taking the templates $\boxed{P}\ x$ and $*\ x\ \boxed{P}\ y$ respectively. They also generalize traditional relational indexes: assuming the encoding of relational databases as in [7], an index on attribute A of the relation $R1$ can be captured with the template $(R1.\mathit{tup})\ x\ A\ y\ \boxed{F}\ z$. Finally, they generalize path indexes in OODBs. For example Kemper and Moerkotte describe in [14] *access support relation* (ASR), an index structure for query paths in OODBs. ASR's are designed to evaluate efficiently paths of the form $o.A_1.A_2 \ldots A_n$, where o is an object and $A_1, \ldots, A_n$ are attribute names. They define an *access support relation*, ASR, to be an $n+1$-ary relation R such that $(u, u_1, u_2, \ldots, u_n) \in R$ iff there exists a path $u \overset{A_1}{\to} u_1 \overset{A_2}{\to} u_2 \ldots u_n$ in the database. Ignoring the mismatch between the object-oriented and the semistructured data model, there exists a close relationship between an ASR and the T-index for the template $*\ x\ A_1\ x_1\ A_2\ x_2\ \ldots\ A_n\ x_n$. The graph structure of the T-index would be a chain of $2n$ nodes $[(r)] \to [(r)]^\$ \to [(u, u_1)] \to [(u, u_1)]^\$ \to [(u, u_1, u_2)] \to \ldots \to [(u, u_1, u_2, \ldots, u_n)]^\$$, where the last (terminal) node has an associated extension: this extension is precisely the ASR.

6 Conclusions

We presented an indexing mechanism, called T-index, aimed to assist in evaluating query paths in semistructured data. A T-index captures the (possibly partial) knowledge about the structure of data and the type of queries in the query mix, as described by a *path templates.*

Abiteboul and Vianu consider in [3] First-Order equivalence classes over tuples of values in the database. Two tuples $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ are equivalent if they are indistinguishable by any FO formula. The language equivalences on which we base our index constructs are only superficially related to the FO equivalence classes: the queries we consider to distinguish between two tuples are only chain queries. Hence the language equivalences are coarser than FO equivalences, and results in fewer equivalence classes.

Buchsbaum, Kanellakis, and Vitter consider in [5] the problem of incrementally maintaining query paths given by a fixed regular expression under either database insertions or deletions (but not both). They describe an efficient method for incremental updates. Since their method refers to a fixed regular expression, it could be used in incremental updates of T-indexes but only when the template is restricted to constant regular expressions. We do not address index maintenance here, but note that a possible alternative to incremental maintenance can be based on the optimization technique mentioned in the Appendix, of pointing back to the data, doing so whenever a portion of the index graph is invalidated by an update.

Acknowledgment: We thank Micky Frankel for the implementation of the 1-, 2- and T-indexes.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] Serge Abiteboul. Querying semi-structured data. In *ICDT*, 1997.

[3] Serge Abiteboul and Victor Vianu. Generic computation and its complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*, 1991.

[4] Elisa Bertion and Won Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[5] Adam Buchsbaum, Paris Kanellakis, and Jeffrey Scott Vitter. A data structure for arc insertion and regular path finding. *Annals of Mathematics and Artificial Intelligence*, 3:187–210, 1991.

[6] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[7] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1996.

[8] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In Richard Snodgrass and Marianne Winslett, editors, *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

[9] P. Flajolet and R. Sedgewick. Digital search trees revisited. *SIAM Journal on Computing*, 15:748–767, 1986.

[10] Harold Gabow and Robert Tarjan. Faster scaling algorithms for network problems. *SIAM Journal of Computing*, 18(5):1013–1036, 1989.

[11] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *VLDB*, September 1997.

[12] G. Gonnet. Efficient searching of text and pictures (extended abstract). Technical Report OED-88-02, University of Waterloo, 1988.

[13] Monika Henzinger, Thomas Henzinger, and Peter Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of 20th Symposium on Foundations of Computer Science*, pages 453–462, 1995.

[14] Alfons Kemper and Guido Moerkkotte. Access support relations: an indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.

[15] Alon Levy, Alberto Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the 14th Symposium on Principles of Database Systems*, San Jose, CA, June 1995.

[16] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proceedings of the Fourth Conference on Parallel and Distributed Information Systems*, Miami, Florida, December 1996.

[17] Robin Milner. *Communication and concurrency.* Prentice Hall, 1989.

[18] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.

[19] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: concise representation of semistructured, hierarchical data. In *ICDE*, 1997.

[20] Robert Paige and Robert Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16:973–988, 1987.

[21] A. Salminen and F. W. Tompa. Pat expressions: an algebra for text search. In *Papers in Computational Lexicography: COMPLEX'92*, pages 309–332, 1992.

[22] L. J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. In *5th STOC*, pages 1–9. ACM, 1973.

A Appendix

Bisimulation, Simulation For completeness, we include here the definition of a bisimulation and a simulation. Note that we need to slightly modify the traditional definitions and "reverse" the directions of edges, because L_v in our context refers to the set of paths leading *into* v, rather than *from* v as typically found in the literature.

Definition A.1 *Let DB be a data graph. A binary relation $\sim$ on its nodes is called a* reversed bisimulation *if it satisfies:*

1. *If $v \sim v'$ and v is a root, then so is v'.*
2. *Conversely, if $v \sim v'$ and v' is a root, then so is v.*
3. *If $v \sim v'$, then for any edge $u \xrightarrow{a} v$ there exists an edge $u' \xrightarrow{a} v'$, s.t. $u \sim u'$.*
4. *Conversely, if $v \sim v'$, then for any edge $u' \xrightarrow{a} v'$ there exists an edge $u \xrightarrow{a} v$, s.t. $u \sim u'$.*

A binary relation $\sim$ is called a reversed simulation*, if it satisfies conditions 1 and 3.*

We say that two nodes v, u are *reversed bisimilar*, in notation $v \approx_b u$, iff there exists a reversed bisimulation $\sim$ s.t. $v \sim u$. There always exists a maximal reverse bisimulation, that it is an equivalence relation, and that it is precisely $\approx_b$. Paige and Tarjan [20] describe an $O(m \log n)$ time algorithm for computing the maximal bisimulation on a unlabeled graph with n nodes and m edges, which can be easily adapted to a $O(m \log m)$ algorithm for labeled graphs [6].

For the case of reversed simulation, there also exists a maximal one, which we denote $\preceq$, however it is not an equivalence relation in general, but a preorder[6]. We say that two nodes v, u are *reversed similar*, if $v \preceq u$ and $u \preceq v$, and use the notation $v \approx_s u$. Henzinger, Henzinger, and Kopke[13] give an $O(mn)$ algorithm for computing the simulation relation on an unlabeled graph with n nodes and m edges, from which one can derive an $O(m^2)$ algorithm for labeled graphs [6].

Proof of Proposition 3.1 Recall that we only consider *accessible* graph databases, i.e. in which every node is accessible from some root. We will show that $\equiv$ is a reversed bisimulation: this proves that $v \equiv u \Longrightarrow v \approx_b u$, and the proposition follows. We check the four conditions in Definition A.1. If $v \equiv u$ and v is a root, then $\varepsilon \in L_v$, hence $\varepsilon \in L_u$, so u is a root too. This proves items 1 and 2. Let $v \equiv u$ and let $v' \xrightarrow{a} v$ be some edge. Hence $L_v = L_1.a \cup L_2$, where $L_1 = L_{v'}$, while L_2 is a language which does not contain any words ending in a (because DB has unique incoming labels). It follows that $L_u = L_1.a \cup L_2$. Since v' is an accessible node in DB, we have $L_1 \neq \emptyset$, hence there exists some edge $u' \xrightarrow{a} u$ entering u, and it also follows that $L_{v'} = L_{u'}$.

[6]That is, $\preceq$ is reflexive and transitive, but not necessarily symmetric.

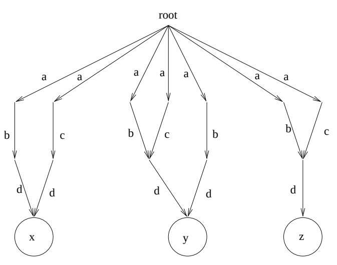

Figure 4: A data graph on which the relations $\equiv$, $\approx_s$, and $\approx_b$ differ.

Proof of Proposition 3.2 The inclusion $L_v \subseteq L_{[v]}$ holds for any equivalence relation $\approx$, not only approximations: this is because any path $v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} v_2 \ldots$ in DB, with v_0 being a root node, has a corresponding path $[v_0] \xrightarrow{a_1} [v_1] \xrightarrow{a_2} [v_2] \ldots$ in I. For the converse, we prove by induction on the length of a word w that, if $w \in L_{[v]}$, then $w \in L_v$. When $w = \varepsilon$ (the empty word), then $[v]$ is a root of I: hence $v \approx r$ for some root r. This implies $L_v = L_r$, so $\varepsilon \in L_v$. When $w = w_1.a$, then we consider the last transition in I: $s \xrightarrow{a} [v]$, with $w_1 \in L_s$. By definition there exists nodes $v_1 \in [s]$ and $v' \in [v]$ and an edge $v_1 \xrightarrow{a} v'$ and, by induction, it follows $w_1 \in L_{v_1}$. This implies $w \in L_{v'}$. Now we use the fact that $\approx$ is an approximation, to conclude that $w \in L_v$.

Proof of Theorem 3.4 It suffices to prove the statement for the case when $\approx$ is the reversed bisimulation relation. We will show that in this case there are at most $c(k, k, p)$ equivalence classes under reversed bisimulation, in any database DB with k-short paths and at most p distinct labels. Here $c(k, d, p)$ is defined by:

$$c(k, 1, p) \stackrel{\text{def}}{=} 2(k+1) \qquad (4)$$

$$c(k, d+1, p) \stackrel{\text{def}}{=} 2k + 1 + 2^{p \times c(k,d,p)} \qquad (5)$$

First a matter of terminology. Due to our particular setting, our edges turned out to be in the opposite direction than traditionally. Thus we have in this proof reversed bisimulations, and reversed trees, i.e. with paths leading into the root, rather than out of. We will drop the attribute "reversed". Note that in the case of trees edges lead now from children to parents.

Consider some node $u \in DB$. Define $T(u)$ to be the infinite, reversed unfolding of DB at u. That is $T(u)$ is a (possibly infinite) tree, having u as its root, such that for every path in DB labeled $a_1.a_2 \dots a_n$ ending in u, there exists a unique corresponding path in $T(u)$ ending in u, with the same labels. Each node in DB may correspond to several nodes in $T(u)$, possibly to infinitely many. We will use the same notation for the nodes in DB and those in $T(u)$: thus we will talk about two nodes x, y in $T(u)$ as being "equal", $x = y$. Recall that DB had its own root node(s): we call their unfoldings in $T(u)$ the *old roots*. The importance of these trees $T(u)$ is the following. For any two nodes u, v in DB, we have $u \approx v$ iff there exists a bisimulation $\sim$ between $T(u)$ and $T(v)$. Hence, in order to count the number of equivalence classes $[u]$ it suffices to count the number of equivalence classes of infinite trees $T(u)$. Here, and in the sequel, the definition of a bisimulation between two graphs $T(u)$ and $T(v)$ is exactly as in Definition A.1, where items 1 and 2 are required both for "roots" and for "old roots".

As a matter of terminology, we will classify $T(u)$'s nodes into levels: thus the root u is on level 1, it children are on level 2, etc. For each such tree $T(u)$ we identify a certain set of nodes which can be cut. For that we consider all paths of length $\leq k+1$ in $T(u)$ ending in the root u, $x \to u_{j-1} \to \dots \to u_2 \to u_1 = u$, such that $u_{j-1}, u_{j-2}, \dots, u_1$ are distinct nodes while x is equal to one of $u_{j-1}, \dots, u_2, u_1$, say $x = u_i$, for $i < j$. We define x to be a *cut node*, and call i its *index*. Note that the subtrees of $T(u)$ rooted at u_i and at x are isomorphic. Since DB is k-short, all its cut nodes are on levels $\leq k+1$, and there are only finitely many.

Next we construct a finite tree $D(u)$, by actually performing the cuts. That is, for each cut node x as before, we delete all its children (and children's children etc.): x becomes a leaf. We label the new leaf with a special symbol σ_i, where i is the index of x, with the intend to recapture the information lost by cutting: the level number i will help us restore the lost information. Repeating this for all cut nodes gives us a finite tree, $D(u)$, of depth $\leq k$, in which some of the leaves are labeled with one of $\sigma_1, \dots, \sigma_k$.

The importance of $D(u)$ lies in the following fact. If there exists a bisimulation between $D(u)$ and $D(v)$, then there also exists a bimimulation between $T(u)$ and $T(v)$. Hence there will be at most as many bisimulation equivalence classes for the $T(u)$'s as for the $D(u)$'s: the latter are easier to count.

We prove the fact first. For these new kind of trees $D(u)$ we change the definition of bisimulation by adding the requirement that, whenever $x \approx y$ and x is labeled with σ_i, then y is labeled σ_i too, and vice versa. Consider a bisimulation $\approx$ between $D(u)$ and $D(v)$. To show that $T(u)$ and $T(v)$ are bisimilar, we consider some intermediate construction first. Namely define $B(u)$ (and $B(v)$ similarly) to be the graph obtained from $D(u)$ by fusing every cut node x with u_i, where i is the index of x. That is, we delete the node x, redirect the unique edge $x \to y$ to $u_i \to y$, and keep the same label on the edge: we call the new edge a *back edge*. $B(u)$ is not a tree anymore, since back edges introduce cycles. However, its infinite unfolding at the node u is isomorphic to $T(u)$, and similarly for $T(v)$ and $B(v)$. Hence it suffices to prove that $B(u)$ and $B(v)$ are bisimilar. Recall that we have a bisimulation relation $\approx$ between $D(u)$ and $D(v)$. Define first a subset of the binary relation $\approx$ as: $x \sim y$ iff $x \approx y$, x, y are on the same level, and their parents x', y'[7] satisfy $x' \sim y'$. The fact that $D(u)$ and $D(v)$ are trees ensures that $\sim$ remains a bisimulation. We prove now that $\sim$ is a bisimulation between $B(u)$ and $B(v)$. Obviously it satisfies conditions 1 and 2 of Definition A.1, both for all old roots and for the "real" roots u and v. We check condition 3: assume $y \sim y'$, and consider only back edges $u_i \to y$ in $B(u)$ (for regular edges it is trivial), with u_i a node on level i. It corresponds to an edge $x \to y$ in $D(u)$, where x is labeled σ_i. Since $\sim$ is a simulation btw. $D(u)$ and $D(v)$, there exists an edge $x' \to y'$ in $D(v)$, with x' also labeled σ_i: hence x' is a cut node too, and in $B(v)$ we will find a corresponding back edge $u'_i \to y'$, with u'_i also on level i. Since $y \sim y'$ and both u_i and u'_i are their ancestors, and on the same level, it follows that $u_i \sim u'_i$ (that's the way we designed $\sim$). This proves that $B(u)$ and $B(v)$ are bisimilar.

Finally we count the number of equivalence classes under bisimulation for finite trees of depth $\leq d$, in which some leaves may be labeled with $\sigma_1, \dots, \sigma_k$, and in which some nodes may have been designated "old roots". We prove that $c(k, d, p)$, given by equations (4) and (5) is an upper bound for that number. Indeed, for $d = 1$, each such tree consists of a single node, which is by necessity the ("real") root. In addition it may be designated an old root or not, and it may be labeled with one of $\sigma_1, \dots, \sigma_k$, or not at all. In total there are $2(k+1)$ choices. For the induction case, consider some tree of depth $\leq d+1$. It is either a single node (which brings us back to the previous case, and give the $2(k+1)$ summand in Equation (5)), or has a "real" root with a non-empty set of direct children. In the latter case we start by dropping the duplicate subtrees. We obtain a bisimilar tree, which the root has at most $p \times c(k, d, p)$ children (every label on the edge paired with every possible bisimulation equivalence class for trees of depth $\leq d$). Hence,

[7]That is, there exists edges $x \to x', y \to y'$.

Data	Graph Size	1-index	2-index
Bibtex	150	40	50
Web site	1521	198	1100

Table 1: Experiments showing index size

there are at most $2^{p \times c(k,d,p)} - 1$ equivalence classes under bisimulation: adding the two summands gives us Equation (5).

Experiments Recall that index storage consists of two parts: the graph I and the extents. The graph carries the schematic information, and its size is critical for query performance: the graph distinguishes our index structures from traditional indexes. We are currently conducting a series of experiments to asses the size of the index graphs: some of the results are reported in Table 1. We are testing the technique on a variety of graph types, including relatively structured ones (Bibtex data), loosely structured Web data (in particular the Web site of the CS department of Tel-Aviv university), randomly generated graphs, and mixed graphs composed of components of the above types. We briefly describe these experiments here. In order to asses the schematic information we measure only the the number of non-leaf nodes in the graphs.

We started by considering 1 and 2-indices. Not surprisingly, the smallest indices were obtained for the BibTex data: although the structure of BibTex items may vary (hence a collection of such items is naturally modeled by the semi-structured data model), the number of possible paths between nodes is rather limited. We considered increasingly growing files and their corresponding graph representation. Already at 150 nodes the size of the 1 and 2-indices almost stabilized having about 40 and 50 vertices resp., staying at about the same size regardless of the growth of the data, and thus providing significant performance improvement when querying large files. Observe that the independence of the index size from the data size is also implied here from Theorem 3.4, but the experimental results show that in practice the index size is much smaller than the theoretical upper bound induced by the proof of the theorem.

To evaluate the technique in a less structured environment we considered the Web site of the CS department in the Tel-Aviv university. It should be noted that the pages in the site are each built and maintained individually by distinct people without significant constraints on the structure, and are *not* automatically generated by some application, as done in some organizations, hence the structure is rather loose and makes the site a typical example for semi-structured information. For a graph of about 1500 nodes, the size of the 1-index amounts to about 13% of the original size, and that of the 2-index to about 72%. Observe that the later is only 0.0475% of the potential upper bound on the size of the 2-index, which is the square of the number of nodes in the graph ! This also implies that the effort in evaluation of queries of the form $* \; x_1 \; P \; x_2$ on the original data can potentially be as much as square of that needed when using the 2-index. (Since on DB we need to evaluate the query from each node, while on I we just evaluate $P \; x_2$ from the I's root.)

The usage of T-indices for focusing on specific, more interesting, parts of the data was tested on mixed graphs combining randomly generated subgraphs with BibTex or Web site-like data, and using templates focusing on the BibTex/Web parts. The reduction is size was similar to the one reported above and more, depending on the size of the random-generated parts being ignored in the construction due to the given template.

Techniques for Reducing the Size of an Index Graph We describe here three such methods. We first explain how they work for 1-indexes, then describe briefly how the techniques are generalized to 2- (or T-) indexes.

Normalizing labels: In many cases, distinct labels in the database graph are synonyms denoting the same concept. For example, the labels "first name", "first-name", "fname", "First Name", may all refer to the same thing. But still, the 1-index, as described above, stores each of them separately. To avoid this, on may chose to "normalize" the database before constructing the index by applying some normalization function ν to all the labels in DB, and thus reduce the size of the index. We denote the 1-index thus obtained by $I(\nu(DB))$. It is easy to see that

Proposition A.2 *If all the formulas in a query path* $q = Px$ *have the property that for every value* $d \in \mathcal{D}$, $f(d)$ *holds iff* $f(\nu(d))$ *holds, then* q*, when evaluated using* $I(\nu(DB))$ *computes exactly the answer of* q *for the original* DB.

Pointing back to the database: Assume we have some equivalence class (node) s in the 1-index having many descendants, all representing very small equivalence classes. Note that we gain very little by computing on this part of the index because it is almost as big as the corresponding part of the data graph. So we would like to remove it and avoid the duplication of data.

Let S be a set of nodes in I having this property. We can reduce I's size by redirecting all edges leaving from a node s in S to point into the database DB. That is, for every node $s \in S$ we delete all its outgoing edges $s \xrightarrow{a} s'$, and for each of them we introduce new edges from I into DB, from s into each $v \in \mathsf{extent}(s')$. The root(s) of the resulting combined structure will still be I's old root(s), and as before, the queries will be evaluated on the hybrid index starting from these roots. The space gain consists in removing those parts of I which are no longer accessible from the root(s). This may come at the cost of increased computation time, since now part of the search is done on DB.

Note that this hybrid index does not have anymore the property of 1-indexes that all the node extents are disjoint, (because a database node may appear in both the I and the DB parts of the structure.) Nevertheless, we prove in the full paper that computing on this hybrid structure still yields a correct result.

Dropping extents: Alternatively, we can keep the entire index I, but delete some of the extents. The computation of a regular path expression on such a reduced index proceeds as follows. Let A be some (nondeterministic) automaton equivalent to regular expression, and let $G = (I \times A)^{acc}$ be the standard product automaton, in which we only retain the accessible part. Consider all states (s, t) of G, where t is a terminal state in A. If $s \in I$ has an associated extent, then include $\mathsf{extent}(s)$ in the result: so far the computation is similar to that described in Section 3. Otherwise, we have to "backtrack" in I up to some state which does have an extent, and proceed from there in the database. This should be easy to visualize when I is a tree. In the general case, the backtrack step proceeds as follows. We compute in G a *cut*, meaning a set of nodes S which separates G's initial states from (s, t): more precisely, S has the property that any path from an initial state in G to (s, t) goes through some node in S. A cut can be found efficiently (in PTIME, with low degrees [10]). Moreover, in computing the cut, we only consider states of the form (s', x) where s' has an extent in I. Then, the backtrack step consists in considering for each $(s', x) \in S$ the automaton $A(x, t)$ obtained from A by considering x the initial state and t the only terminal state, and computing $A(x, t)$ on DB with $\mathsf{extent}(s')$ as roots.

2- (and T-) indexes: A combination of the techniques discussed above can help us keep the size under control: First, the previous result regarding the normalization of labels holds here as well. Next, regarding dropping the extents, note that the extents here contain pairs (or tuples) of nodes and not individual objects. So rather than dropping an extent completely, we can also take a compromising approach and just drop one (or some) of the attributes. For example, if we know that most of the queries are interested only in the x_1 variable of $* \ x_1 \ P \ x_2$, then the x_2 attribute can be dropped, thus reducing the size of the extents. To restore it, if needed, we can switch to the x_1 nodes on DB and look for the corresponding x_2's there. This can be combined with the technique for pointing back to the data, accept that now whenever the computation is moved from the index to the database, we need to remember which x_1 value caused the transition and pair it with the retrieved x_2 nodes. Furthermore, when queries are interested only in the x_1 values, then more reduction can be obtained by the following observations. Note that in acyclic parts of the graph, the x_1 values in a node extent is he union of the x_1 values in the extents of its children. So we may decide to drop an extent, and if needed compute it (perhaps recursively) from its children. Finally, if the index contains chains of nodes all having the same extent and the same set of outgoing edges, we can skip those nodes, and just record the number of repetitions.