Index Structures for Path Expressions

Tova Milo[1] and Dan Suciu[2]

[1] Tel Aviv University milo@math.tau.ac.il
[2] AT&T Labs suciu@research.att.com

1 Introduction

In recent years there has been an increased interest in managing data that does not conform to traditional data models, like the relational or object oriented model. The reasons for this non-conformance are diverse. On the one hand, data may not conform to such models at the physical level: it may be stored in data exchange formats, fetched from the Web, 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 schema may be too complex or changes too often. The term *semistructured* data has been used to refer to such data. The semistructured data model 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 *regular* 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:

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

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

How are such queries evaluated? A naive evaluation that scans the whole database 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. Index structures developed for traditional data models are tied to a pre-defined schema: e.g. relational databases index on a specific attribute of a specific relation, while object-oriented databases index on a specific path in the object-oriented schema [4, 14], e.g. *document.section.title*. Hence, these index structures are not applicable to semistructured data, because here the schema is unavailable. Full text indexing systems take an opposite approach: given no knowledge on the structure of information, they index *all* the data. But this is of limited use for semistructured data, where some (perhaps very partial) knowledge on the structure may be available and exploited in path expressions: e.g. the query above insists that a *Dinner* item appears inside a *Menu*.

Recent work has addressed the problem of path expressions evaluation in 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. Dataguides are restricted to a single regular expression and are not useful in more complex queries with several regular expressions, like the one above.

In this paper we propose a novel, general index structure for semistructured databases, called *template index*, or T-index. It improves over the previous approaches in several ways. First, T-indexes 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. Another example is $(*.Restaurant)\ x\ \boxed{P}\ y$, in which the first regular expression is fixed to $*.Retaurant$: this T-index takes less space but is less general. Second, T-indexes can be efficiently constructed. Dataguides [11] require 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 expression 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. Forth, we show that T-indexes turn out to be an 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].

Our technique consists in 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 [6]. 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 regular path expressions can be rewritten to take advantage of a given T-index. In this formulation the problem generalizes the query rewriting problem [15] to the case of queries with regular path expressions which, 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 is non obvious, because the associated deterministic automaton in this case is still exponential in the size of the regular expression.

In section 2 we review the data model and query language for semi-structured data and introduce the notion of *path templates*. We first consider two specific templates in Sections 3 and 4 whose corresponding indexes we call 1 and 2-index respectively: we illustrate details of our techniques on these simpler cases. General T-indexes are presented in Section 5. We conclude in section 6.

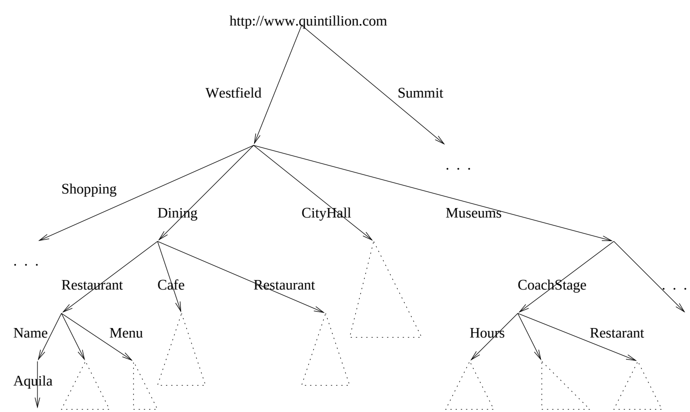

Fig. 1. Example of a semistructured database with small towns in New Jersey.

2 Review: Data Model and Query Languages

We review here the basic framework on semistructured databases and queries.

The data model: Semistructured data is modeled as a labeled graph, in which nodes correspond to the objects in the database, and edges to their attributes. We assume an infinite set $\mathcal{D}$ of *data values* and an infinite set $\mathcal{N}$ of *Nodes*.

Definition 1. *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: 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 boolean combinations of such predicates. We assume that we have effective procedures for evaluating the truth values of sentences $f(d)$ and $\exists x.f(x)$, for $f \in \mathcal{F}$ and $d \in \mathcal{D}$.

We define *regular path expressions*, or *path expressions*, P, over formulas in $\mathcal{F}$: $P ::= \emptyset \mid \epsilon \mid f \mid (P|P) \mid (P.P) \mid P*$. We denote with $L(P)$ the regular language defined by P, and with $W(P)$ the set of all words $w = a_1 \dots a_n$ 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 words obtained by replacing each formula by some value that satisfies it). 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 $W(P)$.

We denote with _ the predicate $True$, and abbreviate _$*$ with $*$. Each constant $d \in D$ also denotes a predicate d s.t. $\forall x \in D$, $d(x)$ is true iff $x = d$. For example $*.Restaurant. * .Name.\_.Fridays$ is a 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 $1 \leq i_1 < i_2 < \dots < i_k \leq n$. That is, a query consists of a query path and a set of head variables. The query in Section 1 has this form. 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. In the sequel we will assume w.l.o.g. that the head variables are either $x_1, \dots, x_{n-1}$ or $x_1, \dots, x_n$. In the latter case we will refer to the query by giving only the query path.

Path Templates: A *path 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 query path q is obtained from t by instantiating each of the $\boxed{P}$ place holders by some regular path expressions, and each of the $\boxed{F}$ place holders by some formula. The query path thus obtained is called an *instantiation* of the path template t. The set of all such instantiations is denoted $inst(t)$.

For example, consider the path 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 path 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 other queries as well). 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 path template such as the one above. As another example, assume we know nothing about the database, but users never ask for more than k objects on a path. Then we may take $t = \boxed{P} x_1 \boxed{P} x_2 \ldots \boxed{P} x_k$, and build the corresponding index.

In the next two sections we focus on two particular templates $t_1 = \boxed{P}\ x$ and $t_2 = *\ x_1 \boxed{P}\ x_2$, and illustrate most technical details on these simpler indexes, called 1- and 2-indexes. We present the general case in Sec. 5.

3 1-Indexes

Our goal here is to compute efficiently queries $q \in inst(\boxed{P}\ x)$.

A First attempt: A naive way (which we will soon refine) is the following. For each node v in DB, let $L_v(DB)$, or L_v in short, be the set of words on paths from some root node to v:

$$L_v(DB) \stackrel{\text{def}}{=} \{w \mid w = a_1 \ldots a_n, \exists \text{ a path } v_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} v, \text{with } v_0 \text{ a root node}\}$$

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

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

We denote with $[v]$ the equivalence class of v. Clearly, there are no more equivalence classes than nodes in DB. 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 $\mathsf{extent}(s_i)$ for which this intersection is not empty.

This naive approach is inefficient, for two reasons.

- *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.

[1] *Distinguished* means that one node is in the query's answer while the other is not.

We next address these problems. To tackle the construction cost we consider refinements. An equivalence relation $\approx$ is called a *refinement* if:

$$v \approx u \Longrightarrow v \equiv u \tag{1}$$

As we shall see, any refinement is fine for constructing 1-indexes, as long as it is efficiently computable: we illustrate below two examples. The basic idea to tackle the index size was introduced in [19], and consists in a more concise representation of 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 an even more compact structure.

Refinements: We discuss here two choices for refinements: *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 here. Unlike standard definitions in the literature we need to traverse edges "backwards", because L_v refers to the set of paths leading *into* v.

Definition 2. *Let DB be a data graph. A binary relation $\sim$ on its nodes is a* backwards bisimulation *if:*

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 $\preceq$ is a backwards simulation*, if it satisfies conditions 1 and 3.*

Since we consider only backwards simulations and bisimulations in this paper we safely refer to them as *simulation* and *bisimulation*.

Two nodes v, u are *bisimilar*, in notation $v \approx_b u$, iff there exists a bisimulation $\sim$ s.t. $v \sim u$. Paige and Tarjan [20] describe an $O(m \log n)$ time algorithm for computing $\approx_b$ 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].

Two nodes v, u are *similar*, in notation $v \approx_s u$, if there exists two simulations $\preceq, \preceq'$ s.t. $v \preceq u$ and $u \preceq' v$. Henzinger, Henzinger, and Kopke[13] give an $O(mn)$ algorithm for computing $\approx_s$ on an unlabeled graph with n nodes and m edges, which can be easily adapted to an $O(m^2)$ algorithm for labeled graphs [6].

We have: $v \approx_b u \Longrightarrow v \approx_s u \Longrightarrow v \equiv u$, hence both $\approx_b$ and $\approx_s$ are refinements, i.e. satisfy Equation 1. The implications are strict as illustrated in Figure 2, where $x \equiv y \equiv z$, $x \not\approx_s y \approx_s z$, and $x \not\approx_b y \not\approx_b z$.

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 a refinement 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

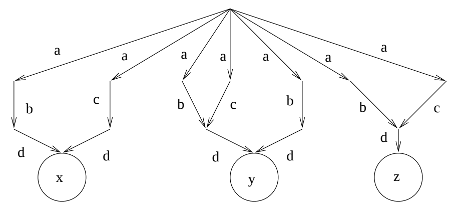

Fig. 2. A data graph on which the relations $\equiv$, $\approx_s$, and $\approx_b$ differ.

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.

Proposition 1. *If DB is a graph database with unique incoming labels, then $\equiv$, $\approx_s$, and $\approx_b$ coincide.*

Proof. 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 bisimulation: this proves that $v \equiv u \Longrightarrow v \approx_b u$, and the proposition follows. We check the four conditions in Definition 2. 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 $L_{v'} = L_{u'}$.

1-Indexes: We can now define 1-indexes. Given a database DB and a refinement $\approx$, the 1-index $I(DB)$ is a rooted, labeled graph defined as follows. Its nodes are equivalence classes $[v]$ of $\approx$; for each edge $v \xrightarrow{a} v'$ in DB there exists an edge $[v] \xrightarrow{a} [v']$ in $I(DB)$; the roots are $[r]$ for each root r in DB. When DB is clear from the context we omit it and simply write I.

We store I 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 contain each node in DB exactly once. This is similar to an unclustered index in a relational databases, where each tuple id occurs exactly once in the index.

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:

Proposition 2. *Let $\approx$ be a refinement (i.e. satisfies Equation (1)) on DB. Then, for any node v in DB, $L_v(DB) = L_{[v]}(I(DB))$.*

Proof. The inclusion $L_v \subseteq L_{[v]}$ holds for any equivalence relation $\approx$, not only refinements: this is because any path $v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} v_2 \ldots$ in DB, with v_0 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 edge 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, $w_1 \in L_{v_1}$. This implies $w \in L_{v'}$. Now we use the fact that $\approx$ is a refinement, to conclude that $w \in L_v$.

The cost of evaluating a query $P\ x$ on a graph is polynomial in the size of the graph and that of the query path[2]. Since $I(DB)$ is likely to be smaller than DB, evaluating a query on $I(DB)$ rather than DB is faster. 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 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 1. Fig. 3 (a) illustrates a graph data DB and Fig. 3 (b) its 1-index I. Considering the query $q = t.a\ x$, its evaluation follows the two paths $t.a$ in I (rather than the 5 in DB), and unions their extents: $\{7, 13\} \cup \{8, 10, 12\}$.

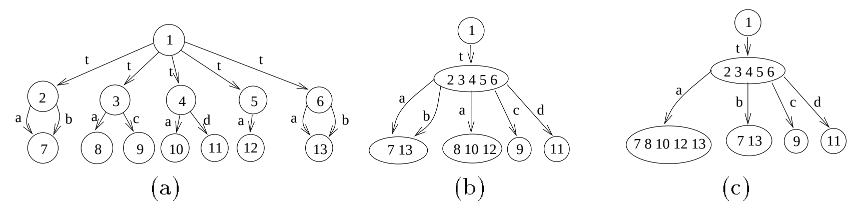

Fig. 3. A data graph (a), its 1-index (b), and its strong dataguide (c)

[2] We assume here that each predicate $p(d)$ can be computed in $O(1)$, for each $d \in \mathcal{D}$.

The Size of a 1-Index: The storage of a 1-index consists of the graph I and the sum of all extents. Query performance is dictated by the former, which we discuss here. On the experimental side we computed $I(DB)$ for a variety of databases, obtaining results which show that in common scenarios I is significantly smaller than DB. A brief discussion of the experiments is given in the Appendix. On the theoretical side we identified two parameters which alone control the size of I. These are: (1) the number of distinct labels in DB, and (2) the longest simple path[3]. We show here that there exists an upper bound for the size of I which depends only on these two parameters and is otherwise independent on DB. Technically this is one of the hardest results in this paper, whose proof will be included in the full version (omitted here for lack of space), 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 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 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 1. *Let DB be a k-short database having at most p distinct labels, and let $\approx$ be any refinement which is at least as coarse as a bisimulation[4]. Then the size of I is bounded by some number depending only on k and p, and is independent on the size of DB.*

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 each path in the data is represented exactly once in the dataguide. If we view DB as an automaton by making each each node a terminal state and each root an initial state, then a dataguide is by definition a deterministic automaton equivalent to DB. Among all dataguides only one has states which are related to sets of nodes in DB (what we call here *extents*): this is called a *strong dataguide*, and is precisely the standard powerset automaton of DB. Unlike 1-indexes, the extents in a strong dataguide may overlap. Hence, the storage for dataguides can be larger than 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 exactly the number of nodes in DB. A contribution of our work is to show that, by relaxing the determinism requirement

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

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

imposed on dataguides, the 1-indexes can be constructed and stored more efficiently, while at the same time serve similar goals. We pinpoint the relationship between dataguides and 1-indexes in the following proposition. (Proof omitted.)

Proposition 3. *Let $\approx$ be any refinement 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 construct coincides with the strong dataguide.*

Referring to the data graph DB in Fig. 3 (a) and index I in Fig. 3 (b), the strong dataguide is shown in Fig. 3 (c): it is the powerset construct of both DB and I. On tree databases however, 1-indexes coincide with strong dataguides (because here I is deterministic).

4 2-Indexes

In this section we describe index structures for answering queries of the form $\mathsf{select}\ x_1, x_2\ \mathsf{from} * x_1\ P\ x_2$, with P a regular path expression: the template is $* x_1 \boxed{P}\ x_2$. We are interested in pairs of nodes (x_1, x_2), so we define:

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

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 expensive, so we consider (efficiently computable) refinements, $\approx$, satisfying:

$$(y, u) \approx (v', u') \Longrightarrow (v, u) \equiv (v', u') \tag{2}$$

As before, there exist efficient refinements based in simulation and bisimulation. (Details are omitted for lack of space). We define the 2-index $I^2(DB)$ of DB to be the following rooted graph. Its nodes are equivalence classes, $[(v, u)]$, of $\approx$; the roots are all the equivalence classes of the form $[(x, x)]$; finally, for each edge $u \stackrel{a}{\to} u'$ and each node v in DB there is an edge $[(v, u)] \stackrel{a}{\to} [(v, u')]$. We store I^2 as two logical components: the graph itself and the extent $\mathsf{extent}(s) \stackrel{\text{def}}{=} [(v, u)]$, for each node s representing the equivalence class $[(v, u)]$.

Proposition 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 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 $\mathsf{select}\ x, y\ \mathsf{from} * x\ P\ y$, we compute the query path $P\ y$ on I^2 and take the union of the extents. Note that this saves the $*$ search: but we may have to start at several roots in I^2. Often, these are only a few. For example, in acyclic databases, I^2 has a single root, because[5]

[5] This statement applies also to $\approx_b$ and $\approx_s$.

$(u, u) \equiv (v, v)$ for every nodes $u, v \in DB$. Figure 4 shows the 2-Index (without extents) for the database in Figure 3 (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.

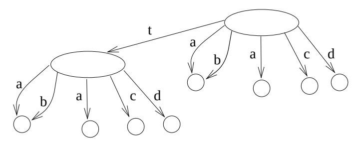

Fig. 4. A 2-index for the data graph

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 is no more expensive than querying the database, we would like to keep the index as small as possible. Our experiments (described briefly in the Appendix) indicate that in practice the index size is 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. On the theoretical side, Theorem 1 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.

Connection to Related Work: Patricia Trees – We conclude this section by mentioning 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*, ... 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. To a sequence of characters $a_1, a_2, \ldots$ associate a data graph DB with a single long chain: $v_1 \xrightarrow{a_1} v_2 \xrightarrow{a_1} \ldots$ Then the 2-index I^2 for DB is a tree, and the Pat tree can be obtained from I^2 by performing some simple optimizations: (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. These and other optimizations will be described in the full version of the paper.

5 T-Indexes

We now turn to general T-indexes. 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.

For the remainder of this section we fix a 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 that can be evaluated. We will discuss below how to test whether a given query can be evaluated using a T-index.

Definitions: We assume that the query binds the n variables $x_1, \ldots, x_n$ in that order. A partial binding will correspond then to an i-tuple $(v_1, \ldots, v_i)$ of nodes in DB, for some $i = 1, n$. For each such i-tuple we will consider the languages $L_{(v_{j-1},v_j)}$ for $j = 1, i$ defined in Sec. 4. (by convention $L_{(v_0,v_1)} \stackrel{\text{def}}{=} L_{v_1}$, defined in Sec. 3). We fix $n+1$ fresh symbols $\$, S_1, \ldots, S_n$ not occurring in $\mathcal{D}$. ($\mathcal{D}$ is the domain of data values from Definition 1.)

Definition 3. *Let $t = T_1\ x_1 \ldots T_n\ x_n$ be a path template and $(v_1, \ldots, v_i)$ an i-tuple of nodes in DB, $i = 1 \ldots n$. $T_{(v_1,\ldots,v_i)}(DB)$ is the language over the alphabet $\mathcal{D} \cup \{\$, S_1, \ldots, S_n\}$ generated by the regular expression $R_1.\$.R_2.\$ \ldots \$.R_i$, where the R_j's , $j = 1 \ldots i$ are the regular expression below:*

- *If $T_j = \boxed{P}$ (path), then $R_j \stackrel{\text{def}}{=} L_{(v_{j-1},v_j)}$.*
- *If $T_j = \boxed{F}$ (formula), then $R_j \stackrel{\text{def}}{=} L_{(v_{j-1},v_j)} \cap \mathcal{D}$.*
- *If $T_j = P_j$ (constant), if $L_{(v_{j-1},v_j)} \cap W(P_j) \neq \emptyset$ then $R_j \stackrel{\text{def}}{=} S_j$, otherwise $R_j \stackrel{\text{def}}{=} \emptyset$.*

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.

Computing $\equiv$ is expensive, so we consider refinements, $\approx$, satisfying:

$$(v_1, \ldots, v_i) \approx (u_1, \ldots, u_i) \Longrightarrow (v_1, \ldots, v_i) \equiv (u_1, \ldots, u_i) \tag{3}$$

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

Given $\approx$, the T-index $I^t(DB)$ for t is the following rooted, labeled 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 - For each i-tuple there is an edge labeled \$ from $[(v_1 \ldots v_{i-1}, v_i)]^\$$ to $[(v_1 \ldots v_{i-1}, v_i, v_i)]$, $1 \leq i < n$. 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 $T_i = P_i$, 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 $[(v_1, \ldots, v_n)]$.[6]

Example 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.

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

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 edge $s_1 \xrightarrow{S_1} s_1^\$$, continued with $s_1^\$ \xrightarrow{\$} [(u, u)]$, for $u \in s_1$, has edges $[(u, v)] \xrightarrow{a} [(u, v')]$ for edges $v \xrightarrow{a} v'$ in DB and $u \in s_1$, and finally has edges $[(u, v)] \xrightarrow{\epsilon} [(u, v)]^\$$, 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. $q = P_1\ x_1\ \ldots\ P_n\ x_n \in inst(t)$: we assume that each wildcard _ is replaced with (not $(\$) \wedge$ not $(S_1) \wedge \ldots \wedge$ not (S_n)), since our alphabet becomes now $\mathcal{D} \cup \{\$, S_1, \ldots, S_n\}$. First, let $P_q \stackrel{\text{def}}{=} P_1'.\$ \ldots \$.P_n'$, where:

$$P_i' \stackrel{\text{def}}{=} \begin{cases} S_i \text{ when } T_i \text{ is a constant} \\ P_i \text{ when } T_i \text{ is } \boxed{P} \text{ or } \boxed{F} \end{cases}$$

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 4. *(1) Let $\approx$ be a refinement 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 $W(P_q) \cap T_{(v_1,\ldots,v_n)}(DB) \neq \emptyset$.*

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 3. Let $t = \boxed{P}\ x\ ((B.A)*)\ y\ C\ z$ and $q = ((A.B)*).A\ y\ C\ z$. 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 4. Let $t = A\ x\ B\ y\ C\ z$ and $q = A\ x\ B\ y\ (C.D)\ u\ E\ v$. 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 "side branch" 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 general problem (for regular expressions it is still open) but restrict ourselves to prefix replacement.

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 $p \in inst(t)$ whose variables are renamed to include a prefix $x_1, \ldots, x_i$ of q's variables, in that order, and (2) a postfix q' of q containing the variables $x_i, \ldots, x_n$, such that the query select $(x_1, \ldots, x_n)$ from p, q' is equivalent to q. Note that the new query consists of a query paths which includes the variables $x_1, \ldots, x_n$, and a possible side branch (everything after x_i in p; see Example 4). Checking whether a query path q admits a prefix replacement is PSPACE-hard, by reduction to the equivalence problem for regular expressions (which is PSPACE-complete [22]): R, R' are equivalent iff the query path $q = R\ x$ has a prefix replacement w.r.t. the template $t = R'\ y$. In the full version of the paper we prove that one can check in PSPACE whether there exists a prefix replacement (and find one, when it exists).

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. Summarizing:

Proposition 5. *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 $(R1.tup)\ x\ A\ y\ \boxed{F}\ z$. 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 $u \xrightarrow{A_1} u_1 \xrightarrow{A_2} u_2 \ldots u_n$ 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 $[(r)] \rightarrow [(r)]^{\$} \rightarrow [(u, u_1)] \rightarrow [(u, u_1)]^{\$} \rightarrow [(u, u_1, u_2)] \rightarrow \ldots \rightarrow [(u, u_1, u_2, \ldots, u_n)]^{\$}$, 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 semi-structured 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: in our setting equivalence classes are distinguished by path queries, while in their setting they are distinguished by first order formulas. Hence the language equivalences are coarser than FO equivalences, and result 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 an optimization technique, presented in the full version of the paper, 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

Experiments: Recall that index storage consists of two parts: the graph and the extents. We argued that the graph size is critical for query performance, and we are currently conducting a series of experiments to asses its size. Some of the results are reported in Table 1. The Bibtex data is a relatively structured one, while the Web site (of the CS department of Tel-Aviv University) is far less structured. We also considered randomly generated graphs, and mixed graphs composed of both structured and unstructured components. 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.

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

Table 1. Experiments showing index size

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 1, 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.

The Web example is far less structured: many pages at the site are each built and maintained individually by distinct people without coordinating the structure. 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.