Indexing Web Form Constraints

Ronaldo dos Santos Mello[1], Ramesh Pinnamaneni[2], Juliana Freire[2]

[1] Universidade Federal de Santa Catarina, Brazil
ronaldo@inf.ufsc.br
[2] University of Utah, USA
{pinnaman,juliana}@cs.utah.edu

Abstract. Millions of online databases are available today on the Web that cover many different domains. These databases are accessible through forms and provide several useful services from searching for rental cars and airfares to used cars and genes. To leverage this information and locate online databases that are relevant for particular information needs, we have created a search engine that is specialized in forms that serve as the entry points to these databases. This search engine, however, only provides a keyword-based interface that greatly limits the kinds of queries that can be posed. In this paper, we study the problem of supporting structured queries over Web form collections. We formalize the problem of querying Web forms as satisfying constraints that hold between form attributes and their values, form metadata, as well as dependencies across distinct attributes. We also propose an indexing method that leverages the constraints to support efficient query processing. Because the proposed index extends traditional inverted indexes, our method can be easily combined with existing text searching tools. An experimental evaluation, where we compare query performance under the proposed index against different storage configurations using a relational database, shows that with our index, structured queries can be evaluated at a fraction of the time required by the relational database. We also show the higher precision of the result sets generated by our structured queries against traditional keyword-based queries over Web forms.

Categories and Subject Descriptors: H. Information Systems [**H.3 Information Storage and Retrieval**]: H.3.1 Content Analysis and Indexing—*Indexing methods*

Keywords: constraints, indexing, Web form collections

1. INTRODUCTION

A large volume of digital data available today is stored in the millions of online databases on the Web [Madhavan et al. 2009]. The content in these databases is referred to *the hidden* or *deep Web*, since they are hidden behind Web form interfaces and are not accessible to traditional Web crawlers.

Although the problem of finding and organizing the forms that serve as entry points to online databases has been received substantial attention [Barbosa et al. 2007], [Barbosa and Freire 2007a], [Barbosa and Freire 2007b], [Fang et al. 2007], [Madhavan et al. 2008], [Wang et al. 2009], the same cannot be said of query support for exploring form collections. In the context of *DeepPeep* [Freire et al. 2010], we have done preliminary work on querying form collections. DeepPeep is a search engine specialized in forms. The system, however, provides a very limited query interface which supports only keyword-based queries and selection of forms based on the online database domain (e.g., *Used Cars*, *Molecular Biology*, *Airfare*).

As the number of hidden databases grows, effective means for exploring them are necessary, in the same way we query traditional databases based on their known structure. To provide a structured query interface over a Web form collection is important to end users that want to locate databases

This research was funded by brazilian CAPES Foundation, U.S. National Science Foundation (grants IIS-0905385, IIS-0844546, IIS-0746500, CNS-0751152 and IIS-0713637), and the U.S. Department of Energy.

for a particular information need. In a *Used Cars* domain, for example, a user may want to find companies that sell a given car *make/model* in a specific *city*, or to retrieve companies with a certain average *price* for a given used car *model* in a certain *year*.

We model a Web form as a set of attributes that define *constraints* over the database contents, such as range or enumeration of allowed values, or even the possibility of filling out additional attributes that become visible after a certain value is set to another attribute. Again, in a *Used Cars* domain it is very common to find out forms that let the user selects the desired car manufacturer (*Toyota*, *Honda*, etc) through a *make* attribute, and, in turn, a *model* attribute has its values automatically restricted to valid car models for the chosen car make.

Web form constraints are useful in this context because they provide information about the hidden database, and, as a consequence, allow the specification of queries in a declarative way. If we knew, in a *Hotel* domain, that the location of some hotels is restricted to Brazilian State capitals and these hotels provide rooms for one to four people, we would probably get much more relevant hotel forms if we formulate a structured query based on these known constraints, like "*domain = 'hotel' and city = 'Florianópolis' and rooms = 3*" instead of a query with the keywords "*domain hotel city Florianópolis rooms 3*". Knowledge about constraints can be also useful for query optimization. On supposing no hotels exist in *Florianópolis*, the declarative query above would return the empty set without the need for checking the availability of rooms for three people.

In this paper, we study the problem of supporting structured queries over Web form collections, focusing on indexing structured query predicates based on form constraints. In particular, we consider queries over form metadata, allowed values for form attributes, and form attribute dependencies. In the first case, we filter out forms based on their properties, like domains and existing attributes. In the second case, we look for forms that hold specific attribute values, like *make = 'Honda'*. In the last case, we can check, for example, if a specific value dependency for two or more attributes exists in a form, like the need to know if a company sells *Corolla model* manufactured by the *Toyota make*. These three kinds of queries cover the existing constraints in a large sample of Web forms we had analyzed.

The main challenge in this context is how to provide an efficient storage and index structuring for relevant query predicates over Web forms. One alternative is to use a relational database. However, they provide poor support for keyword-based queries, and as we show in Section 4, they also have poor performance for structured queries. This is due to the great number of required table joins in order to match data about forms and their attributes, values and dependencies.

Instead, we propose an indexing structure that extends inverted indexes to support structured queries. Our approach considers a hierarchical representation of Web form properties (including attributes and constraints), and our index entries are path expressions that traverse such a structure, like path expression-based indexes for XML data [Brenes et al. 2008], [Chung et al. 2002], [Kaushik et al. 2004], [Milo and Suciu 1999], [Qun et al. 2003]. The benefits of this approach are that we can index structured query predicates as paths in a simple way, and we can re-use existing inverted index systems such as *Lucene* [PMC 2010] to provide efficient queries.

We also perform *query expansion* to retrieve Web forms with attributes that are similar to the attributes specified in the query predicates. This is possible by accessing a *synonym table* that relates each attribute to a canonic term that represent a cluster of similar attributes in different forms. Attribute clusters and the synonym table are generated by a schema matching process for Web forms, which it is out of the scope of this paper. See the work of [Nguyen 2008] for details about this process, which is based on related work [He and Chang 2003], [Pei et al. 2006], [Wu et al. 2006], being only its output used by our indexing module.

We evaluated the proposed index method against traditional B-tree indexes on a relational database, running queries between 2 and 45 times faster. Besides, our approach had scaled very well for an

increasing number of query predicates, being at most 13% slower in comparison to a 100% increasing in the processing times for the relational access.

The main contributions of this paper are:

—A classification for Web form constraints and related structured query types. This classification is the foundation for a structured query language for Web forms.
—A simple but efficient indexing schema for such constraint/query types.
—A set of experiments that shows that our indexing method outperforms traditional indexes for accessing Web form data stored in a relational database. We compare our method with relational indexes because there is no other work with the same focus to be considered, and also because we have a relational repository for Web forms and our intention is to verify the performance of structured query processing over it. We also show that our proposed classes of structured queries improve the precision of the result set if compared to keyword-based queries.

This paper is organized as follows. Section 2 provides a foundation for Web forms and their classes of constraints. Based on this, Section 3 presents our strategy for indexing Web form constraints. Section 4 describe some experiments that evidence the good performance of our indexes against relational indexes for the same data. Section 5 discusses some related work and Section 6 is dedicated to the final remarks and future work.

2. CONSTRAINTS AND WEB FORMS

In a database context, a *constraint* for a data Dt defines a set of rules that restricts Dt valid states [Codd 1980]. A Web Form (WF) contains several implicit and explicit constraints, including the online database domain and a set of valid values for an attribute. We define a WF from the point of view of its constraints as a tuple $wf = (id, wp, wpt, D, W, A, dep)$, where id is a unique identifier for wf; wp is the URL of the Web page where wf is located and wpt is the title of this page; D are the online database domains that wf belongs to; W is the bag of words in wf (i.e., all terms within the HTML form tag); A denotes the set of attributes in wf[1]; and the dependencies between attributes is denoted by dep[2].

$wf.D$ holds all possible WF domains, like a form for hotel booking, as shown in Figure 1. $wf.W$ maintains the words that provide some relevant information about the WF, like *Holidays* in Figure 1.

Fig. 1. Example of Web form in Hotel domain.

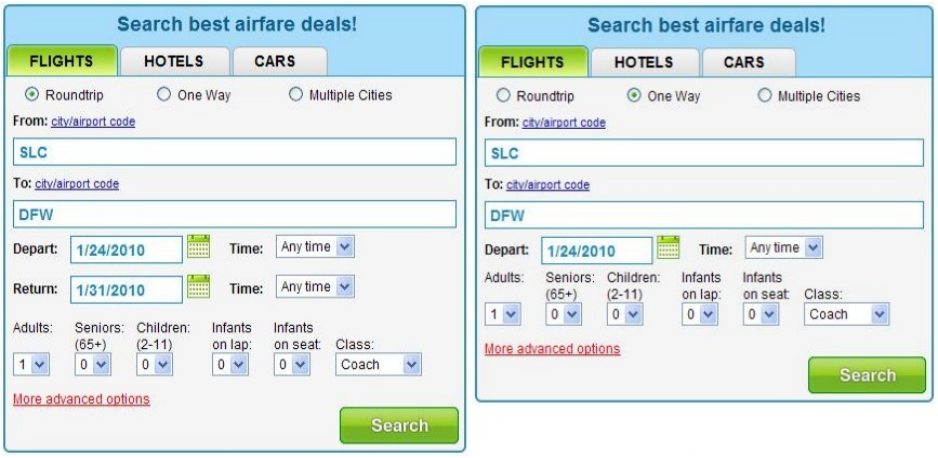
A WF wf also holds a set of attributes A on which the user specifies searching criteria for retrieving data from the hidden database. An attribute $a_i \in A$ is, in turn, a tuple $a_i = (l, V)$, where $a_i.l$ is

[1]We use the terms attribute, field and element interchangeably.
[2]The *dep* property will be further explained.

the attribute name (attribute label) and $a_i.V$ are the allowed values for a_i. In Figure 1, the user can search for available hotel rooms through several filter attributes that are restricted to a set of values, like *Region* and *Duration*.

Based on this WF definition, we specify the classes of constraints for WFs and, as a consequence, the classes of queries that can be formulated over a set of WFs to test these constraints. The classes of constraints are: *(i) context*; and *(ii) dependency*. *Context* constraints refer to the contextualized data in a WF, which can be classified, in turn, as *metadata* or *valid attribute value*. In the first case, we consider WF meta-information that belongs to our WF definition, i.e., the WF *page title*, the *URL* on which the WF is defined, an *attribute label*, an *attribute value*, a *domain name* or a WF *word*. Formally, we say that *metadata(wf)* = {PAGE_TITLE, URL, LABEL, VALUE, DOMAIN, WORD}[3]. The WF of Figure 1, for example, belongs to the *Hotel* domain and has *Country* as an existing attribute label. For querying purposes, WFs are retrieved using a set of specific filters. These filters are defined as equality or inequality predicates based on *metadata(wf)*, like "*DOMAIN = 'Hotel' and VALUE > 2010*".

In the second case, we consider the relationship between attribute labels and values, being a value contextualized as a restriction for an attribute identified by a label, i.e., given $a_i \in wf.A$, $a_i.l$ is constrained by $a_i.V$. *'Mexico'* in Figure 1 is a valid *Country* value. We can query WF data based on this kind of constraint through equality or inequality predicates, like "*Country = 'Mexico' and Duration < 3*".



(a) (b)

Fig. 2. Example of Web form in Airfare domain with *roundtrip* (a) and *one way* (b) selection.

Dependency constraints occur when an attribute value implies restrictions over the values of other attributes, or when an attribute value implies restrictions on the existence of other attributes. We categorize them as *value dependency* and *non-existence dependency*, respectively. Figure 2 (a) and (b) shows, respectively, for a WF in *Airfare* domain, that if the flight type attribute is set to *roundtrip* or *one way*, the *Return* attribute exists or not exists. Figure 1 exemplifies a value dependency between the attributes *Region* and *Country*: if *Region* value is *'North America'*, the allowed values for *Country* are *'Bermuda'*, *'Canada'*, etc.

[3]A WF usually have more than one label, value, domain and word. However, we define it as singular terms in *metadata(wf)* (not LABELS, for example) in order to standardize the naming of all metadata type to be used in index entries as well as in query predicates.

Class	Syntax	Example
Metadata	*metadata(wf)* [= | < | ≤ | > | ≥] *value*	LABEL = 'Country'
Attribute Value	*label* [= | < | ≤ | > | ≥] *value*	Country = 'Brazil'
Value Dependency	$V.predicate_1$ *AND ... AND* $V.predicate_n$	V.Country = 'Brazil' AND V.State = 'Santa Catarina'
Non-Existence Dependency	$V.predicate_1$ *AND ... AND* $V.predicate_n$	V.Type = 'one-way' AND V.Return = '06/30/10'

Table I. Constraint-based query classes for Web forms.

Dependency constraints are represented by the *wf.dep* property, which is defined as a tuple *dep = (VD, NED)*, being *VD* and *NED* the sets of value and non-existence dependencies, respectively. Formally, given a_i, $a_j \in wf.A$ and $v_x \in a_i.V$, $f \in wf.dep.VD$ is defined as $f = (a_i.l$ [= | < | ≤ | > | ≥] $v_x \Rightarrow a_j.V \subseteq \{v_1, ..., v_n\})$, with $\{v_1, ..., v_n\} \subseteq a_j.V$, and $ne \in wf.dep.NED$ is defined as $ne = (a_i.l$ [= | < | ≤ | > | ≥] $v_x \Rightarrow \neg\exists\ a_j)$.

We can pose a query to retrieve WFs with respect to a value dependency. It is represented by a query condition that defines at least two conjunctive predicates with the form $v.predicate_1$ *and ...* *and* $v.predicate_n$, where v is a variable that denotes the intention to test a dependency between the involved query predicates[4]. For example, the query *v.Region = 'North America' and v.Country = 'Canada'* retrieves WFs that hold a dependency between the specified values of the attributes *Region* and *Country*. The same query notation is applied to test non-existence dependencies. For example, on processing the query *v.Type = 'one-way' and v.Return = '6/30/10'*, we can verify *a priori* if there are WFs whose existence of the attribute *Return* is dependent of the restriction *Type = 'one-way'* for the attribute *Type*. On considering this dependency true for a set of WFs S_f (*Return* attribute does not exist if *Type* attribute has *'one-way'* as value), we can eliminate S_f early from the result set.

Table I summarizes the considered constrained-based query classes for WFs with the proposed syntax and examples in *Airfare* domain. This work focuses on providing an efficient indexing schema for all of the query classes. Next section details the proposed indexes.

3. INDEXES FOR WEB FORM CONSTRAINTS

The foundation of our indexing method is a hierarchical organization for the WF instance properties, in a similar way well-structured XML data[5] are represented. A complex structured XML instance has a complex element E_i as root and a bag of sub-elements $B_i = \{e_1, e_2, ..., e_n\}$ that describes E_i properties. A sub-element can be, in turn, a complex element composed by other elements, or a simple element that holds a text value. A traditional way to pose a query on XML instances is to specify a *path expression*, i.e., a traversal on the instance hierarchical structure from the root to the desired element. For example, a path $E_i/e_1/e_1i.../e_1m$ retrieves the content of the element e_1m that is reached from E_i following the chain of sub-elements described by the path.

Figure 3 shows the hierarchical structuring for the WF of Figure 1 that is similar to the representation of complex XML instances[6]. Root node denotes the WF ID and each descendant node represents a property and its type. Because we consider that each property is a direct feature of the WF instance, all non-root nodes are connected to the root node. A deeper hierarchical chain occurs when we have dependencies between WF attributes, or attributes restricted to a set of values. In Figure 1, we can only visualize and select *Country* allowed values after we select a *Region* value. This dependency

[4]Details about the syntax and semantics of a structured query language for WFs are out of the scope of this paper. This subject is the focus of a forthcoming work.
[5]http://www.w3c.org/xml.
[6]We said "similar" hierarchical structuring because a node may have more than one parent node. This is necessary to define all possible paths to be indexed by our indexing structures.

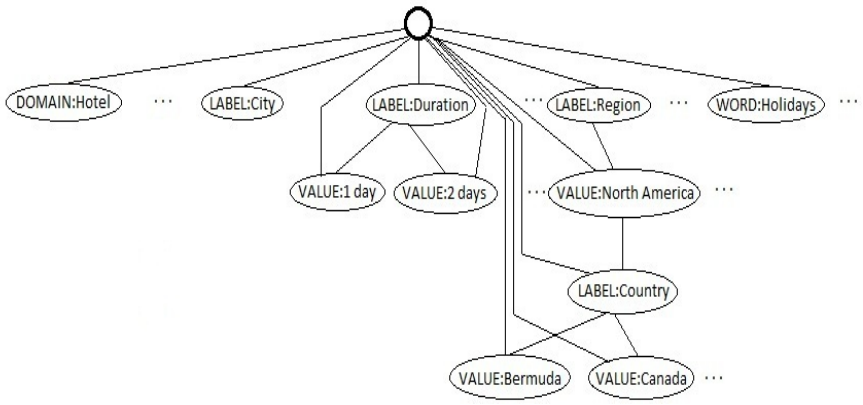


Fig. 3. Example of hierarchical structuring of Web form data.

is exemplified in Figure 3 for the *North America* value of the *Region* attribute, being *Bermuda* and *Canada* values of *Country* positioned as its descendant nodes because they are valid *North America* countries. Some allowed values of the *Duration* attribute are also shown.

	F_1	F_2	...	F_n
...			...	
2000/VALUE	1	0	...	0
...			...	
City/LABEL	0	1	...	1
...			...	
Hotel/DOMAIN	1	1	...	1
			...	

(a)

	F_1	F_2	...	F_n
...			...	
2000/Price	1	0	...	0
2000/Year	1	0	...	1
...			...	
Canada/Country	0	1	...	1
...			...	
Miami/City	0	1	...	1
...			...	

(b)

	F_1	F_2	...	F_n
...			...	
Bermuda/Country/North America/Region	0	1	...	0
Bolivia/Country/South America/Region	0	1	...	0
...			...	
/return/one way/flight type	1	0	...	0
...			...	
corolla/Model/toyota/Make	0	0	...	1
...			...	

(c)

Fig. 4. Index structures: Metadata index (a); Attribute value index (b); Dependency index (c).

We propose three index structures based on a path expression access to the hierarchical representation of a WF. Our indexes are simple extensions of the inverted list index traditionally used by the Information Retrieval area [Baeza-Yates and Ribeiro-Neto 1999], being able to support the indexing of all classes of constraints defined in Section 2. Basically, we allow path expressions as index entries in an inverted list entry instead of keywords, while the index columns keep the WF IDs (F_1, F_2, ..., F_n). Specific types of path expressions are provided for each one of our indexes:

—*Metadata context*: a path expression with the form *keyword/metadata_type*, which is able to index query predicates that test metadata context constraints. The *keyword* component refers to any desired meta-information in a WF and *metadata_type* $\in$ *metadata(wf)*. Figure 4 (a) shows an example of this index (visualized as an incidence matrix);

—*Attribute value context*: a path expression with the form *value/label*, which is able to index query predicates that test valid attribute values context constraints. The *value* and *label* components refer to a valid attribute value in a WF. Figure 4 (b) shows an example of this index;

—*Dependency*: a path expression with the form $value_1/label_1/value_2/label_2$, which is able to index query predicates that test dependency constraints. This index entry means that $label_2$ attribute with a value $value_2$ restricts $label_1$ attribute to $value_1$ as one of its possible values. Figure 4 (c)

shows an example of this index. When specified in a path, the special character '*' denotes a non-existence dependency between two attributes with the form $*/label_i/value_j/label_j$, meaning that $label_j$ attribute with a value $value_j$ forbids the existence of $label_i$ attribute.

Observe that the general format of our path expressions is *node_ value/node_ context* instead of the format *node_ context:node_ value* in the nodes of a WF hierarchical representation. We give priority to the description of values in the index entry because they allow a lexicographic order that provides efficient index lookup for searching a same value in different contexts.

We still highlight two relevant points related to the design of the *Dependency* index. First, real WFs can specify a chain of value dependencies between *label/value* pairs. One example is the WF of Figure 1 which holds a three level dependency chain involving the attributes *Region*, *Country* and *State/Province*, in that order. For this example, a valid value dependency is *Region = 'North America'* ⇒ *Country = 'United States'* ⇒ *State/Province = 'Utah'*. Thus, an index with path expressions composed by an arbitrary number of *label/value* pairs as entries would be expected. We identify three major problems regarding this approach: *(i)* index entries with high redundancy because determinant *label/value* pairs in the first levels of the dependency chain repeat for each possible valid determined pair (or chain of pairs) in the lower level(s); *(ii)* large index entries; and *(iii)* we cannot match indirect value dependencies in the index because we may have an arbitrary number of *label/value* pairs in between. Because of this, we decided to maintain index entries with only two *label/value* pairs holding a valid direct or indirect value dependency. We still have redundancy, but it is less than the previous approach, and we are able to efficiently check direct and indirect dependencies with smaller index entries. In practice, we do not have a large number of WFs with dependency constraints, and the number of dependency chains with more than three levels is reduced.

Another point is related to the choice for indexing a non-existence dependency instead of an existence one. The issue here is that non-existence constraints are the exceptions and, as a consequence, require less index entries than true existences. Besides, as exemplified in Section 2, if we check early a query condition for non-existence dependency, we can discard a set of WFs from the result set, or even to conclude that the result set is empty.

3.1 Indexing Module Overview

We had developed an indexing module in order to implement and validate our indexing method. It basically accomplishes three steps:

(1) *Query parsing.*
(2) *Index access.*
(3) *Generation of the result set.*

The first step is responsible to receive and to validate the input query. The input query is a set of conjunctive and/or disjunctive predicates that follows the syntax presented in Table I. If the input query is correctly validated, each one of its query predicates is converted to a path expression that fits an index entry and the modified input query can now be processed by the second step.

The *Index Access* step is the core of the indexing module, being responsible to analyze the input query, to set the ordering that the indexes are accessed, and to provide some particular processing. Despite of following the traditional precedence order to process a large query filter (conjunctive predicates have precedence over disjunctive predicates), it gives some priorities to the processing of a set of conjunctive predicates, as follows:

(1) *non-existence dependency checking.*
(2) *value dependency checking.*

(3) *metadata and attribute/value checking.*

This ordering is relevant because dependency constraint filters are more selective than, for example, an attribute/value filter given that a small number of WFs holds dependencies. In case of a dependency checking fails, it is not necessary to check the other predicates. Thus, these priorities define the ordering the respective indexes are accessed. This step still accomplishes some specific processing that are relevant in the context of querying WFs. These processing are detailed as follows.

A conjunctive filter for attribute values like "*Make = 'Honda' AND Make = 'Ford'*" is supposed to return the empty set in a traditional database query because an attribute cannot hold two different values in a tuple. However, this query in our context means we want to retrieve WFs that hold these two constraints. Thus, an intersection of the result sets for each predicate must be performed. The same reasoning holds for a query over metadata constraints like "*LABEL = 'Make' AND LABEL = 'Year'*". Other relevant processing issue is related to dependency queries. Given, for example, a query "*f.Year = 2008 AND f.Make = 'Toyota'*", we may have two possible value dependencies in the WFs: *(i) Year = 2008 → Make = 'Toyota'*, or *(ii) Make = 'Toyota' → Year = 2008*. Thus, we have to access these two possible entries in the dependency index and perform an union of the result sets. The more predicates a dependency filter has, the more combinations must be tested.

Finally, the last step produces the result set, which is composed by the WF ID and the WF URL. These two properties are available in the index structure and be directly retrieved from the index entry.

4. EXPERIMENTAL EVALUATION

We base our experiments on a sample of real WFs. Our database research group had developed tools that are able to crawl forms on the Web [Barbosa and Freire 2007a] and to extract information about their location, attributes and constraints, as well as to store these information in a relational repository (see *RelDB* repository in the following). The extraction strategy is centered on a learning classifier that identifies attributes and their properties based on form layout patterns that are learned basically from HTML markup. The approach is able to deal with dynamic forms by simulating, using client-side scripting, the actions a user would perform while selecting different values. Such a capability allows the discovery of attribute and value dependencies. See [Nguyen et al. 2008] for details, which includes experiments that confirm the high extractor accuracy.

The set of experiments aims at validating the efficiency of our indexing methods over traditional relational indexes created in the form repository. The next sections detail the experimental setup and results.

4.1 Experimental Setup

Our data sample is composed of 3,218 WFs on 8 application domains with 18,143 attributes and 287,518 allowed values constraints, as well as 88,017 value dependencies between values[7], with an approximate size of 118 Mb. We ran three kinds of experiments that return WF IDs and URLs:

—*RelDB*: SQL queries executed over a MySQL 5.1 relational repository with B-tree indexes for keys and relevant columns to be filtered. The index size for this repository is 23 Mb.

—*RelViewDB*: SQL queries executed over materialized views of the relational repository. These views organize data about WF constraints in a reduced number of tables that avoids (or minimizes) joins between tables. Relevant columns are also indexed here through B-trees. Index size in this case is 105 Mb.

[7]Only value dependencies were extracted from the data sample.

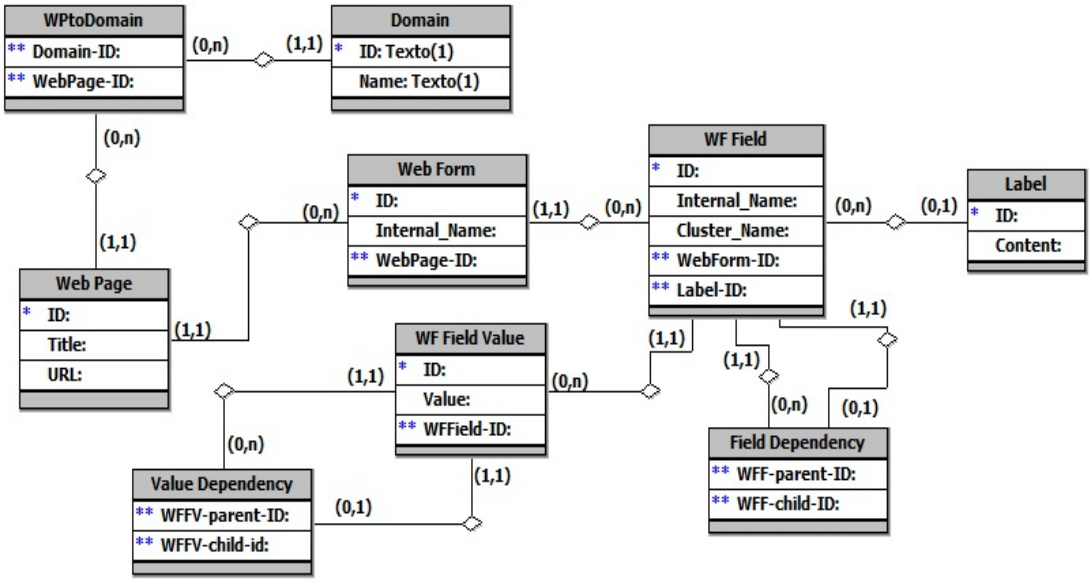


Fig. 5. *RelDB* schema.

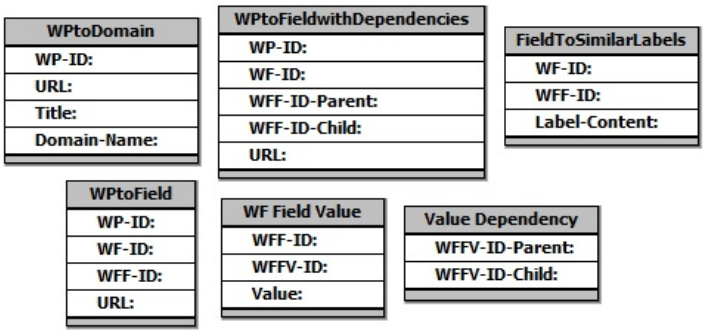


Fig. 6. *RelViewDB* schema.

—*ConstrIL*: queries executed over our WF indexing module. Index size in this case is 57 Mb.

Figure 5 and 6 show the *RelDB* and *RelViewDB* schemata, respectively. Only the main tables and their most relevant columns are shown. *RelDB* schema is centered on the *Web Form* table. From a WF tuple, it is possible to obtain information about its Web page and the domains it belongs, as well as the attributes (fields) that compose it, with their labels and values. Additional tables catalog the attribute dependencies and the attribute values that constraint other attribute values. *WF Field* table holds a *Cluster_Name* column. All synonym fields have the same value for this column. *RelViewDB* schema joins some of the *RelDB* tables and preserves the most important columns for query purposes. The strategy here was to reduce, as most as possible, *RelDB* table joins and, at the same time, to avoid the generation of very large merged tables that could compromise query processing performance. As shown in Figure 6, we obtained a good balance by merging: *(i)* Web pages and their domains; *(ii)* Web pages with their WFs and attributes, and; *(iii)* WFs attributes and their related synonyms. A version of case *(ii)* (*WPtoFieldwithDependencies* table) keeps information about attribute dependencies, being used for dependency queries. Note that *WF Field Value* and *Value Dependency* tables were not merged because they are very large tables.

We implemented the indexing module using the *Lucene* indexing tool [PMC 2010], which supports inverted lists, and coded all necessary algorithms in Java. The experiments were conducted on a machine with 2.20 GHz and 4 GB memory.

We executed seven queries with filters over metadata and attribute value constraints (we call them *Query 1* to *Query 7* - see Section 4.2 for more details): a simple one (with a single predicate), 3 disjunctive queries with 2, 5 and 10 predicates, and 3 conjunctive queries with the same incremental number of predicates. We also ran queries that return WFs with value dependencies among 2 and 3 attributes[8] (we call them *Query 1* and *Query 2*). Query conditions were defined over the most frequent WF attributes, which generated a higher data volume in the result sets.

In order to show an example of input query over each data repository, the expressions in the following show a same query formulated over *RelDB*, *RelViewDB* and *ConstrIL*, respectively. The intention here is to retrieve WFs (in any domain) that holds cars manufactured by *Ford* and cars produced in *2005*. In the first two SQL queries it is possible to see the need for accessing and joining several relational tables, even for *RelViewDB*, including the fetching of WFs with similar attributes (see the condition on *Cluster_Name* column in *RelDB* query and the access to *FieldToSimilarLabels* table in *RelViewDB* query). In the *ConstrIL* case, instead, we have a more simple syntax and a less complex query processing.

```
SELECT *  /******* RelDB query *******/
FROM
(SELECT DISTINCT wp.URL, wf.ID
 FROM WebPage wp JOIN WebForm wf ON wp.ID = wf.webPage-ID
 JOIN WFField wff ON wf.ID = wff.webForm-ID
 JOIN WFFieldValue wffv ON wff.id = wffv.WFField-ID
 WHERE wffv.Value = 'Ford'
 AND wff.Cluster_Name IN (SELECT DISTINCT wff.Cluster_Name
                          FROM Label l
                          JOIN WFField wff on wff.Label-ID = l.ID
                          WHERE l.Content = 'Make')) AS Table_1
NATURAL JOIN
(SELECT DISTINCT wp.URL, wf.ID
 FROM WebPage wp JOIN WebForm wf ON wp.ID = wf.webPage-ID
 JOIN WFField wff ON wf.ID = wff.webForm-ID
 JOIN WFFieldValue wffv ON wff.id = wffv.WFField-ID
 WHERE wffv.Value = '2005'
 AND wff.Cluster_Name IN (SELECT DISTINCT wff.Cluster_Name
                          FROM Label l
                          JOIN WFField wff on wff.Label-ID = l.ID
                          WHERE l.Content = 'Year')) AS Table_2

SELECT * /******* RelViewDB query *******/
FROM
(SELECT DISTINCT wp2f.URL, wp2f.WF-ID
 FROM WPToField wp2f
 JOIN WFFieldValue wffv ON wp2f.WFF-ID = wffv.WFF-ID
 JOIN FieldToSimilarLabels f2sl ON wp2f.WFF-ID = f2sl.WFF-ID
 WHERE wffv.Value = 'Ford'
 AND f2sl.Label-Content = 'Make') AS Table_1
NATURAL JOIN
(SELECT DISTINCT wp2f.URL, wp2f.WF-ID
 FROM WPToField wp2f
 JOIN WFFieldValue wffv ON wp2f.WFF-ID = wffv.WFF-ID
 JOIN FieldToSimilarLabels f2sl ON wp2f.WFF-ID = f2sl.WFF-ID
 WHERE wffv.Value = '2005'
 AND f2sl.Label-Content = 'Year') AS Table_2
```

[8]At most 3 attributes were involved in value dependencies in our data sample.

```
Make='Ford' AND Year=2005;  /******* ConstrIL query *******/
```

4.2 Experimental Results

4.2.1 *Processing Time Analysis.* Figures 7, 8 and 9 present the results of our experiments, which measure the processing times for the proposed set of queries, as described in Section 4.1. The measured times consider query expansion execution, i.e., the time spent to execute the query for each found attribute synonym, besides the given attribute in the input query. For *ConstrIL*, this time considers the access to the synonym table for each described attribute in the input query as well as query rewriting and run for each attribute synonym[9].

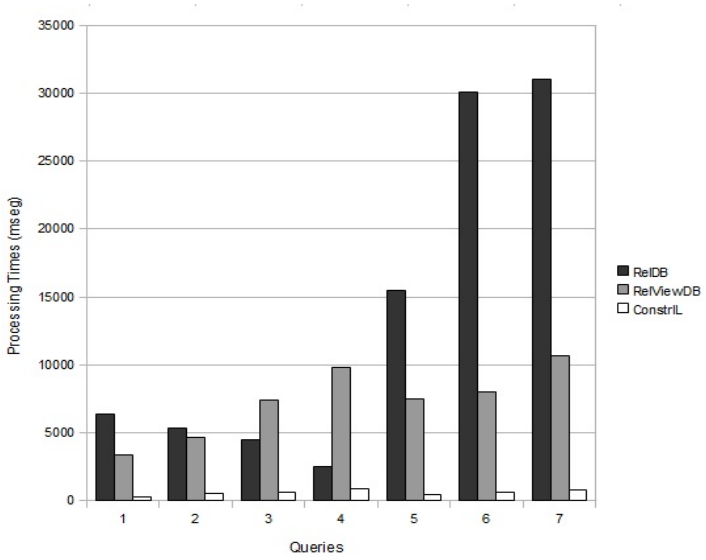


Fig. 7. Processing times for queries over *Metadata* constraints.

According to Figures 7, 8 and 9, our indexing method obtained a very good performance if compared to the relational indexes for accessing the same data. This is particularly true for queries that filter *metadata* and *dependency* constraints. In the first case, *ConstrIL* ran approximately 10 times faster than *RelViewDB* for all types of queries. When *RelDB* is considered, we were around 20 times faster for single predicates, and maintained a range from 3 to 8 and from 35 to 40 times faster for disjunctive and conjunctive queries, respectively. Our processing times for 10 disjunctive and conjunctive predicates (queries 4 and 7) were 877/794 ms against 2.5/31 sec for *RelDB* and 9.8/10.6 sec for *RelViewDB*, respectively. We also obtained excellent results when querying dependency constraints, being around 45 times faster than *RelDB*. We spent 375 ms to process query 2 against 19.8 sec for *RelDB* and 2.3 min for *RelViewDB*.

We observe that *RelViewDB* performance was poor than *RelDB* in some cases. It happened because the materialized views generated large tables after the join of the relevant data. As a consequence, the sizes of the index structures for these tables were also large.

Attribute value index is our largest index structure due to the great number of allowed values for WFs attributes. Besides that, we outperform the relational indexes in all types of queries, ranging

[9]It is important to note that Figures 7, 8 and 9 do not detail the fraction of the query processing time spent with query expansion because our indexing module returns only the total amount of time spent to process a query.

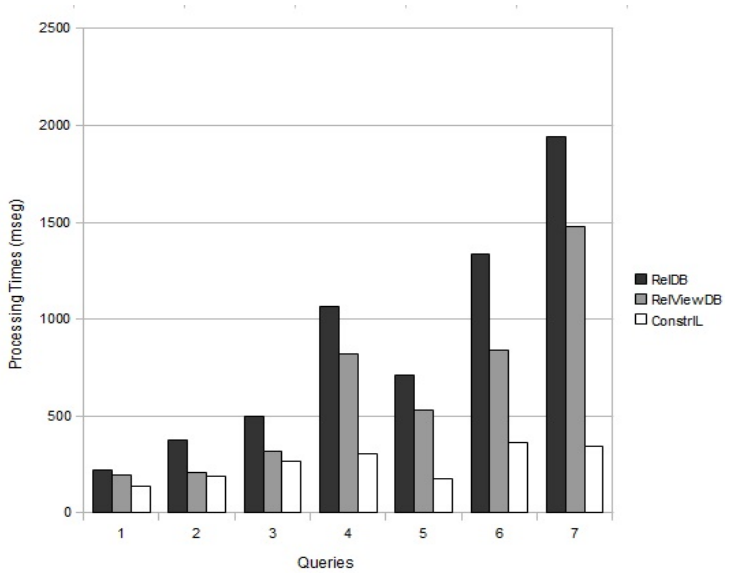


Fig. 8. Processing times for queries over *Attribute Value* constraints.

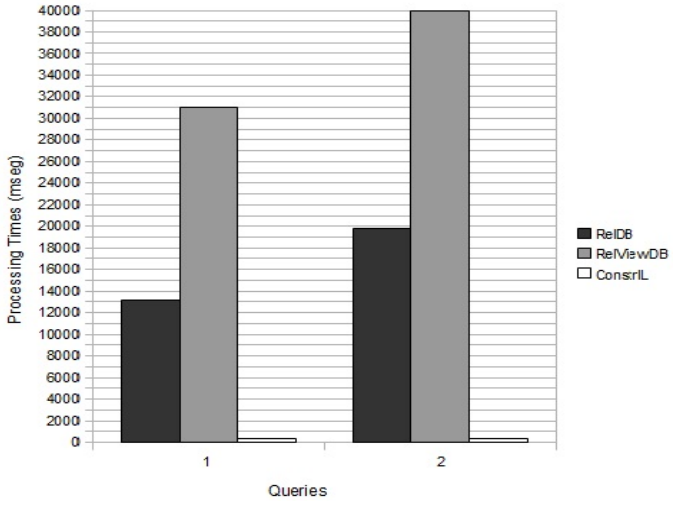


Fig. 9. Processing times for queries over *Dependency* constraints.

from 2 to 4 times faster than *RelViewDB*. We spent 303/345 ms for 10 disjunctive and conjunctive predicates (queries 4 and 7) against 1/1.9 sec for *RelDB* and 0.82/1.4 sec for *RelViewDB*, respectively.

We also note that our indexes scale very well given an increasing number of disjunctive or conjunctive predicates. In fact, processing times remain basically the same when querying *dependency* constraints, and had an increasing of only 170 ms for queries varying from one to 10 disjunctive or conjunctive query predicates over *attribute value* constraints. *RelViewDB*, instead, had reached more than 1.28 sec with respect to processing times increasing. For queries over *metadata* constraints, *ConstrIL* got an average increasing of 450 ms for disjunctive and conjunctive predicates against an increasing of 6.4 sec and 24.7 sec for *RelViewDB* and *RelDB*, respectively.

We observed decreasing processing times in the relational repository only for *RelDB* disjunctive

Query Class	Structured Query	Query Expansion
Metadata	24%	80%
Attribute Value	33%	89%
Dependency	95%	89%

Table II. Quality improvement with structured queries and query expansion.

queries over *metadata* constraints. This exceptional behavior had happened because the result set remains basically the same for all queries, being kept in the database cache.

4.2.2 *Quality Analysis.* Our approach also introduces important improvements in the quality of the query results if compared to traditional exact keyword-based queries provided by search engines for WFs. These quality improvements are related to the following issues:

—*Structured query*: knowledge about several classes of WF constraints makes possible the definition of structured queries with a more accurate semantic intention, if compared to keyword-based queries.
—*Query expansion*: the consideration of WFs with similar attributes during the processing of query predicates improves the recall if compared to exact queries.

Table II presents the percentage of quality improvement for these issues for each WF query class. We had analyzed queries over the more frequent constraints in our WFs data sample, showing the highest percentage of improvement in query results.

All of the query results we obtained from the experiments described in Section 4.2.1 had a 100% precision, i.e., all the returned WFs match the query predicates. Therefore, we assume the following interpretation for the percentages given in Table II:

—*Structured Query*: the purpose here is to analyze the improvement of a structured query result set over the result set of a traditional keyword-based query. Thus, we run a keyword query with the same terms of a given structured query, i.e., for a structured query *"Make = 'Toyota'"*, for example, we run a keyword query *"Make AND Toyota"*. The percentage is given by *100% - (#resultSet(structured query) / #resultSet(keyword query)) * 100*, which measures the fraction of irrelevant WFs that were put away by the structured query.
—*Query Expansion*: we analyze here the improvement in terms of additional WFs that are retrieved by applying query expansion against the result set of the same structured query without considering query expansion. We quantify here the percentage of additional correct WFs based on the number of WFs retrieved without query expansion.

Table II reveals very good query result improvements, specially when query expansion is performed. This is particularly true for *attribute value* and *dependency* constraints, which define predicates based on attributes, which can hold a lot of synonyms. For the *attribute value* case, for example, we had reached the highest quality improvement with the filter *"Year = 2005"*, which results in 241 WFs on running query expansion against 26 WFs without query expansion (an improvement of 89%). It happens because the attribute *Year* has a lot of synonyms in different WFs, like *Year:* and *Copyright year*.

With respect to the quality improvement of structured queries, we had obtained better results for query classes that impose more strict semantic constraints, which is less probable to match a keyword filter. This is also the case of *attribute value* and *dependency* constraints. We got excellent results for dependency queries because they impose a complex constraint. The highest quality improvement comes with the filter *"V.Year = 2005 AND V.Make = 'Ford'"*, which had returned 3 WFs against 57 WFs for the equivalent keyword query *"Year AND 2005 AND Make AND Ford"* (an improvement of 95% in the result precision).

5. RELATED WORK

For the best of our knowledge, there is no other work that focuses on the definition of classes of WF constraints and related structured query classes, as well as their indexing. Several work propose indexes based on path-expressions that are similar to ours, like indexes for XML [Brenes et al. 2008], [Chung et al. 2002], [Kaushik et al. 2004], [Milo and Suciu 1999], [Qun et al. 2003] or dataspace data [Dong and Halevy 2007]. If compared to indexes for XML, an indexed path expression has a different meaning of our approach, i.e., path expressions describe relevant relationships between data that are useful to query related XML elements. Instead, a path expression in our work relates a WF property with its constraints, i.e., it denotes a restriction on their contextualized values or on attribute dependencies, being useful for queries that filter data based on constraints. Analogously, long path expressions for dataspace data denote relationships between complex objects, having also a different semantics.

A drawback of approaches for XML indexing is that they index only the most frequent accessed paths to avoid large index structures. In some cases, an adaptive index is supported, i.e., an index that evolves based on changes in the frequent searched paths. Despite of that, if the user formulates a query whose XML path access does not belong to the frequent access paths, this query will be slow because the path is not indexed. This strategy is not applicable to WFs because the scope of queries over Web data is very large, i.e., the user intention when posing a query over Web data is very flexible, being not relevant to identify the most frequent accessed data. Instead, we gave focus to WF schema matching and the definition of WF attribute synonyms in order to increase the number of relevant forms to be retrieved, which is much more interesting when querying heterogeneous Web data sources.

We identify two related work close to our approach. In [Qiu et al. 2003], hidden database data (not data constraints) are extracted and stored in a relational database. They create a table for each WF attribute and each table basically maintains an indexed column for the values, and an additional column for the WF list or a cluster that groups similiar WF lists for different (but similar) values. We identify the following drawbacks in this proposal: *(i)* A great number of indexes must be created if we want to index values of several attributes; *(ii)* They assume that a schema matching for WFs in a same domain was previously accomplished to generate a relational global schema using a mediation system. This support increases the complexity of the approach; *(iii)* They are able to answer only queries over attribute values. Our approach has a broader scope, being able to query several classes of constraints. Besides these drawbacks, their experiments focus on the efficiency of the indexes compression as well as on the accuracy for avoiding the retrieval of false positives. Our experiments, on the other hand, focus on efficiency in terms of processing time for accessing indexes and answering user queries.

The work of [Dong and Halevy 2007] addresses the more general problem of indexing dataspaces and support only attribute-value queries through similar path-expression-based indexes. Our approach, while also indexes attribute/value pairs, gives additional index support to query metadata context and attribute dependencies. We argue that our approach can be potentially useful to dataspaces, for data that is similar to forms. Besides that, as pointed out before, the semantics of our queries is different because we query WF data based on their specific constraints, and not dataspace complex objects based solely on their attributes' values. Our different semantics require specific processing in order to access the indexes, as described in Section 3.1.

6. CONCLUSIONS

This paper presents an indexing method for WF constraints. By WF constraints we mean restrictions about WF properties available at form interfaces. The motivation for this work is the fact that such constraints provide semantic information about the database hidden behind the WF, which can be useful to filter relevant WFs through structured queries, i.e., declarative predicates based on WF property constraints. No related work focuses on this problematic.

Our indexing solution extends traditional inverted list indexes by allowing path expressions as index entries. A path expression specifies a valid predicate over a WF property according to the WF constraints, providing a direct translation of a structured query predicate to an index entry. Our experiments revealed that this simple approach is efficient and scalable.

Despite of the focus on indexing constraints defined on a WF, we also decided to implement a traditional *keyword* inverted list index. The purpose here was to index any term, like a page title, a label, an allowed value or a word, in the form interface. With this additional index we can mix keyword and structured filters in a query language for WFs, being able, for example, to retrieve WFs that have a given term T, where T could match a label or word, for example. Experiments similar to the ones described in Section 4 had shown that our keyword-based queries run from 8 to 37 times faster than *RelDB*, and from 2 to 31 times faster than *RelViewDB*. We also outperform the relational repositories for this kind of query because a keyword can be related to any WF metadata, requiring joins between most of the tables (Web pages, domains, fields, etc) in the schema.

Our indexing method intends to be the basis of a framework for structured queries over WFs. The main idea here is to design a structured query processing infra-structure for WFs that could support not only simple equality or inequality predicates, but also complex queries that provide data aggregation or data grouping, in the same way SQL query language works. With such a support, we will be able to answer a query like the one exemplified in Section 1 "*to retrieve companies with a certain average* `price` *for a given used car* `model` *in a certain* `year`". In order to process complex queries like this, we have some challenges to deal with, like the need to access data on the hidden database to generate the desired aggregated information. We plan to develop and integrate systematically this infra-structure to *Deeppeep* [Freire et al. 2010], [Barbosa et al. 2010] in order to improve its query capabilities.

Besides the framework initiative, other future work include:

—An index strategy for inequality constraints, like 1,000 <= *Price* < 50,000 and *Year* >= 2000. Inverted lists are not suitable for inequality predicates due to the potential overhead with index scan, so we are investigating other approaches, like hierarchical indexes (Tries [Brenes et al. 2008], [Comer 1979], [Terrovitis et al. 2006] or indexes for XML), that could be extended to organize value range constraints. We intend to provide such a solution as soon as possible, given the high relevance of this kind of constraint for our framework.

—Experiments on a larger number of attributes involved in value dependencies in order to analyze the efficiency of query processing and index accessing for dependency constraints. The intention here is to work on a larger set of real WFs or to produce some synthetic data, including WFs that hold non-existence dependencies, which were not considered in the set of experiments described in this paper.

—Efficient mechanisms for index update and compression, given the increasing availability of hidden databases.

—A query-by-similarity support. We mean here to allow searching for similar WFs based on affinities among their properties and constraints. An index for similarity-based filters could be considered.

REFERENCES

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

Barbosa, L. and Freire, J. An Adaptive Crawler for Locating Hidden Web Entry Points. In *Proceedings of the 16th International Conference on World Wide Web*. Banff, AL, Canada, pp. 441–450, 2007a.

Barbosa, L. and Freire, J. Combining Classifiers to Identify Online Databases. In *Proceedings of the 16th International Conference on World Wide Web*. Banff, AL, Canada, pp. 431–440, 2007b.

Barbosa, L., Freire, J., and da Silva, A. S. Organizing Hidden-Web Databases by Clustering Visible Web Documents. In *Proceedings of the 26th International Conference on Data Engineering*. Istanbul, Turkey, pp. 326–335, 2007.

Barbosa, L., Nguyen, H., Nguyen, T., Pinnamaneni, R., and Freire, J. Creating and Exploring Web Form Repositories. In *Proceedings of the ACM International Conference on Management of Data*. Indianapolis, IN, USA, pp. 1175–1178, 2010.

Brenes, S., Wu, Y., Gucht, D. V., and Cruz, P. S. Trie Indexes for Efficient XML Query Evaluation. In *Proceedings of the International Workshop on Web and Databases*. Vancouver, BC, Canada, 2008.

Chung, C.-W., Min, J.-K., and Shim, K. APEX: An Adaptive Path Index for XML Data. In *Proceedings of the ACM International Conference on Management of Data*. Madison, WI, USA, pp. 121–132, 2002.

Codd, E. F. Data Models in Database Management. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling*. Pingree Park, CO, USA, pp. 112–114, 1980.

Comer, D. Heuristics for Trie Index Minimization. *ACM Transactions on Database Systems* 4 (3): 383–395, 1979.

Dong, X. and Halevy, A. Y. Indexing Dataspaces. In *Proceedings of the ACM International Conference on Management of Data*. Beijing, China, pp. 43–54, 2007.

Fang, W., Cui, Z., and Zhao, P. Ontology-Based Focused Crawling of Deep Web Sources. In *Proceedings of the Knowledge Science, Engineering and Management, Second International Conference*. Melbourne, Australia, pp. 514–519, 2007.

Freire, J., Barbosa, L., Nguyen, H., Nguyen, T., and Pinnamaneni, R. DeepPeep: Discover the Hidden Web. http://www.deeppeep.org/, 2010.

He, B. and Chang, K. C.-C. Statistical Schema Matching across Web Query Interfaces. In *Proceedings of the ACM International Conference on Management of Data*. San Diego, CA, USA, pp. 217–228, 2003.

Kaushik, R., Krishnamurthy, R., Naughton, J. F., and Ramakrishnan, R. On the Integration of Structure Indexes and Inverted Lists. In *Proceedings of the ACM International Conference on Management of Data*. Paris, France, pp. 779–790, 2004.

Madhavan, J., Afanasiev, L., Antova, L., and Halevy, A. Y. Harnessing the Deep Web: Present and Future. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*. Asilomar, CA, USA, 2009.

Madhavan, J., Ko, D., Kot, L., Ganapathy, V., Rasmussen, A., and Halevy, A. Y. Google's Deep Web Crawl. *Proceedings of the VLDB Endowment* 1 (2): 1241–1252, 2008.

Milo, T. and Suciu, D. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory*. Jerusalem, Israel, pp. 277–295, 1999.

Nguyen, H., Nguyen, T., and Freire, J. Learning to Extract Form Labels. *Proceedings of the VLDB Endowment* 1 (1): 684–694, 2008.

Nguyen, T. Prudent Schema Matching for Web Forms. Tech. rep., University of Utah, Salt Lake City, UT, USA, 2008.

Pei, J., Hong, J., and Bell, D. A. A Robust Approach to Schema Matching over Web Query Interfaces. In *Proceedings of the 22nd International Conference on Data Engineering Workshops*. Atlanta, GA, USA, pp. 46–55, 2006.

PMC, L. P. M. C. Welcome to Lucene! http://lucene.apache.org/, 2010.

Qiu, J., Shao, F., Zatsman, M., and Shanmugasundaram, J. Index Structures for Querying the Deep Web. In *Proceedings of the International Workshop on Web and Databases*. San Diego, CA, USA, pp. 79–86, 2003.

Qun, C., Lim, A., and Ong, K. W. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. In *Proceedings of the ACM International Conference on Management of Data*. San Diego, CA, USA, pp. 134–144, 2003.

Terrovitis, M., Passas, S., Vassiliadis, P., and Sellis, T. K. A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes. In *Proceedings of the 2006 ACM International Conference on Information and Knowledge Management*. Arlington, VI, USA, pp. 728–737, 2006.

Wang, Y., Lu, J., and Chen, J. Crawling Deep Web Using a New Set Covering Algorithm. In *Proceedings of the 5th International Conference Advanced Data Mining and Applications*. pp. 326–337, 2009.

Wu, W., Doan, A., and Yu, C. T. WebIQ: Learning from the Web to Match Deep-Web Query Interfaces. In *Proceedings of the 22nd International Conference on Data Engineering*. Atlanta, GA, USA, pp. 44–54, 2006.