Structural text search and comparison using automatically extracted schema

Michael Gubanov
University of Washington
mgubanov@cs.washington.edu

Philip A. Bernstein
Microsoft Research
phil.bernstein@microsoft.com

ABSTRACT

An enormous amount of unstructured information is present on the web, in product manuals, e-mails, text documents, and other information sources. However, there is not enough support to automatically infer sufficient structure from these data sources to be able to pose queries comparable in power to SQL.

We present a prototype of new text database management system capable to automatically infer schema from text using natural language processing. It leverages extracted schema by supporting powerful structural search and fuzzy join operator between extracted entities.

1. INTRODUCTION

Despite vast amount of *unstructured* data on the web, keyword-search [7] is often the only way to find needed information. PageRank - Google's algorithm to rank web pages and display the best ranked pages first to the user currently depends on more than 500 million variables and 2 billion(!) terms. In addition, PageRank also analyzes the full content of a page and factors in fonts, subdivisions, the precise location of each word, and the content of neighboring web pages [1]. To summarize, it is, probably, the world's most complex algorithm and it is getting even more complicated every day, because Google is working to improve it!

By contrast, System R [16] was the first relational database management system prototype that introduced a relational algebra engine for storing and querying *structured* data. Structured Query Language(SQL) - a powerful language was born from relational algebra with the purpose to query structured data represented as entities with attributes and relationships between them or a *database schema*. SQL made possible to focus user query to a *specific structure* within the database schema and retrieve quickly *only* the needed information thus leveraging the *structure* and getting *focused and precise answers* to the query. Of course, if needed, it is possible to do keyword-search over databases largely ignoring available structure (e.g. [4]).

Inferring structure (or schema) is a key problem in any solution trying to support a richer query language than *keyword search* over unstructured data in order to provide more focused and precise results. Our main contributions are the following novel algorithms that comprise a new text database management system (TDBMS). It is implemented on top of a relational engine.

- a fully automatic algorithm to extract schema from text and perform structural search using the schema
- algorithms for *fuzzy join* between entities in the extracted schema and ranking join results
- a *concept matching* algorithm to detect similar concepts expressed by different words in text

We apply our results to perform structural search and automatic software comparison by extracting schema from freely available product manuals. We convert the manuals to plain text before running the algorithms (plain text files are 3-4 Mb each).

InnoDB
InnoDB offers all four transaction isolation levels described by the SQL standard
InnoDB provides full ACID compliance
InnoDB supports multiple granularity locking which allows coexistence of record locks and locks on entire tables
...

Table 1: Structural search on InnoDB/supports

Structural search is focused on the extracted structure and therefore returns more precise results. For instance, having extracted the schema from the *MySQL* manual we can issue a selection query on entity *InnoDB*[1] and its attribute *support* (and its synonyms and grammatical forms). This returns 26 sentences, three of which are shown in Table 1. By contrast, keyword search on the same data using keywords *InnoDB* + (*provide* or *support* or *offer*) returns near 50 sentences, ≈ 35% of which were not matching the focus of a structured query (e.g. "You can omit these command lines if you to not require InnoDB or BDB support"). Thus, structural search is more focused and therefore performs more precisely than keyword-search at the expense of coverage (in this case, missed 4 useful sentences (e.g. "Support for XA transactions is available for the InnoDB storage engine").

Next, we applied our *fuzzy join* algorithm to compare indexing support in *PostgreSQL* and *MySQL* database servers. Comparing software is a widely known complex problem. It is especially important for large enterprises that commonly have to choose between expensive products. Usually, the

[1]one of the MySQL storage engines

Storage engine	PostgreSQL	rank	optrank
Storage engine allowable index types are MyISAM: B-TREE, InnoDB: B-TREE, Memory: HASH B-TREE	PostgreSQL provides several index types: B-tree R-tree Hash GiST	4.2	$1\frac{6}{15}$
MEMORY storage engine implements both HASH and B-TREE indexes	The PostgreSQL query planner will consider using an R-tree index whenever an indexed column is involved in a comparison using one of these operators: $\ll$ $\gg$ $\ll\vert$ $\gg\vert$ ...	$1\frac{1}{20}$	$1\frac{1}{20}$
All storage engines support at least 16 indexes per table and a total index length of at least 256 bytes	PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved	0.1	$\frac{1}{20}$

Table 2: Compare indexing support of MySQL and PostgreSQL by joining *Storage engine* and *PostgreSQL* concepts

problem is solved by hiring specialists in the art or by subcontracting to an external company to do the analysis manually. Our TDBMS can be used to alleviate this problem by partially automating this task. Table 2 shows top distinct results of comparing *MySQL* and *PostgreSQL* database servers by supported index types (ranking is described in Section 4). This is done by joining the entity *Storage engine* in the text database generated from the *MySQL* manual with the entity *PostgreSQL* from the database generated from the *PostgreSQL* manual. Clearly, this is not an exhaustive comparison of two database servers. However, it does demonstrate what can be done *automatically*, without human intervention to help resolve this complex problem.

As a general algorithm operating on concepts extracted from text *fuzzy join* can be used to automatically compare any concepts from any text. For example, it also can be applied to compare skills in resumes, product features in manuals, or concepts in research papers.

Finally, we applied the *concept matching* algorithm to match similar concepts expressed by different words in two database servers manuals. For instance, the following concepts were detected to be similar even though they do not have any textual similarity: *command* and *statement*, *you* and *user*, *section* and *chapter*, *storage engine* and *PostgreSQL*.

Similarly to *fuzzy join*, *concept matching* is a general algorithm operating on extracted concepts and it can be applied to any text. More generally, it can be used as a document similarity *semantic metric*.

The rest of the paper is structured as follows. Section 2 describes automatic schema extraction algorithm and experimental results. Structural search and intuition behind the algorithm is described in Section 3. Automatic comparison, *fuzzy join* and ranking of join-results are in Section 4. Our concept matching algorithm and experimental results are described in Section 5 in more detail. Section 6 describes query language for the new TDBMS. We review related work in Section 7 and conclude in Section 8.

2. AUTOMATIC SCHEMA EXTRACTION

We use natural language processing to parse the *MySQL* and *PostgreSQL* manuals and incrementally construct two separate schemas using a state-of-the art English sentences parser.

The grammar in Figure 1 shows the parsing algorithm. Each sentence S is split into an arbitrarily long sequence of N noun phrases and V verb phrases. For instance, the

$$S \rightarrow NVS \mid N \mid \epsilon$$
$$V \rightarrow \mid \epsilon$$

Figure 1: Sentence parsing grammar

sentence "MySQL supports indexes widely used to improve performance" will be split into the sequence"MySQL[N_1], supports[V_1], indexes[N_2], used to improve[V_2], performance[N_3]". After that, we load all the parsed sentences into the table T with columns $(N_1, V_1, N_2, V_2, ...)$ in a relational databases. Each text file is processed separately and is loaded in a separate table.

To start schema extraction, we notice that the first column in T usually contains the subject of a parsed sentence (MySQL in the example above). Clearly, it is not always true, e.g. for questions or complex sentences, but it is still more the rule than the exception. We leverage this to retrive the main *concepts* (sentences' subjects) of the documents loaded into T by taking the most frequent values from $T.N_1$.

A better metric, called *concept weight* counts only distinct sentences for a given subject. This is a stricter metric, because it ignores all sentences that have the same predicate and object for a given subject and therefore promotes the concept only for participating as a subject in substantially different sentences. Table 3 illustrates the main concepts extracted from two database servers' user manuals, sorted by *concept weight*. Notice, that all top concepts are exactly what one would expect for a database server manual. Moreover, the lists were generated independently from two different text databases (each populated from a manual, excluding noise words), but the top concepts are very similar (see Table 3).

We further construct the concept structure by extracting the most frequent actions it can perform and defining them to be its attributes. They can be extracted from T by taking the most frequent values of $T.V_1$ for a given concept in $T.N_1$. For instance, the action *allows* occurred 7 times as a predicate in the sentences where MySQL was the subject. Similarly to *concept weight* discussed above a better metric is to count the number of distinct objects that appear in the sentences with a given subject and predicate. This is a better metric (define it to be *attribute weight*) than predicate frequency, because it ignores the sentences that have the same predicate and object for a given subject. Table 4 illustrates the extracted structure of two concepts *MySQL* and *You* sorted by *attribute weight*.

Concept	Weight	Concept	Weight
you	350	you	990
we	105	we	142
function	50	MySQL	110
PostgreSQL	42	server	58
option	34	table	57
query	24	statement	50
table	22	option	42
command	21	value	40
file	19	Innodb	38
server	19	file	33
user	19	function	31
application	18	variable	25
value	17	column	23
system	17	section	22
database	16	query	20
index	15	client	19
frontend	14	slave	18
view	14	user	18
column	14	mysqld	18
row	13	privilege	18
...	...	row	17
		...	...

Table 3: MySQL and PostgreSQL concepts

The set of structured concepts inferred from parsed text using the algorithms above is the *schema* for our text database. Notice, that no data is stored in the inferred schema. Parsed sentences are in table $T(N_1, V_1, N_2, V_2, ...)$ in the relational database.

Also, notice that this is a very general algorithm, since it relies only on the natural language sentence structure and does not depend on any specific terms, words or patterns [6]. It is also *not* restricted to any specific area of interest and therefore works for text on any topic.

3. STRUCTURAL SEARCH

Below we describe how we can do structural search by leveraging inferred schema.

Consider the problem to find in MySQL manual (automatically) what InnoDB (a MySQL storage engine) supports. One of the approaches would be to do keyword search through the manual on 'InnoDB' + (*provide* or *support* or *offer*). This returns near 50 sentences, $\approx 35\%$ of which do not match the focus of structured query (e.g."You can omit these command lines if you to not require InnoDB or BDB support").

On the other hand, we can use the schema extracted from MySQL manual and issue a selection query on entity *InnoDB* and its attribute *support*. It will be automatically mapped by our engine into a *select* statement on the underlying relational table $T(N_1, V_1, N_2, V_2, ...)$ and retrieve the sentences that contain *InnoDB* as a subject and *provide* or *support* or *offer* or their synonyms and derived forms as a predicate. This returns 26 sentences, three of which are shown in Table 1. We missed only 4 useful sentences (e.g. 'Support for XA transactions is available for the InnoDB storage engine'), but filtered out ≈ 20 that do not match the focus of structured query. Thus, structural search performs more focused and therefore more precise at the expense of coverage.

You	Weight	MySQL	Weight
have	34	uses	40
can use	23	supports	21
must own	14	is	10
need	14	has	7
must have	10	converts	7
can create	8	does not support	6
can do	8	can use	6
get	6	allows	6
want	6	creates	5
write	6	provides	4
will need	4	handles	4
...	...	...	...

Table 4: Concept structure

4. AUTOMATIC COMPARISON

Consider another problem of comparing two concepts from text. For example, let us compare *Storage engine* from the MySQL manual and *PostgreSQL* from the PostgreSQL manual. Similarly to the previous section, as a first approach, consider doing keyword search by *Storage engine* over the MySQL manual and by *PostgreSQL* over the PostgreSQL manual and matching all the resulting sentences. There should be good matches, but it is very hard to find them in a large result set even with high quality sentence matching and further ranking. This is because it is important for matching *where* in the sentence the matching terms occur.

The first important (and probably obvious) observation is that by the nature of language sentences are about similar concepts if their *subjects are similar* or mean similar concepts (see the next section on how to detect similar concepts expressed using different words). The next important observation is that if two sentences, in addition to having similar subject, have the same or similar predicate in common, they should match even better, because they are about a similar concept that does a similar action. So, how can we leverage this inferred structure to filter them out and match better?

To do much more precisely than just using keywords, consider selecting the sentences only with the subject containing *storage engine* for the MySQL manual and *PostgreSQL* for the PostgreSQL manual and predicate containing verbs *is, support, implement, has, can* or their synonyms and derived grammatical forms (e.g. *are, provide(s), allow(s), ...*). This significantly reduces the number of retrieved sentences and makes both result sets *strictly focused* on these two concepts performing specified actions.

Next, we describe two algorithms for fuzzy *join* between these two sentence sets and a *ranking* function to sort join results.

4.1 Fuzzy join and ranking function

To match most similar sentences, consider tokenizing the sentences' *tails*[2] for both sets into words and then extracting stems from these words. This results in a set of stems for each sentence in both sentence sets. After that, we *join* the sentences from both sets pairwise by matching the extracted stems for each sentence and sorting the join result by computed join rank. Only the best match for each sen-

[2]a sentence without subject, predicate and other verbs - $T.N_2 + T.N_3 + T.N_4 + ...$

tence is included in join result. Table 2 illustrates joining *Storage engine* from MySQL manual with *PostgreSQL* from PostgreSQL manual on *is, support, implement, has, can* and their derived grammatical forms. To further narrow the focus of comparison, we require the sentences to contain the keyword *index*. This guarantees the concepts are compared only on indices.

To rank join results we use the following ranking function:

$$joinrank(s_1, s_2) = \sum_{i=1}^{n} k_{t_i} \cdot \frac{1}{weight(t_i)}$$

where s_1, s_2 are the joined sentences, n is the number of tokens in one of the sentences tails (assume w.l.o.g. s_2), t_i is the token, k_{t_i} is the number of matches that generates t_i^{th} stem, weight(t_i) is t_i^{th} *concept weight* computed during automatic schema extraction.

This formula conveys the idea that the matches of more specific concepts are more valuable than those of general ones (therefore weight(t_i) is in the denominator). In addition, it encourages multiple matches of the same term(or stem) by multiplying the inverse weight by the number of successful matches. For example, the top row in Table 2 has the following rank: $3 \cdot 1/3 + 3 \cdot 1/1 + 1 \cdot 1/5 = 4.2$. Since *B-tree* from s_2 tail matched b-tree from s_1 three times and the concept b-tree has concept weight 3, therefore the first sum item is $3 \cdot 1/3$. The stem of *R-tree* tree, matched three times and *R-tree* has weight 1. *Hash* matched once and has weight 5, which sums up to 4.2.

The complexity of this join algorithm is $O(n^2)$ where n is the number of words in text. We can significantly improve performance by using full-text indexing on sentence tails by slightly sacrificing precision. Consider the same algorithm, which instead of counting the number of matches k_{t_i} does full-text index search on each token stem. This will raise the complexity to $O(log(m)) \cdot s_{max} \cdot s \leq O(log(n)) \cdot s_{max} \cdot s \leq O(log(n) \cdot \frac{n}{s_{min}}) \propto O(nlog(n))$ where m is the number of distinct indexed terms (whose upper bounded is the number of words n), s_{max} is the constant specifying the number of words in the longest sentence, s is the number of sentences to join (which has upper-bound $\frac{n}{s_{min}}$), n is the number of words in text, and s_{min} is the number of words in the shortest sentence.

The second algorithm performs much faster at the expense of slightly decreased accuracy, because full-text index lookup is unable to detect the number of matches. We compute joinrank for the second algorithm by using the same formula and setting $k_{t_i} \equiv 1, \ \forall i$ (optrank column in Table 2).

5. CONCEPT MATCHING

To detect similar concepts in text that are expressed using different words we present a concept matching algorithm that works on the extracted schema. For instance, it can detect that the concept *PostgreSQL* in the PostgreSQL manual is similar to *MySQL* and *InnoDB*[3] from the MySQL manual even though there is no textual similarity between them.

The basic intuition behind the algorithm is that two concepts are similar if they do similar actions on similar objects. In terms of inferred schema (see Table 4) this implies that:

- the more attributes two concepts have in common (e.g. *PostgreSQL* and *MySQL* both have *supports* attribute)

[3]MyISAM, InnoDB, Memory are MySQL storage engines

Concept$_1$	**Concept$_2$**	**Sim**
section	chapter	4.62
MySQL	PostgreSQL	2.75
you	user	1.75
server	PostgreSQL	1.66
InnoDB	PostgreSQL	1.54
statement	command	0.29
MaxDB	PostgreSQL	0.21
MySQL	query	0.14
...	...	

Table 5: General concept matching

Concept$_1$	**Concept$_2$**	**Sim**
Storage engine	PostgreSQL	0.12
MyISAM	PostgreSQL	0.12
index	PostgreSQL	0.036
table	PostgreSQL	0.036
column	PostgreSQL	0.036
user	PostgreSQL	0
MySQL	PostgreSQL	0
...	...	

Table 6: Focused concept matching

- the more similar objects in the original sentences correspond to the common attributes (e.g. PostgreSQL supports *indexes* ...; MySQL supports *indexes* ...)

then the more similar the concepts are.

In addition, there is a difference between detecting generally similar concepts and detecting concepts that are similar in some specific way (*focused* concept matching). Consider comparing indexes in PostgreSQL and MySQL by using *fuzzy join* between concepts *PostgreSQL* and *MySQL*. The result will be a table similar to Table 2, however it will contain many fewer rows. This is because supported index types in MySQL server depend on the storage engine[3], whereas PostgreSQL does not support multiple storage engines. Therefore the majority of sentences about index types have a specific storage engine as a subject instead of MySQL. Therefore, the concept *PostgreSQL* has more in common with the concept *Storage engine* than with *MySQL* for purposes of comparing indexes, whereas in general it is more similar to *MySQL*.

Briefly, the algorithm works by iterating over the concept list (Table 3) and accumulating (for each concept) all distinct predicates that appear in the sentences containing a concept in the subject. After that it computes pairwise *concept similarity* using the following similarity function:

$$sim(c_1, c_2) = m \cdot \sum_{i=1}^{m} \frac{1}{w(a_i)} + n^2 \cdot \sum_{i=1}^{m} \frac{\log \frac{w_{max}}{w(a_i)}}{\sum_{j=1}^{n(i)} w(o_{ij})}$$

where m is the number of distinct attributes a_i two concepts c_1, c_2 have in common, n is the number of common distinct objects, and $n(i)$ is the number of distinct common objects for i^{th} common attribute.

The first summand is a sum of inverse weights of concepts' common attributes multiplied by the number of distinct common attributes. Having a rare attribute in common is more important than having a frequent one. This is why *inverse* weights are summed.

If in addition to the attribute (such as *supports* in the example above), the concepts have a common object (e.g. *indexes*), it is taken into account by second summand, which is a sum of inverse weights of distinct common objects. According to the experiments, it is considered to be quadratically more important than having common attributes and is reflected by multiplying the second sum by n^2.

The absolute value how much a common object contributes to concept similarity also depends on the attribute to which the common object corresponds. This is the intuition behind why all sum members are weighted by $\log(w_{max}/w(a_i))$. How important for concept similarity an attribute is determined by the ratio of the absolute maximum attribute weight w_{max} and the attribute weight $w(a_i)$. *log* is used to reduce potentially large absolute values for rare objects. Several similar concepts are shown in Table 5 as an example of applying this metric.

Focused concept matching works similarly, except it accumulates for each concept all the distinct predicates that appear in the sentences containing a concept in the subject and a specific *focus* keyword in the object. The results of concept matching focused on indexes (focus keyword is *index*) are in Table 6.

Our prototype caches concept similarity and suggests similar concepts to the user for each *join* query. For example, for a query *PostgreSQL fuzzy join MySQL on index* along with the join result it will output as a suggestion top similar concepts for both *PostgreSQL* and *MySQL* resulted from focused matching (i.e. storage engine, MyISAM).

6. QUERY LANGUAGE

In this section we describe the query language of the new text database management system (TDBMS) prototype. Its schema consists of entities with attributes. Table 4 illustrates two sample entities *You* and *MySQL* and their attributes.

Two types of queries are currently supported - *selection* and *fuzzy join* queries. Both of them return a set of sentences as a result set. The result set for *fuzzy join* queries is sorted by join rank (described in Section 4.1) in descending order. Everything that is in square brackets is optional.

- $e_1[, .., e_n]/[a_1, .., a_m]$ [where K]
- $e_1[, .., e_n]/a_1[, .., a_m]$
  fuzzy join $e_1[, .., e_k]/a_1[, .., a_l]$ [on K]

The first one is a *selection* query on entities e_1, .., e_n and their attributes a_1, .., a_m or its synonyms and grammatical forms. As an example consider the query $InnoDB/supports$ and its result set in Table 1. A *selection* query on several entities and/or attributes returns the union of results of selection queries from all the entities and attributes involved. For instance, the query $MyISAM, InnoDB/requires, allows$ will return all the sentences containing $MyISAM$ or $InnoDB$ in the subject and $requires$ or $allows$ (and its synonyms and grammatical forms) in the predicate. If *where* K clause is specified the result set is filtered by requiring all the sentence tails[4] to contain the set of keywords K.

The second one is a *fuzzy join* query. First, it will select two sentence sets according to the semantics of the *selection* queries. If K - a set of keywords is specified, all the sentences in both intermediate result sets will be restricted to contain K in their tails (equivalent to specifying *where* K clause for both participating selection queries). Finally, it will execute *fuzzy join* algorithm between two intermediate result sets and sort the output by join rank in descending order. As an example, consider the query
$Storage\ engine/supports, is, implements$ fuzzy join $PostgreSQL/supports, is$ on *index* and its result set in Table 2.

[4]a sentence without subject, predicate and other verbs - $T.N_2 + T.N_3 + T.N_4 + ...$

7. RELATED WORK

In [6] Brin introduced an algorithm to extract tuples from the Web that are similar to a small"training set" of pairs (e.g. $\ll author, title \gg$). The basic idea was to match a given set of tuples against web pages to generate a general pattern that can be further bootstraped to retrieve more results. In [3] Agichtein extended this idea by using named-entity tagging (e.g. location, organization), weighting and confidence metrics to compose better patterns. Downey et al. in [11] suggested using learned patterns as extractors and discriminators to improve both coverage and accuracy of information extraction. Banko et al. in [5] described a question-answering system reformulating the specific questions into likely substrings of declarative answers to the question and submitting them to Google. E.g. for "Where is the Louvre Museum located" the reformulations will be"+the Louvre Museum +is located","+the Louvre Museum +is +in", "+the Louvre Museum +is near", etc. Ask.com is, probably, the most widely known commericial question-answering system that also works by reformulating specific questions and matching the resulting phrases against the Web.

Our approach is substantially different as it is neither restricted to a specific pattern format, nor aims to extract a specific relation or answer a specific question.

Etzioni et al. in [13] uses a more general approach to extract hyponym relation from Web pages by using domain independent patterns (cf. Hearst [14]). Crescenzi in [9] tries to generalize wrapper generation by matching dynamically generated HTML pages of data-intensive web sites (e.g. online book stores) and approximating the underlying database schema. Mindnet [17] is, probably, the most general system to automatically construct an approximation of a semantic network [10] that is a graph representing semantic relationship between words. [12] is a question-answering system capable to construct a *meta-repository* of *semantic objects* either automatically by extracting triples of form $\ll N, V, N \gg$ from text or manually through the user interface. It is able to match user requests to its *semantic objects* and output relevant results. In [8] Cafarella et al. build a large-scale (from 90-million Web-page corpus) extraction graph from triples similar to [12] and show experimental results of keyword-based spreading-activation search with depth 1 over this graph.

Our approach is substantially different from [17], [12], and [8], because instead of constructing a semantic network, meta-repository or extraction-graph, we first extract the main concepts from text and infer *schema* that is built around them reflecting their structure. Therefore, we are able to support *structural* search, *join* as well as concept matching that are not available in either [17], [12] or in [8].

[15] is an exhaustive survey of typical web data extraction

approaches and tools classified into six groups: Language for wrapper development, HTML-aware tools, NLP-based tools, Wrapper induction tools, modelling-based tools, and ontology-based tools. Finally, [2] is a search-engine that leverages NLP to resolve part-of-speech-, phrasal-, and contextual ambiguity and provide better search experience.

8. CONCLUSION

In this paper we presented a new text database management system (TDBMS) based on novel algorithms to automatically extract schema from text, perform *fuzzy join* between extracted entities, and detect similar semantic concepts expressed using different words (Table 5, Table 6). We applied it to perform powerful structural search (Table 1) and automatic software comparison (Table 2). Our experimental results justify that structural search is more focused and therefore performs more precisely than keyword search at the slight expense of coverage. Demonstrated results of automatic software comparison can be used by end-users, software consultants, and large enterprises that commonly have to choose between software products.

Finally, all the presented algorithms are very general because they operate on concepts extracted from text. Therefore, *fuzzy join* can be used to automatically compare concepts from any text, e.g. to compare skills in resumes, product features in manuals, or concepts in research papers. Similarly, *concept matching* can be used as a document similarity *semantic metric*.

9. REFERENCES

[1] http://www.google.com/corporate/tech.html.

[2] The infocious web search engine: Improving web searching through linguistic analysis. *Infocious Inc.*, 2005.

[3] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *ACM DL*, 2000.

[4] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[5] M. Banko, E. Brill, S. Dumais, and J. Lin. Askmsr: Question answering using the worldwide web. In *EMNLP*, 2002.

[6] S. Brin. Extracting patterns and relations from the world wide web. In *EDBT*, 1998.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[8] M. Cafarella, M. Banko, and O. Etzioni. Relational web search. Technical Report UW-CSE-06-04-02, 2006.

[9] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001.

[10] F. Crestani. Application of spreading activation techniques in informationretrieval. *Artif. Intell. Rev.*, 11(6):453–482, 1997.

[11] D. Downey, O. Etzioni, S. Soderland, and D. Weld. Learning text patterns for web information extraction and assessment. In *AAAI*, 2004.

[12] M. Elder. Preparing a data source for a natural language query. *United States Patent Application 20050043940*, 2004.

[13] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Web-scale information extraction in knowitall. In *WWW*, 2004.

[14] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. Technical Report S2K-92-09, 1992.

[15] A. Laender, B. Ribeiro-Neto, A. Silva, and J. Teixeira. A brief survey of web data extraction tools. In *SIGMOD Record*, 2002.

[16] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD Record*, 1979.

[17] L. Vanderwende, G. Kacmarcik, H. Suzuki, and A. Menezes. Mindnet: An automatically-created lexical resource. In *HLT/EMNLP*, 2005.