Indexing Relations on the Web

Sergio Luis Sardi Mergen
Universidade Federal do Rio Grande do Sul(UFRGS)
Av. Bento Gonçalves, 9500,Porto Alegre
RS - Brasil
mergen@inf.ufrgs.br

Juliana Freire
School of Computing – University of Utah
Salt Lake City, U.S
juliana@cs.utah.edu

Carlos Alberto Heuser
Universidade Federal do Rio Grande do Sul(UFRGS)
Av. Bento Gonçalves, 9500,Porto Alegre
RS - Brasil
heuser@inf.ufrgs.br

ABSTRACT

There has been a substantial increase in the volume of (semi) structured data on the Web. This opens new opportunities for exploring and querying these data that goes beyond the keyword-based queries traditionally used on the Web. But supporting queries over a very large number of apparently disconnected Web sources is challenging. In this paper we propose index methods that capture both the structure of the sources and connections between them. The indexes are designed for data that is represented as relations, such as HTML tables, and support queries with predicates. We show how associations between overlapping sources are discovered, captured in the indexes, and used to derive query rewritings that join multiple sources. We demonstrate, through an experimental evaluation, that our approach scales to a large number of sources.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms

Algorithms

Keywords

Dataspaces, Indexing, Search Engines

1. INTRODUCTION

There is a very large volume of (semi) structured data on the Web. A recent study reports that there are over 144 million relations published as HTML tables in Google's index [3]. Other sources of structured information include Web services and online databases. Whereas Web documents have been traditionally modeled as bag of words and queried through simple keyword-based interfaces, having structured information opens up new opportunities for exploring and querying these data. By leveraging the structure, more precise queries can be posed which in turn lead to higher-quality answers. Furthermore, answers to queries can correlate information from multiple sources, enabling on-the-fly integration at an unprecedented scale.

But supporting structured queries over a very large number of data sources creates new challenges [1]. A practical solution must deal with the scale as well as with the dynamic nature of the Web, where new data sources are constantly added and existing data sources updated. In addition, the metadata associated with the data sources is often scarce. For example, for Web tables, usually, only the attribute names can be automatically extracted—no information is provided about attribute types or integrity constraints.

Dataspaces have been proposed as a new paradigm for managing structured Web sources [9]. Dataspace systems aim to support large networks of interconnected data sources with (an implicit or explicit) structure. These systems must address several problems, from supporting large volumes of heterogeneous data (in different formats) from autonomous data sources, to providing an integrated means for searching and querying the information.

In this paper, we focus on the problem of supporting structured queries in dataspaces. Our underlying model is based on three interconnected entities: relations, attributes and values. The relationships between these entities are captured in specialized indexes. In addition, the indexes capture relationships across relations, *i.e.,* relations that share attributes and values.

By treating each individual data source as a relation, we propose a new querying mechanism which is scalable, *i.e.,* able to handle a very large number of structured Web sources; and that supports simple, yet expressive queries that exploit the structure inherent in the data. Our goal is to let users declare queries with attributes, and to optionally specify selection predicates. Furthermore, if required, answers may join information from multiple sources.

We have performed an experimental evaluation of our indexes using a large set of HTML tables automatically gathered from the Web. Our results show that using our approach, queries can be answered efficiently. We also position our work against related work, through a comparison of different features, specially considering memory cost.

The remainder of the paper is organized as follows. Related work is discussed in Section 2. In Section 3, we present the basic concepts underlying our model and the problem definition. Section 4 describes the indexing mechanism we have proposed. Section 5 gives a detailed description of the search algorithms we use to find possible ways to answer user queries. In Section 6 we present the experimental results. In Section 7, we present a real application based on our solution, which allows interesting queries to be posed against information gathered from multiple relations. Section 8 presents our concluding remarks.

2. RELATED WORK

The problem of providing seamless access to structure data on the Web have already been addressed by a number of different works. Salles et al. [18] classifies these works into two different categories: schema first approach (SFA) and no schema approach (NSA).

SFA assumes the sources of information are already integrated into a global schema. This integration is materialized as mappings between the source schemas and the global schema. Classical solutions include global-as-view (GAV[17]) and local-as-view(LAV[10]), whose main difference is the direction of the mappings. Nevertheless, they all serve the same purpose: support query rewriting, in which a query posed to the global schema must be broken down into queries that conform to the local schemas[5].

Creating mappings and maintaining them as sources evolve and the global schema changes can be expensive and time consuming. This effort is required and justified for applications that are mostly static, cater to well-defined information needs, and integrate a relatively small number of sources. However, for exploring and integrating information sources on the Web, expecting the existence of a pre-defined global schema is not practical. To define a global schema, one first needs to know about which sources are available. And on the Web, there are too many of them. Besides, a single global schema is unlikely to be sufficient to fulfill the information needs of all users. Even if it is possible to model a very large global schema that contains constructs for comprehensive set of concepts, the effort spend on defining the mappings to the sources would become unfeasible, not to mention that Web sources are too volatile, which would require the mappings to be constantly updated.

In the other direction, NSA assumes that a global schema is not available. In such scenario, the goal is to match the user desire, expressed in a query language, to the set of sources available. To make matters worst, this process needs to be done on the fly.

In most NSA works, the user expresses queries as keywords, given the simplicity of the language and its proven success on search engines for unstructured data. Additionally, at some extent, NSA approaches tend to look at the sources as if they were collection of tuples (a.k.a. relations).

Following this direction, some works propose keyword search over relational databases. Such approaches model the solution as graphs whose nodes are connected based on foreign key relationships. Two possibilities emerge: instance-level graph[14] and schema-level graph[2, 13]. In the instance-level graph, the nodes are tuples of the database. Given a query, keywords are mapped to nodes in the graph. Each subset of the graph that links all of the selected tuples become individual answers. Results are then presented as nested information, according to the connections between the tuples. In the schema-level graph, the nodes refer to elements of the database schema. Like the instance level counterpart, keywords of a query map to nodes in the graph. Then, each subset of the graph that links all of the selected nodes derives a query that contains some answers.

Originally designed to relational databases(in the close-world-assumption sense), these approaches could be extended to Web information. However, it is not clear if and how they would be suitable to the large amount of information available on the Web, specially considering that foreign key relationships are unknown.

The structured universal relation (SUR) was also proposed as a way to query multiple structured Web sources using a simple keyword based interface [6]. Generally speaking, a database is virtually flattened as a single relation with a large number of attributes, so the user is free from having to provide join information. SUR requires an expert to specify compatibility constraints for relations as well as concept hierarchies for attributes.

To support queries over Web tables, Cafarella et al. [3] proposed an inverted list that links cells to the tables in which they appear. The horizontal and vertical offsets of the cell inside the table are also mapped. Similarly to the previously presented works, queries are expressed by keywords. The result of the search are pages whose inner tables contain terms that appear as keywords in the query. In order to effectively find the most relevant information first, the results are ranked according to a set of features. For instance, tables whose keywords appear as part of the header are deemed more relevant than tables whose keywords appear as part of a record. As a drawback of this approach, structure is not taken into account for the purpose of answering structured queries. As a consequence, selection predicates are not allowed, and the results are limited to information encountered on single tables.

Dong and Halevy [7] take a step forward with a solution that is able to perform queries with structure. The indexing schema they propose is based on keywords that points to a list of instances where they can be found. The index can be enriched with hierarchical information (terms that can substitute a keyword) and association labels, so that information from multiple sources can be joined. This solution is the first attempt to provide means for the user to express queries with selection predicates, even though predicates are restricted to the equality operator, unlike our approach that supports more types of comparison operators.

There are also pay-as-you-go approaches that lie in-between the SFA and the NSA. Works of this kind start with few mapping information and evolve over time, taking user feedback into account. Madhavant et al [15] states some architectural issues that need to be addressed in such a scenario. Salles et al. [18] provides a concrete framework that supports dataspace enhancing by adding relationships among the data.

3. BACKGROUND

In order to query structured Web data, we envision *structured* search engines that support simple, yet expressive queries which explore the underlying structure of the data.

Figure 1 shows a high-level overview of our architecture. Relevant relations are extracted from the Web and fed into the *Indexer*. Given a structured query, our goal is to find all possible ways to answer this query using relations found in the Index, *i.e.,* all possible rewritings [10].

The *Rewriter* is composed by search algorithms that access the indexes in order to derive rewritings for user queries.

Each rewriting can then be individually processed (by retrieving the current version of the relations from the Web or using cached relations), and its resulting tuples returned to the user.

The indexes (along with the search algorithms) we propose in this paper are important building blocks for these search engines, as they not only improve the performance for finding the rewritings, but also aid in the selection of relevant relations.

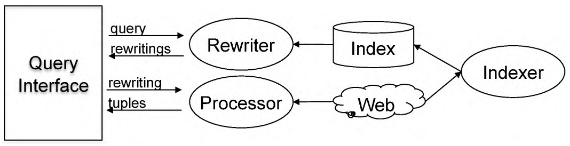


Figure 1: Architecture Overview

Data Model. We treat Web sources as relations. Relations provide a natural model for a significant number of structured Web sources, in particular, HTML Tables. We assume that the schema (*i.e.,* attribute names) and instances can be automatically extracted from the Web sources [4].

Figure 2 shows excerpts of (real) HTML tables whose relational structure was automatically extracted. Relation T_1 brings a list of movies starred by Leonardo DiCaprio, T_2 brings a ranked list of the best movies ever, T_3 brings a ranked list of movie scores and T_4 brings a list of movies from Martin Scorsese. These relations are used throughout the paper in order to exemplify how the querying process works.

Query Language. Our query language extends the traditional keyword-based interfaces with structural semantics. A query is defined as a list of attributes, followed by a (optional) list of selection predicates.

Example 1: *Find the title and year of movies released after 1990.* $Q_1 \rightarrow$ **title year [year > 1990]**

The bold sentence in Example 1 shows how to express a user request in the proposed query language. The attributes involved in brackets are the (optional) selection predicates, whereas the remaining attributes are projection predicates (attributes that need to be returned as part of the answer).

The user needs not to specify the sources (*i.e.,* the underlying relations). Instead, a search algorithm selects the relevant sources and derives queries that retrieve data from these relations. If the specified attributes span multiple relations, queries are derived that join these relations.

Rewritings. In our work, a rewriting is a conjunctive query where each subgoal refers to a relation. In traditional approaches for query rewriting[11], the answer to a query is presented as a union of rewritings (maximally contained rewriting). Instead of computing this union, we present to the user each local rewriting in isolation. Several reasons motivate this decision:

1. Depending on the size of the data set, the cost for computing all answers becomes prohibitive. Besides, the user would normally focus on the first results only.
2. It is easier to relate the resulting tuples to their respective sources. Knowing the provenance of the tuples helps to identify the best sources of information.
3. It is possible to associate a specific rewriting to answers understood as irrelevant by a user and ignore all answers that come from this rewriting.
4. It is possible to conceive a feedback mechanism, where the user is able to inform that the answers of a specific rewriting are not correct. Based on this information, the system could adapt the rewriting in order to enhance accuracy.

T1		
TITLE	GENRE	YEAR
Titanic	Romance	1997
The Aviator	Biography	2004
The Departed	Crime	2006

T3		
RANK	TITLE	COMPOSER
1	Star Wars IV	John Williams
2	Gone With the Wind	Max Steiner
3	Lawrence of Arabia	Maurice Jarre

T2	
RANK	TITLE
1	Citizen Kane
2	Casablanca
3	Titanic

T4			
TITLE	YEAR	NOMINATIONS	OSCARS
Casino	1995	1	1
GoodFellas	1990	6	1
The Aviator	2004	11	5

Figure 2: Source Relations

Figure 3 shows examples of queries/rewritings based on the relations of Figure 2. Although in our implementation rewritings are internally expressed in a simplified version of SQL, for clarity, here we show them using Datalog (attributes are described in the abbreviated form).

Rewritings that extract data from a single relation are called *single-relation rewritings.* An example of this kind of rewriting is given in R_{x1}. Conversely, rewritings that extract data from more than one relation are called *multi-relation rewritings.* An example of this kind of rewriting is given in R_{y1}. The relations of this rewriting are joined by the shared variable `title`.

Query		Possible Rewriting
Q_x	title year	R_{x1}(t,y) :- T_1(t, g, y)
Q_y	title genre oscar	R_{y1}(t,g,o) :- T_1(t,g,y1),T_4(t,y2,n,o)

Figure 3: Example of Rewritings

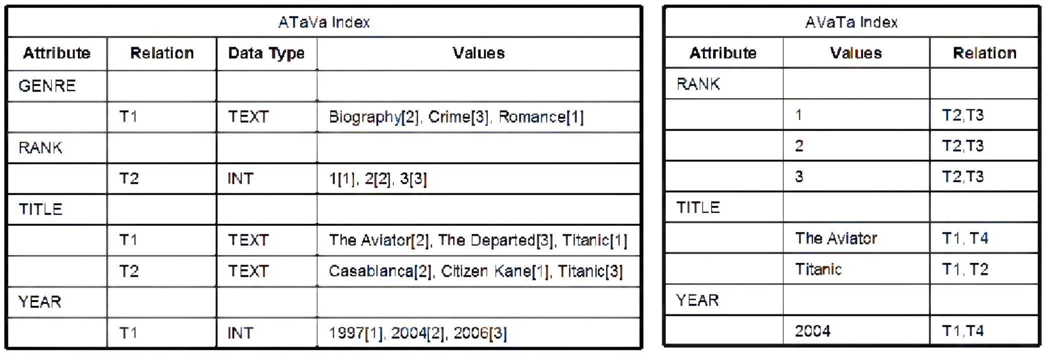
4. INDEXING

Information from Web relations are kept into two index structures: ATaVa and AVaTa.

ATaVa is a tree based index that allows navigation in the following direction: attribute $\rightarrow$ relations $\rightarrow$ values. Figure 4 shows some of the relations from Figure 2, indexed in the ATaVa (the remaining sources are not shown for the sake of space). This index is implemented as a tree, and for presentation reasons, they are visually presented as a table.

At the root level, the attributes are linked to their respective relations. Additionally, each pair attribute/relation (a, r) is associated with a data type and the list of values of a that appear in tuples of the relation r. Each value is enriched with the tuple ID that contains the value.

The data type of an attribute is normally not defined in Web data, but it can be easily discovered through a template based approach. For instance, FOCIH also has a library



ATaVa Index			
Attribute	**Relation**	**Data Type**	**Values**
GENRE			
	T1	TEXT	Biography[2], Crime[3], Romance[1]
RANK			
	T2	INT	1[1], 2[2], 3[3]
TITLE			
	T1	TEXT	The Aviator[2], The Departed[3], Titanic[1]
	T2	TEXT	Casablanca[2], Citizen Kane[1], Titanic[3]
YEAR			
	T1	INT	1997[1], 2004[2], 2006[3]

AVaTa Index		
Attribute	**Values**	**Relation**
RANK		
	1	T2,T3
	2	T2,T3
	3	T2,T3
TITLE		
	The Aviator	T1, T4
	Titanic	T1, T2
YEAR		
	2004	T1,T4

Figure 4: Examples of the ATaVa / AVaTA Indexes

of regular-expression recognizers for values in common formats[8].

The goal of the ATaVa index is to provide the location of the relations that contain the query attributes. Additionally, the data type and the list of values are used to verify if a relation satisfies the query predicates.

A simplified version of this index is obtained by removing the values level. In this case we call it the ATa index. While ATa reduces the storage requirements, it does not support queries with predicates. The experiments of Section 6.1 shows the behavior of these indexes on real Web data.

Similarly to ATaVa, AVaTa is also a tree based index. However, it allows navigation in the following direction: attribute → values → relations. Figure 4 shows the relations from Figure 2 indexed in the AVaTa. The structure resembles the one used in the ATaVa, but the order between the values and the relations is inverted, and no tuple ID is kept.

This index is populated only with attributes that appear in more than one relation, and only when there is at least one value in common as content of these attributes. Attributes that appear in AVaTa are called *join attributes*. Join attributes act as links between otherwise disconnected relations.

The goal of the AVaTa index is to provide the location of the relations that contain complementary data, which is useful for discovering multi-relation rewritings.

5. QUERY REWRITING

In what follows, we present a series of examples that illustrate the proposed rewriting algorithms. Special attention is dedicated to the process of finding multi-relation rewritings, where we start with a naïve solution and continue with more scalable approaches. It is also shown how the AVaTa is able to reduce the search space required to find relations that contain complementary data.

Example 2: *Find the title and year of movies.* Q_2**: title year**

Algorithm 5.1 describes how the relations that provide answers for Q_2 are located. The ATaVa index is scanned for each of the query attributes a_i. The function $locateRelations(a_i)$ returns the corresponding relations of a_i from the ATaVa (the relations that contain a definition for a_i).

The list of relations from each attribute are intersected, and the result of the intersection are the relations that can provide tuples that cover all attributes. Each of these become the single-relation rewritings r_{21}(t,y):-T_1(t,g,y) and r_{22}(t,y):-T_4(t,y,n,o).

```
findCompleteRelations(Query Q) returns C

1: //list of relations that contain all query attributes.
2: $C \leftarrow \{\}$ //Initially empty
3: for every attribute $a_i$ of Q do
4:     intersectRelations($a_i$,$C$)
5: end for
intersectRelations($a_i$,$C$)
1: if ($i == 0$) then
2:     $C \leftarrow locateRelations(a_i)$
3: else
4:     $C \leftarrow C \cap locateRelations(a_i)$
5: end if
```

Algorithm 5.1: The algorithm that finds relations that contain all query attributes

Example 3: *Find movies released after 2005.* Q_3**: title [year > 2005]**

This case involves a query with a selection predicate ([**year** > 2005]). Similarly to the previous example, the index is scanned for each of the query attributes. However, if a_i is a selection variable, the function $locateRelations(a_i)$ returns only the relations that satisfy the selection predicate.

The list of values of the ATaVa index are traversered in order to find out if a relation satisfies a predicate or not. The data type indicates the semantic of the comparison predicates. For example, **year** is a numeric value in the index, so it should be compared as such. Since the values are ordered in alphabetic order, this operation can be done in logarithmic time.

The lookup for the **year** attribute in Q_3 returns relation T_1 (this is the only relation that satisfies the predicate over **year**). As a result, only a single rewriting is produced: $r_{31}(t,y)$:-$T_1(t,g,y), y > 2005$.

Example 4: *Find romance movies released after 2000.* Q_4**: title [year > 2000] [genre = romance]**

Example 4 is more restrictive than example 3, as the predicates show. The only relation that covers all attributes of query Q_4 is T_1. However, in order for this relation to fully satisfy the selection predicates of the query, both predicates should be satisfied for the same tuples. In this case, no tuple of relation T_1 satisfies predicates `genre = Romance` and `year> 2000` at the same time, since there is no overlap between the tuples that satisfy the first predicate (tuple one) and the tuples that satisfy the second predicate (tuples two and three).

Given the tuple ID of the ATaVa, it is possible to obtain, for each table, the list of tuples that satisfy a specific selection predicate. With these lists, it is possible to discover whether a rewriting is able to return any answers at all. Section 5.5 gives more details about how this verification is performed.

Example 5: *Find the title, year and director of movies.*
Q_5**: title year director**

In this example, there is no source relation that covers all attributes of the query. For instance, relation T_1 covers only `title` and `year`, and relation T_2 covers only `title`. These relations, called incomplete, produce incomplete rewritings, where one or more attributes of the query are absent, such as $r_{51}(t, y, d)$:-$T_1(t, g, y)$ and $r_{52}(t, g, y)$:-$T_2(r, t)$.

Algorithm 5.2 shows how to find the list of incomplete relations. This list contains relations that cover at least one of the attributes (D), except those that cover all attributes (C). Moreover, the selection attributes (if any) always need to be covered (S). This last condition is required to prevent the generation of answers that may not be contained in the query. For instance, in Example 3, it is not possible to assure that the rewriting $r_{32}(t, y)$:-$T_2(r, t)$ is contained in the query Q_3, because it is unknown whether all movies from T_2 were released after 2005.

```
findIncompleteRelations(Query Q) returns D

 1: //relations that contain all query attributes.
 2: $C \leftarrow \{\}$ //Initially empty
 3: //relations that contain only some of the query at-
    tributes.
 4: $D \leftarrow \{\}$ //Initially empty
 5: //relations that contain all of the selection attributes.
 6: $S \leftarrow \{\}$ //Initially empty
 7: for every attribute $a_i$ of Q do
 8:    intersectRelations($a_i$,$C$)
 9:    $D \leftarrow D \cup (locateRelations(a_i))$
10:    if ($a_i$ is a selection attribute) then
11:       intersectRelations($a_i$,$S$)
12:    end if
13: end for
14: if (query contains selection predicates) then
15:    $D \leftarrow (S - C)$
16: else
17:    $D \leftarrow (D - C)$
18: end if
```

Algorithm 5.2: The algorithm that finds relations that contain all (C) and some (D) of the query attributes

Example 6: *Find the title, director and genre of movies.*
Q_6***: title director genre.***
For this example only, consider the following data sources: $T_5(t, g)$, $T_6(t, d)$, $T_7(t, g)$ *and* $T_8(t, g)$.

Again, every source relation is incomplete with respect to the query. However, it is possible to find combinations of these relations that cover all query attributes (and that joined together may produce the query result).

The first step is to separate the incomplete relations into join lists (JLs). A JL is a list that clusters together relations that cover the same join attribute. Recall that a join attribute is an attribute that appear in the AVaTa index. From the attributes of Q_6, just `title` is a join attribute, so there is a single JL.

Given a JL, the goal is to find all combinations of relations that form valid multi-relation rewritings - i.e. rewritings that are both complete and minimal.

A *minimal* rewriting is one that produces different results if any of its subgoals are removed (non minimal rewritings are more time consuming since the number of relations that need to be joined is larger). Thus, a combination cannot form a multi-relation rewriting if one or more relations could be removed from it and it still covers all attributes of the query.

A *complete* rewriting is one that covers all query attributes. It is semantically equivalent to the AND operator of traditional keyword-based query languages. Thus, a combination cannot form a valid rewriting if its relations do not cover all attributes of the query. This restriction simplifies the process of finding the combinations. The support for incomplete multi-relation rewriting is left for future work.

The JL created from query Q_6 is composed by {T_5[d], T_6[g], T_7[d], T_8[d]}. The attributes inside the square brackets represent the complement of each relation (the attributes of the query that are not covered by the relation). In the running example, relations T_5,T_7 and T_8 do not cover attribute `director` and relation T_6 does not cover attribute `genre`.

In what follows we present different strategies to scan the relations in the JL in order to form the valid combinations.

5.1 Naïve Algorithm

The naïve algorithm finds valid combinations using a simple search strategy[1]. Algorithm 5.3 shows how this strategy works. The entries of the JL are processed sequentially, and every possible combination of entries is performed. For each combination, the complements of the combined relations are intersected. The complement of a relation t_i is returned by the function *getComp*(t_i).

Figure 5(a) shows all combinations viewed as a tree. It is important to remark that this tree does not reside in memory, since the paths are built and consumed as the JL is processed. The path from a leaf to the root forms a combination.

Paths that lead to a valid combination can be tested by intersecting the complements of the nodes that belong to the path. An empty intersection means that all attributes of the query are covered.

In the example, three valid paths emerge(/T_5/T_6, /T_6/T_7, /T_6/T_8), which generate the following rewritings: r_{61}(t,g,d):-T_5(t,g), T_6(t,d), r_{62}(t,g,d):-T_6(t,d), T_7(t,g) and r_{63}(t,g,d):-T_6(t,d), T_8(t,g). Note that the join attribute of the JL becomes the join attribute of the rewritings.

[1]The naïve algorithm is introduced just to simplify the presentation of the algorithms we are actually proposing.

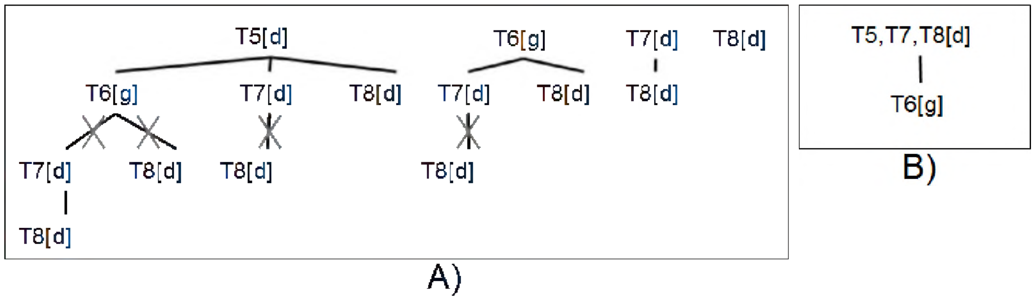


Figure 5: Trees of Possible Combinations. The trees presented in a) and b) show respectively the combinations found when the Stream-Driven algorithm and the Template-Driven algorithms are used.

```
findValidPaths_1(JL)
 1: for every relation $t_i$ of JL do
 2:     $MA \leftarrow getComp(t_i)$
 3:     findValidPaths_2(JL,i,MA)
 4: end for
findValidPaths_2(JL,x, MA)
 1: for every relation $t_i$ of JL, where $i > x$ do
 2:     $MA_{aux} \leftarrow MA \cap getComp(t_i)$
 3:     if ($MA_{aux} == \{\}$) then
 4:         this is a valid path
 5:     end if
 6:     $findValidPaths_2(JL, i, MA_{aux})$
 7: end for
```

Algorithm 5.3: The algorithm that finds valid paths, i.e. paths whose corresponding relations form valid multi-relation rewritings

5.2 Stream-Driven Algorithm

The Naïve Algorithm is based on a brute-force backtracking approach. As a consequence, is accesses branches of the tree that cannot form valid combinations (*e.g.*, the path $/T_5/T_7/T_8$ is not complete and the path $/T_5/T_6/T_7/T_8$ is not minimal). The Stream-Driven Algorithm reduces the number of node visits by removing some of the invalid paths (see the cuts depicted in Figure 5A)), which prevents the subtree underneath the removed path from being processed.

Given a path $p = \{n_1..n_j\}$, a node n_{j+1} (n_{j+1} being a node that succeeds n_j in the JL) is not traversed if:

- the intersection of the complements of p is already empty (*e.g.*, the path $/T_5/T_6$). This means that the path traversed so far is already a complete rewriting, and it makes no sense to continue along this path.

- the intersection of n_1 to n_{j+1} is the same as the intersection of n_1 to n_j. In other words, the last node of the path does not reduce the intersection. (*e.g.*, the path $/T_5/T_7$). This means that one of the nodes along the path leads to a non-minimal rewriting.

Algorithm 5.4 overrides the function $findValidPaths_2$ of Algorithm 5.3 by adding the cutting rules.

5.3 Template-Driven Algorithm

Observe from the previous example that some entries in the JL have the same signature (their complements are equal).

```
findValidPaths_2(JL,x, MA)
 1: for every relation $t_i$ of JL, where $i > x$ do
 2:     $MA_{aux} \leftarrow MA \cap getComp(t_i)$
 3:     if ($MA_{aux} == \{\}$) then
 4:         //valid path
 5:     else if ($MA - MA_{aux} == \{\}$) then
 6:         //path does not reduce intersection(not minimal)
 7:     else
 8:         //only in this case the sub-paths are processed
 9:         $findValidPaths_2(JL, i, MA_{aux})$
10:     end if
11: end for
```

Algorithm 5.4: The algorithm that finds valid paths and cuts the invalid ones

In order to find valid rewritings, entries with the same signature cannot be part of the same combination, since they would lead to non-minimal rewritings.

In the template-driven algorithm, relations with the same complement are grouped within a single entry. In other words, there will be no more than one entry with the exact same complement. In this case, the size of the JL tends to be small, since it is no longer related to the number of relations, but to the number of possible different complements.

Equation 1 shows how to compute the number of possible different complements for a query with $n+1$ attributes. For instance, having a query with five attributes (one attribute that is fixed (the join attribute) and four other attributes), in the worst case there would be 24 entries (combination of one,two,three or four attributes).

$$Maximum\ \#\ of\ complements = \sum_{k=1}^{n} \frac{n!}{k!(n-k)!} \qquad (1)$$

After the JL is generated, a template tree is build. This tree contains only paths that lead to valid combinations. Algorithm 5.4 serves as a base to produce the template tree from the JL (line 4 would have to be replace by a routine that reads the selected path and adds it to the template tree).

The template tree for example 6 is described in Figure 5(b). The valid combinations are formed by traversing the tree in a left-to-right, breath-first strategy, as described in Algorithm 5.5. The path from the leaf to the root leads to a group of combinations, where each combination comes from the cartesian product of the list of relations that are part of each node.

Conversely to the previous algorithms, this tree is built in memory. However, it is considerably smaller then the naïve /streamed tree, as the example shows. Another positive aspect of the Template-Driven approach is that the number of validity checks (check if complete and minimal) that need to be performed can be greatly reduced, when dealing with a large number of relations.

```
findCombinations(TREE)
1: for every root node $n_i$ of the template tree $TREE$ do
2:     for every relation $t \in n_i$ do
3:         $T_{aux} \leftarrow t$
4:         $findCombinations(n_i, T_{aux})$
5:     end for
6: end for
findCombinations(node, T)
1: if ($node\ is\ a\ leaf\ node$) then
2:     //$T$ is a valid combination
3: end if
4: for every child node $n_i$ of $node$ do
5:     for every relation $t \in n_i$ do
6:         $T_{aux} \leftarrow T \cup t$
7:         $findCombinations(n_i, T_{aux})$
8:     end for
9: end for
```

Algorithm 5.5: The algorithm that finds valid combinations from a template tree

5.4 Using the AVaTa Index

The AVaTa index is use to optimize the process of finding multi-relation rewritings. Instead of a single large JL for each join attribute, several small JLs are created for the same join attribute. These small JLs only contain relations that share at least one value for the join attribute.

Example 7: *Find the title, rank and year of movies.* Q_7: **title rank year**.

Example 7 is a case that would require relations to be joined. Without the information from the AVaTa, and considering `title` as a join attribute, the JL would be (T_1, T_2, T_3, T_4). However, the AVaTa tells us that only (T_1,T_4) and (T_1,T_2) have overlapping movies. Thus, the large JL is split into the smaller JLs (T_1,T_2) and (T_1,T_4). In the experiments, the benefits from using smaller JLs become more evident, in cases where the number of relations is higher.

An indirect benefit of this index is that it prunes rewritings that would not produce any valid answers. For instance, without the AVaTa, one of the possible rewritings of Q_7 would be $r_{71}(t, r, y)$:-$T_2(r, t), T_4(t, y, n, o)$. However, this rewriting would bring no answers, since movies from the involved relations do not overlap.

5.5 Emptiness Check

It is possible that some of the produced rewritings return empty sets as answers. These are called empty rewritings. These may happen when the relations involved do not provide tuples that satisfy the selection and / or the join predicates, specially when there is a multitude of relations available.

To prevent empty rewritings from being presented to the user, the emptiness check verifies if a given tuple provides the necessary information that is required by a user query.

Example 8: *Find the title, genre and oscars of movies released after 2000.* Q_8**: title genre oscars [year>2000]**.

Since no source relation contains all attributes of the query Q_8, the rewritings would necessarily contain join predicates, as well a selection predicate for the `year` attribute.

Given the tuple ID of the ATaVa, it is possible to obtain, for each relation of a rewriting, a list of the tuples that satisfy a specific predicate that involves the relation. Given a selection predicate over a relation r, the list would contain the instances of r that satisfy this predicate. Likewise, each join predicate generates two lists, one for each attribute of the join. These lists contain the tuples that are able to satisfy the join.

All lists of instances that refer to the same relation are intersected. If the intersection is not empty, it means that at least one tuple of the relation satisfies all predicates.

For consistency sake, the lists of the selection predicates are intersected first. This measure prevents dangling tuples, from instances that were joined before the selection predicate had the chance to remove them.

Note that, in special cases, the predicates can be applied sooner if the process of rewriting discovery, and the emptiness check becomes unnecessary. Two situations are possible: 1) when the rewriting contains a selection predicate, and no join predicate. In this case, the relations filtering occurs inside the *locateRel*() function, before any rewriting is produced. This verification can be processed in this early stage even if more than one selection predicate is provided, as long as they refer to the same attribute (e.g. "year>1995 year<2000"). 2) when the rewriting contains join predicates and no selection predicate. Under these circumstances, the emptiness check is irrelevant, as long as the AVaTa is used during query rewriting. This index assures that all involved relations in a rewriting share at least one value for the join attribute.

Interestingly, it is possible to answer a query using only information from the indexes, without ever accessing the real sources. This computation is rather similar as the one used to perform the emptiness check. Given the tuple IDs that satisfy the predicates, the values from the satisfying tuples can be retrieved from the indexes.

6. EXPERIMENTAL EVALUATION

6.1 Combination Cost

In this section we discuss how variations of the search strategies affect the cost to find multi-relation rewritings.

The data set used in this experiment contains 195 Web Pages extracted from Wikipedia sites related to movies. From these sources, we apply a template extraction to find HTML tables that contain information about movies. A table matches a template if it contains one of the following attributes: `film`, `movie` and `title`. A total of 671 tables were indexed. By manually analyzing 10% of the indexed tables, we estimate that 95% of them are indeed part of the movie domain. This data set is intended to show how our approach scales for relations that are part of the same domain.

Four different settings are compared, as Figure 6 shows. First, the template-driven(TD) and the stream-driven(SD)

	AVaTa	¬ AVaTa
Stream-Driven(SD)	SD+	SD
Template-Driven(TD)	TD+	TD

Figure 6: Settings Used to Measure Combination Cost

algorithms are executed in isolation. Then, for each algorithm, the AVaTa index is used to split the JL into smaller lists.

Figure 7 shows the number of nodes visits required to find valid combinations(a) and the time required to find the combinations (b). The number of node visits indicates the number of times a node of a JL had to be accessed in order to find the rewritings. These results correspond to the query `title year director cast genre role`. Other queries showed similar results, so we omit them here.

Approaches that use the AVaTa index are clearly better—they stand at the bottom part of the graph. The reason is that only a small fraction of the rewritings are not empty, and this index assures that only this small fraction is processed.

Also note that there is a reduction in the number of visits when the TD algorithm is used (a). This clearly shows that, for practical cases, the template tree is indeed much more compact than the streamed tree.

On the other hand, the time difference between TD and SD is almost irrelevant (b), which indicates that the time required to traverse the nodes is smaller than the time required to perform the other operations, such as the emptiness check.

Interestingly, the TD approach is worse than the SD approach when the AVaTa is used (b). The reason is that the AVaTa creates small JLs (the size of the larger one was 13), and the cost of computing template trees becomes the most expensive operation when the JLs are too small. In the future, we intend to investigate how to automatically choose the best strategy for each case. One possible tuning would be to use the template tree only when a JL is larger than a pre-defined threshold.

6.2 Cost of the Emptiness Check

We also evaluated the benefits from having the emptiness check performed by the search engine, as opposed to leaving it up to the user to find it out by himself.

Figure 8 shows the query workload we have used. The first three queries differ in the number of attributes required. No source relation covers all query attributes, so all rewritings of these queries need to perform joins. The last two queries show variations in the selection predicate for the attribute year.

Query	Type of Predicate
Q1: title year director cast genre notes	Join Predicate
Q2: title genre director year cast	Join Predicate
Q3: title cast role	Join Predicate
Q4: title year [year = 2000]	Selection Predicate
Q5: title year [year < 1995]	Selection Predicate

Figure 8: Query Workload

Table 1 shows the results achieved. Column c1 refers to the total number of rewritings found and column c2 refers to the number of rewritings that are not empty.

The comparison between c1 and c2 shows that only a small part of the total number of rewritings actually return some information. Also, note that the number of rewritings is very large for the first three queries.

Column c3 refers to the number of empty rewritings the user would have to open until 10 non empty rewritings are found (if no emptiness check is applied). This column indicates the level of effort the user is submitted to when reading the answers of a query. For instance, in the best case scenario, only 2 empty rewritings would have to be opened (for query Q5 and having a data set of 101 source relations). However, in other cases, this number can grow to the order or thousands (Q1 and Q2).

6.3 Comparison with Related Work

The work of Dong and Halevy [7] is pretty much similar to ours in nature, in that the ultimate goal is to seamlessly query a large corpus of structured information available on the Web. In what follows we present a list of topics related to this goal, in which we compare the main differences between these two index mechanisms.

Memory Cost: In order to evaluate memory cost, we collect data from the WT10G data set. This collection contains over 1.5M web pages crawled from the Web (`http://ir.dcs.gla.ac.uk/test_collections/wt10g.html`). From these sources, we apply a more general template extraction than the one used in the movie data set. A table matches a template if it contains one of the following attributes: **`artist, city, company, country, name, product and title`**. In the end, a total of 3471 tables were indexed (out of 2845 pages).

From the work of Dong and Halevy [7], we have evaluated two index variations: ATIL and AAIL. The ATIL is a list of keywords, where each keyword corresponds to the concatenation of a value and a column. Each keyword contains a inverted list of tuple instances. For example, 2004//year:[T1-2, T4-3] indicates that the column `year` contains the value 2004 in two different tuples: T1-2, represents the second row of table T1 and T4-3 represents the third row of table T3.

The AAIL is a variation of ATIL that supports associations. An association is a role between two related tuples. Consider Figure 2 as an example. There is an association between the second row of T_1 (T1-2) and the third row of T_4 (T4-3), since they expose information about the same movie.

Associations are bi-directional, and each direction is given a role name. This name makes it possible to store the associations in the index. For instance, consider `awarded` is the role name for the directional association T1-2 to T4-3. This association is stored in the AAIL with a number of keywords, one for each column of T4-3, where the value of the column is the prefix (e.g. The Aviator//awarded, 2004//awarded, 11//awarded and 5//awarded). Additional keywords are needed to represent the other side of the relationship as well.

Our approach has no support to identifying associations as proposed in [7]. Instead, we updated AAIL with the associations we were able to find automatically, for tuple instances that share the same column name and value. The name of the association became the name of column prefixed with the word `same`. For instance, if the column name is title, the name of the association becomes `same title`, in both directions.

All indexes are represented as Java primitive data types.

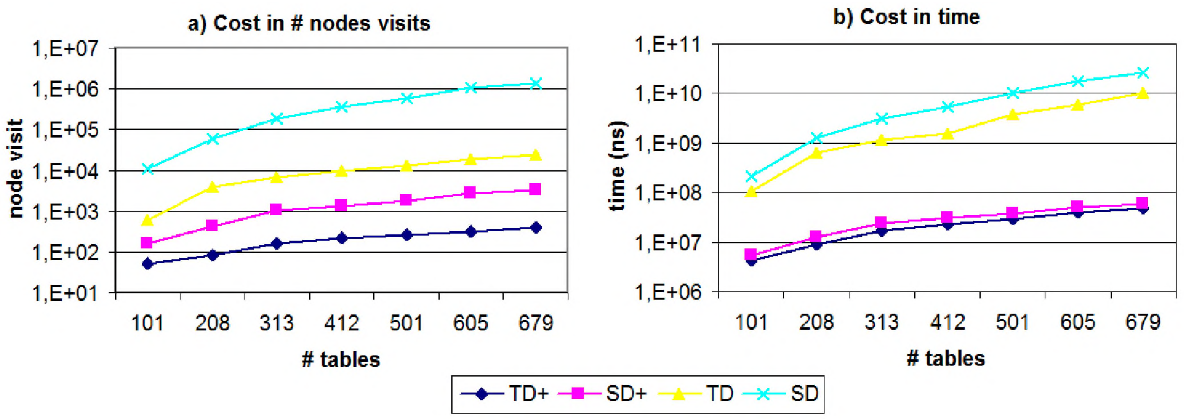


Figure 7: Cost for Finding Rewritings

#tables	Q1			Q2			Q3			Q4			Q5		
	c1	c2	c3	c1	c2	c3	c1	c2	c3	c1	c2	c3	c1	c2	c3
101	529	23	346	529	23	346	231	0	231	7	1	6	7	5	2
208	3780	40	1246	3780	40	1239	1755	14	570	28	8	20	28	19	5
313	6496	78	624	6496	78	628	3016	51	288	28	8	20	28	19	5
412	9212	102	1513	9212	102	1520	4277	74	719	28	8	20	28	19	5
501	12494	128	289	12006	130	269	5796	100	269	29	9	20	29	20	5
605	18469	161	829	17272	163	907	7620	116	832	34	10	24	34	22	5
679	23560	205	403	20340	207	391	9605	157	391	36	12	24	36	24	5

Table 1: Statistics about the Queries

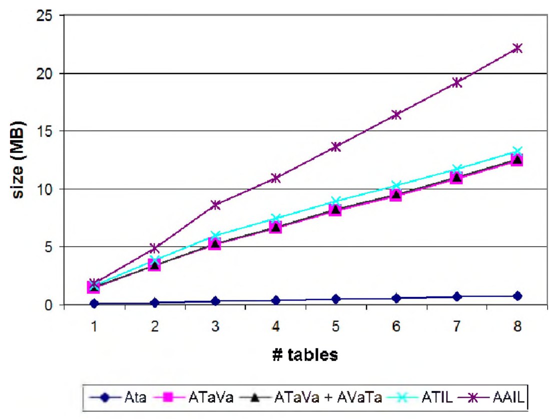


Figure 9: Memory Consumption for the WT10G Data Set

Plus, the required inverted lists are structured as primitive arrays. The size of an index is computed as the total amount of bytes the index occupies. It is important to remark that we care about optimizing memory consumption, for all evaluated indexes. To accomplish this, we prevent duplications of objets - unique objects are represented only once. The data used in the experiments as well as the source code that computes the memory cost are available at `http://www.inf.ufrgs.br/~heuser/atava.zip`.

The size of all indexes grows linearly with respect to the number of tables, as Figure 9 shows.

At the very low part of Figure 9 is the ATa index. Despite its low memory consumption, it does not provide enough information to perform the emptiness check. On the other hand, the ATaVa can perform the emptiness check, but it takes a considerably higher amount of memory.

ATaVa is slightly better than ATIL. These indexes are analogous, in the sense they do not store associations but allow queries with selection predicates.

We also compare the associations-aware indexes ATaVa + AVaTa and AAIL.

The memory cost in ATaVa + AVaTa is lower that its analogous AAIL. The reason is the overhead involved when adding an association in AAIL. Adding an association to an instance with n columns, implies in adding n new keywords (except when the keyword already exists - in this case it is represented only once).

Interestingly, adding the AVaTa to the ATaVa hardly increases the memory consumption. Recall that the AVaTa only keeps information about the join attributes. Naturally, the size of this index varies according to the total number of join attributes. However, we have found that only a small part of the whole data set needs to be stored in this index, even considering all existing attributes. Our statistics (considering both movie and wt10g data sets) indicate that 73% of the table values are unique, 22% of them appear in different relations, but for attributes with different names, and only 5% appear in different relations for attributes with the same name.

Ability to answer queries without comparison predicates: ATIL was designed with the purpose of answering selection predicate queries, and not to merely define the attributes of interest. In order to find out which tuples contain information about a specific attribute, a full scan in the index is required, since the attribute part of the indexed keywords is hidden behind the value part. One possible overcome would be to index additional keywords, using only attribute names. We did implemented a variation of the ATIL that adds these new keywords. The results are positive, and indicate that this variation does not have a severe impact on the index.

In our approach, this type of query can be answered using information from the ATaVa. In fact, the ATa index proves to be enough, since the values level is not required.

Ability to answer queries with comparison predicates: ATIL does not easily support queries with comparison predicates other than equality. The index would have to be scanned considering only the value part of the keyword. From the several keywords that may match the predicate, only those that correspond to the correct attributes should be considered. Furthermore, the index does not smoothly accommodate data types, which are required in order to understand the semantics of a comparison predicate.

In the ATaVa, given a selection predicate, the correct column/data type is directly identified. Additionally, since the values in the ATaVa are stored in alphabetic order, selection predicates can be answered efficiently.

Index Distribution: The ATaVa/AVaTa indexes can be distributed in several nodes, creating partitions at the attribute level. Given a query, only a few nodes need to be accessed (the ones that define the query columns).

Partitions in the ATIL would have to be created at the values level, since the keywords are prefixed by the value. Queries without comparison predicates, such as `"find title, year and director"` would have to access all partitions.

Ability to find associations among relations: AAIL supports a general kind of association. An association is a role between two tuples, and its not conditioned to the equality of column names. However, automatically finding associations between tuples and defining their role names is a complex process, and probably would have to be manually consolidated. In [7], it is not clear how this information is discovered.

Associations in our approach are restricted to tuples whose information share the same column name and value. This reduces the number of associations found, but it is a process that can performed automatically. Of course, this comes with a price of lower precision / recall rates, when false positive/negatives are identified, specially when mixing information from multiple domains into a single index.

Nevertheless, in the context of exploratory search, this behavior is acceptable, and sometimes, can even produce interesting answers. For instance, given the movie source `(title, director)` and the book source `(title, year)`, the query `title director year` would return the title and director of a movie, along with the year in which a homonymous book was released.

Despite the benefits, there is a caveat in our solution. In order to find associations that provide answers to a query, it is required that the query contains a join attribute (an attribute that acts as a link between tuples). In the examples demonstrated in this paper, all queries contained the join attribute `title`, even when the title itself was not necessary as part of the answer. If the provided columns of the query are not good join attributes, the relevant associations may not be found.

We intend to leverage this limitation by artificially adding join attributes to the query. Given a query, we intend to use schema-completion techniques [3] to discover which lacking attributes best match the schema of the query, and try to use these as join attributes.

#	Query
1.	city [state = UT]
2.	city [zipcode > 53701]
3.	zipcode [city = madison]
4.	product [company = ibm]
5.	restaurant telephone_ [city = malibu]
6.	company location ticker
7.	country country_code monetary_unit

Figure 10: Example of Arbitrary Queries

Ability to answer keyword queries: Keyword queries (where no metadata information is provided) are easily answered with the ATIL, since all indexed keywords are prefixed by the value. In our approach, it would be too cumbersome to perform this type of query.

7. APPLICATION

In this section we create an application that demonstrates how the search engine behaves on real data retrieved from the Web. As data set, we used the tables extracted from the WT10G collection. This data set is much larger than the movie data set and the universe of discourse is much more diverse.

Interestingly, from the seven join attributes used in the extraction template, several different domains could be retrieved, such as restaurants, industry and politics. Therefore, this data set is particularly interesting for wide purpose queries, such as the ones asked in a horizontal search engine.

Figure 10 shows a list of arbitrary queries posed to our search engine. Queries from one to five are answered using single sources of information (a single Web table). The emptiness check assures that only sources that satisfy the predicates are selected. For instance, one of the rewritings for the first query returns a list of 16 cities of Utah.

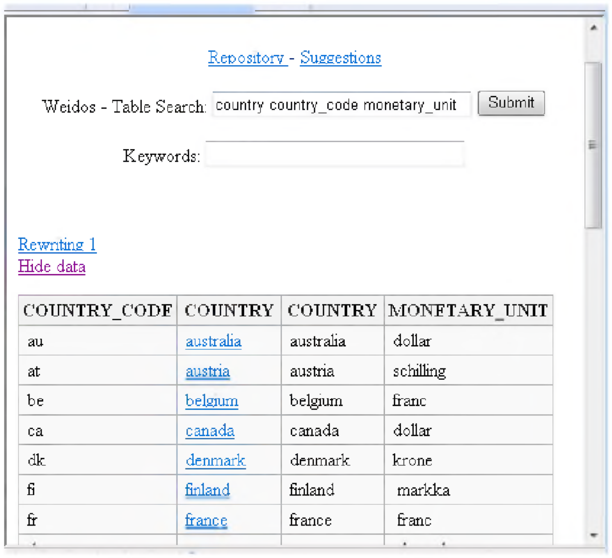
Queries six and seven are answered using multiple sources of information, having `company` and `country` as join attributes, respectively. Some of the results obtained for query seven are shown in Figure 11.

It is important to notice that the number of relations from the WT10G data set that belong to the same domain is very small. Despite the size of the collection, there are only few relations that contain the same or similar schema. We did try to perform scalability experiments with this collection, but the limited amount of information that overlaps did not give us enough data to draw conclusions. However, we do expect that scalability results using larger overlapping schemas would be similar to the ones provided for the movie data set.

8. CONCLUSIONS

We propose a new model for querying and correlating information from multiple structured Web data, where the data sources are treated as relations. This model allows users to pose keyword queries that contain attribute names and predicates over these attributes.

So that these queries can be effectively and efficiently translated into queries over the underlying relations, we designed an indexing mechanism and a set of algorithms. Besides supporting predicate queries, these algorithms also enable the derivation of rewritings that join multiple data sources that contain complementary data. Our experimen-



COUNTRY_CODE	COUNTRY	COUNTRY	MONETARY_UNIT
au	australia	australia	dollar
at	austria	austria	schilling
be	belgium	belgium	franc
ca	canada	canada	dollar
dk	denmark	denmark	krone
fi	finland	finland	markka
fr	france	france	franc

Figure 11: Some Results for the Query "country country_code monetary_unit"

tal results are promising and indicate that our approach is both scalable and efficient.

For future work, we intend to leverage the way associations between different relations are discovered. Currently, associations are created between relations that cover the same attribute, which becomes the join attribute of the association. However, such approach may lead to incorrect associations, such as cases where an improbable attribute is chosen as a join attribute(e.g.year). Additionally, schema matching techniques will be investigated in order to allow associations to be created between attributes whose names are different [12, 16].

Furthermore, there is a need to rank the rewriting so that the most relevant results appears first. The number of attributes covered and the estimated number of tuples returned are possible features that may be used, and that are easily extracted from the indexes.

9. ACKNOWLEDGMENTS

This paper was supported by CNPq(projects CT-INFO - 550891/2007-2, Universal - 481055/2007-0, scholarship) and CAPES(scholarship).

10. REFERENCES

[1] R. Agrawal and et al. The claremont report on database research. *SIGMOD Rec.*, 37(3):9–19, 2008.

[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *International Conference on Data Engineering (ICDE)*, pages 5–16, Washington, DC, USA, 2002. IEEE Computer Society.

[3] M. J. Cafarella, A. Halevy, D. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *VLDB Endowment*, 1(1):538–549, 2008.

[4] M. J. Cafarella, A. Y. Halevy, Y. Zhang, D. Z. Wang, and E. W. 0002. Uncovering the relational web. In *International Workshop on the Web and Databases (WebDB)*, 2008.

[5] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

[6] H. Davulcu, J. Freire, M. Kifer, and I. V. Ramakrishnan. A layered architecture for querying dynamic web content. In *SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 1999. ACM Press.

[7] X. Dong and A. Halevy. Indexing dataspaces. In *ACM SIGMOD international conference on Management of data*, pages 43–54, New York, NY, USA, 2007. ACM.

[8] D. W. Embley, D. Campbell, Y. Jiang, S. Liddle, D. Lonsdale, Y. k. Ng, and R. Smith. Conceptual-model-based data extraction from multiple-record web pages. *Data & Knowledge Engineering*, 31:227–251, 1999.

[9] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, 2005.

[10] A. Y. Halevy. Theory of answering queries using views. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):40–47, 2000.

[11] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[12] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *Sigmod*, pages 217–228, 2003.

[13] V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *International conference on Very Large Data Bases (VLDB)*, pages 670–681. VLDB Endowment, 2002.

[14] A. Hulgeri and C. Nakhe. Keyword searching and browsing in databases using banks. In *International Conference on Data Engineering (ICDE)*, pages 431–440, 2002.

[15] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350. www.crdrdb.org, 2007.

[16] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[17] J. D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.

[18] M. A. Vaz Salles, J.-P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi. itrails: pay-as-you-go information integration in dataspaces. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 663–674. VLDB Endowment, 2007.