IJDCST @October Issue- V-1, I-6, SW-46
ISSN-2320-7884 (Online)
ISSN-2321-0257 (Print)

Information Extraction Based Stanford Dependencies Using Relational Databases

M.Suryanagamani[1], K.Kiran Kumar[2]

[1]Student, Nova College of Engineering and Technology for Women, Ibrahimpatnam,Krishna Dist, Andhra Pradesh, India

[2] Assistant Professor, Nova College of Engineering and Technology for Women, Ibrahimpatnam,Krishna Dist, Andhra Pradesh, India

Abstract: Commercial relational database management systems (RDBMSs) generally provide querying capabilities for text attributes that incorporate state-of-the-art information retrieval (IR) relevance ranking strategies, but this search functionality requires that queries specify the exact column or columns against which a given list of keywords is to be matched. We describe a novel approach for information extraction in which extraction needs are expressed in the form of database queries, which are evaluated and optimized by database systems. Using database queries for information extraction enables generic extraction and minimizes reprocessing of data by performing incremental extraction to identify which part of the data is affected by the change of components or goals. In this we will introduce the natural language parser named as Stanford Parser. The SD representation has seen considerable use within the biomedical text mining community. It has been used to give a task relevant evaluation scheme for parsers and as a representation for relation extraction

Keywords: keyword search, relational database, information retrieval, Text mining, query languages, information storage and retrieval, Stanford Parser.

I. INTRODUCTION

Commercial RDBMSs generally provide querying capabilities for text attributes that incorporate state-of-theart information retrieval (IR) relevance ranking strategies. This search functionality requires that queries specify the exact column or columns against which a given list of keywords is to be matched. For example, a query: *SELECT * FROM Complaints C WHERE CONTAINS (C.comments, 'disk crash', 1) > 0 ORDER BY score(1) DESC* on Oracle 9.1 1 returns the rows of the *Complaints* table that match the keyword query [disk crash], sorted by their *score* as determined by an IR relevance-ranking algorithm. Intuitively, the score of a tuple measures how well its *comments* field matches the query [disk crash]. The requirement that queries specify the exact columns to match can be cumbersome and inflexible from a user perspective: good answers to a keyword query might need to be "assembled" –in perhaps unforeseen ways– by joining tuples from multiple relations: IE is typically seen as a one-time process for the extraction of a particular kind of relationships of interest from a document collection. IE is usually deployed as a pipeline of special-purpose programs, which include sentence splitters, tokenizes, named entity recognizers, shallow or deep syntactic parsers, and extraction based on a collection of patterns.

A key contribution of this paper is the incorporation of IR-style relevance ranking of tuple trees into our query processing framework. In particular, our scheme fully exploits single-attribute relevance-ranking results if the RDBMS of choice has text-indexing capabilities (e.g., as is the case for Oracle 9.1, as discussed above). By leveraging state-of-theart IR relevance-ranking functionality already present in modern RDBMSs, we are able to produce high quality results for free-form keyword queries. For example, a query [disk crash on a netvista] would still match the *comments* attribute of the first

Complaints tuple above with a high relevance score, after word stemming (so that "crash" matches "crashed") and stop-word elimination (so that the absence of "a" is not weighed too highly). Our scheme relies on the IR engines of RDBMSs to perform such relevance-ranking at the attribute level, and handles both AND and OR semantics.

Unfortunately, existing query-processing strategies for keyword search over RDBMSs are inherently inefficient, since they attempt to capture all tuple trees with all query keywords. Thus these strategies do not exploit a crucial characteristic of IR-style keyword search, namely that only the top 10 or 20 most relevant matches for a keyword query – according to some definition of "relevance"– are generally of interest. The second contribution of this paper is the presentation of efficient query processing techniques for our IR-style queries over RDBMSs that heavily exploit this observation. As we will see, our techniques produce the top-*k* matches for a query –for moderate values of *k*– in a fraction of the time taken by state-of-the-art strategies to compute all query matches. Furthermore, our techniques are *pipelined*, in the sense that execution can efficiently resume to compute the "next-*k*" matches if the user so desires. A natural language parser is a program that works out the grammatical structure of sentences, for instance, which groups of words go together (as "phrases") and which words are the subject or object of a verb. Probabilistic parsers use knowledge of language gained from hand-parsed sentences to try to produce the *most likely* analysis of new sentences. These statistical parsers still make some mistakes, but commonly work rather well.

II. RELATED WORK

In database research, there has been some work on ranked retrieval from a database. The early work considered vague/imprecise similarity-based querying of databases. The problem of integrating databases and information retrieval systems has been attempted in several works. Information retrieval based approaches have been extended to XML retrieval. Keyword-query based retrieval systems over databases have been proposed in various papers. SQL extensions in which users can specify ranking functions via soft constraints in the form of preferences. The distinguishing aspect of our work from the above is that we espouse automatic Extraction of PIR-based ranking functions through data and workload statistics.

DBXplorer and DISCOVER exploit the RDBMS schema, which leads to relatively efficient algorithms for answering keyword queries because the structural constraints expressed in the schema are helpful for query processing. These two systems rely on a similar architecture, Unlike DBXplorer and DISCOVER, our techniques are not limited to Boolean-AND semantics for queries, and we can handle queries with both AND and OR semantics. In contrast, DBXplorer and DISCOVER (as well as BANKS) require that all query keywords appear in the tree of nodes or tuples that are returned as the answer to a query. Furthermore, we employ ranking techniques developed by the IR community, instead of ranking answers solely based on the size of the result as in DBXplorer and DISCOVER. Also, our techniques improve on previous work in terms of efficiency by exploiting the fact that free-form keyword queries can generally be answered with just the few most relevant matches. Our work then produces the "top-*k*" matches for a query fast, for moderate values of *k*.

The lexicalized probabilistic parser implements a factored product model, with separate PCFG phrase structure and lexical dependency experts, whose preferences are combined by efficient exact inference, using an A* algorithm. Or the software can be used simply as an accurate unlexicalized stochastic context-free grammar parser. Either of these yields a good performance statistical parsing system. A GUI is provided for viewing the phrase structure tree output of the parser.

The architecture of our query processing system relies whenever possible on existing, unmodified RDBMS components. Specifically, our architecture (Figure 3) consists of the following modules:

A) IR Engine

As discussed, modern RDBMSs include IR-style text indexing functionality at the attribute level. The *IR Engine* module of our architecture exploits this functionality to identify all database tuples that have a non-zero score for a given query. The *IR Engine* relies on the *IR Index*, which is an inverted index that associates each keyword that appears in the database with a list of occurrences of the keyword; each occurrence of a keyword is recorded as a tuple attribute pair. Our implementation uses Oracle Text, which keeps a separate index for each relation attribute. We combine these individual indexes to build the *IR Index*. When a query Q arrives, the *IR Engine* uses the *IR Index* to extract from each relation R the *tuple set* $RQ = \{t \in R \mid Score(t,Q) > 0\}$, which consists of the tuples of R with a non-zero score for Q. The tuples t in the tuple sets are ranked in descending order of $Score(t,Q)$, as required by the top-k query processing algorithms

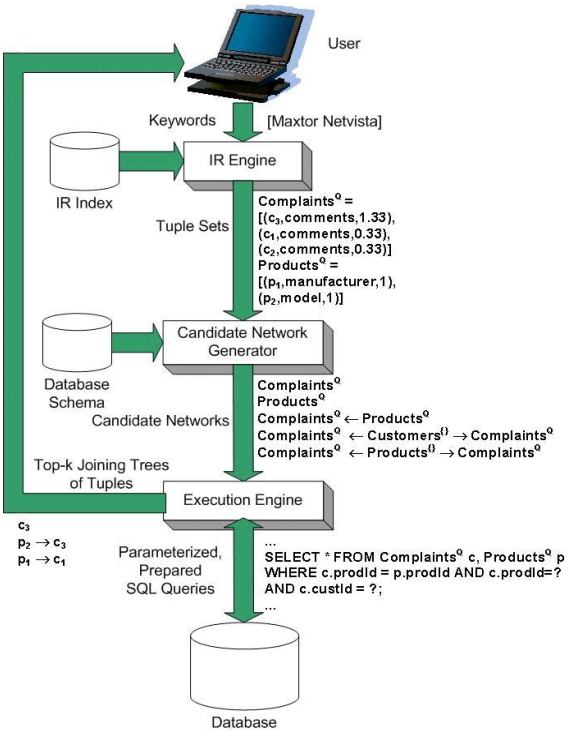

Figure 1: Architecture of our query processing system.

B) Candidate Network Generator

The next module in the pipeline is the *Candidate Network (CN) Generator*, which receives as input the non-emptytuple sets from the *IR Engine*, together with the database schema and a parameter M that we explain below. The key role of this module is to produce CNs, which are join expressions to be used to create joining trees of tuples that will be considered as potential answers to the query. Specifically, a CN is a join expression that involves tuple sets plus perhaps additional "base" database relations. We refer to a base relation R that appears in a CN as a *free tuple set* and denote it as $R\{\}$. Intuitively, the free tuple sets in a CN do not have occurrences of the query keywords, but help "connect" (via foreign-key joins) the (non-free) tuple sets that do have non-zero scores for the query. Each result T of a CN is thus a potential result of the keyword query.

We say that a joining tree of tuples T *belongs* to a CN C ($T \in C$) if there is a tree isomorphism mapping h from the tuples of T to the tuple sets of C. For example, in Figure 2, $(c1 \leftarrow p1) \in (Complaints\ Q \leftarrow Products\ (Q)$. The input parameter M bounds the size (in number of tuple sets, free or non-free) of the CNs that this module produces. The notion of CN was introduced in DBXplorer and DISCOVER. As discussed, DISCOVER and DBXplorer require that each joining tree of tuples in the query answer contain all query keywords. To produce all answers for a query with this AND semantics, these systems create *multiple* tuple sets for each database relation. Specifically, a separate tuple set is created for each combination of keywords in Q and each relation. This generally leads to a number of CNs that is exponential in the query size, which makes query execution prohibitively expensive for queries of more than a very small number of keywords or for values of M greater than 4 or so.

In contrast, we only create a single tuple set RQ for each relation R, as specified above. For queries with AND semantics, a post processing step checks that we only return tuple trees containing all query keywords. As we will see, this characteristic of

our system results in significantly faster executions, which in turn allows us to handle larger queries and also consider larger CNs. The CN generation algorithm is based on that of the DISCOVER system, and is not explained here in full detail due to lack of space. Conceptually, we first create the *tuple set graph* from the database schema graph and the tuple sets returned by the *IR Engine* module. We progressively expand each CN $s \epsilon S$ by adding a tuple set adjacent to *s* in the tuple set graph. We consider *s* to be a CN and hence part of the output of this module if it satisfies the following properties:

1. *The number of non-free tuple sets in s does not exceed the number of query keywords m:* This constraint guarantees that we generate a minimum number of CNs while not missing any result that contains all the keywords, which is crucial for Boolean-AND semantics. That is, for every result *T* that contains every keyword exactly once, a CN *C* exists such that $T \epsilon C$.
2. *No leaf tuple sets of s are free:* This constraint ensures CN "minimality 3. *s does not contain a construct of the form* $R \rightarrow S \leftarrow R$*:* If such a construct existed, every resulting joining tree of tuples would contain the same tuple more than once.

The size of a CN is its number of tuple sets. All CNs of size 3 or lower for the query [Maxtor Netvista] are shown in Figure 3.

C) Execution Engine

The final module in the pipeline is the *Execution Engine*, which receives as input a set of CNs together with the non-free tuple sets. The *Execution Engine* contacts the RDBMS's query execution engine repeatedly to identify the top-*k* query results. The *Execution Engine* module is the most challenging to implement efficiently.

D) Stanford Architecture

This package is a Java implementation of probabilistic natural language parsers, both highly optimized PCFG and lexicalized dependency parsers, and a lexicalized PCFG parser. The original version of this parser was mainly written by Dan Klein, with support code and linguistic grammar development by Christopher Manning.

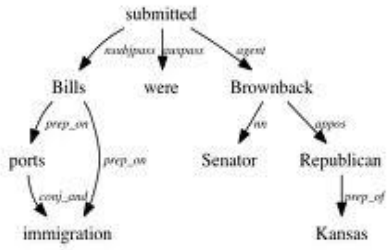

Figure 2: Data Representation in Stanford process.

Extensive additional work (internationalization and language-specific modeling, flexible input/output, grammar compaction, lattice parsing, *k*-best parsing, typed dependencies output, user support, etc.) has been done by Roger Levy, Christopher Manning, Teg Grenager, Galen Andrew, Marie-Catherine de Marneffe, Bill MacCartney, Anna Rafferty, Spence Green, Huihsin Tseng, Pi-Chuan Chang, Wolfgang Maier, and Jenny Finkel.

III. PARSE TREE QUERY LANGUAGE

However, the inability of expressing immediate following siblings and immediate-preceding siblings in this standard XML query languages. PTQL is an extension of the linguistic query Language that allows queries to be performed not only on the constituent trees but also the syntactic links between words on linkages. A PTQL query is made up of four components:

1. Tree patterns,
2. Link conditions,
3. Proximity conditions, and
4. Return expression.

A tree pattern describes the hierarchical structure and the horizontal order between the nodes of the parse tree. A link condition describes the linking

requirements between nodes, while a proximity condition is to find words that are within a specified number of words. A return expression defines what to return.

IV. STANFORD DEPENDECIES

Dependency is a one-to-one correspondence: for every element (e.g. word or morph) in the sentence, there is exactly one node in the structure of that sentence that corresponds to that element.

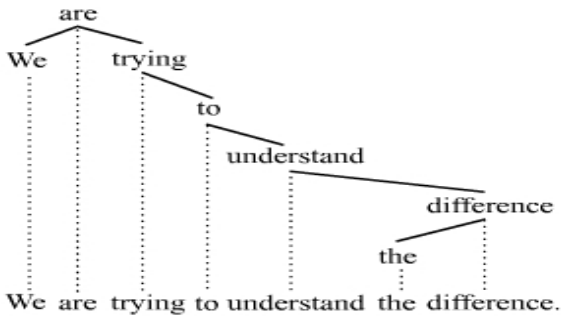

Figure 3: Dependency naturals in representation of natural parse architecture

The result of this one-to-one correspondence is that dependency grammars are word (or morph) grammars. All that exist are the elements and the dependencies that connect the elements into a structure. This situation should be compared with the constituency relation of phrase structure grammars.

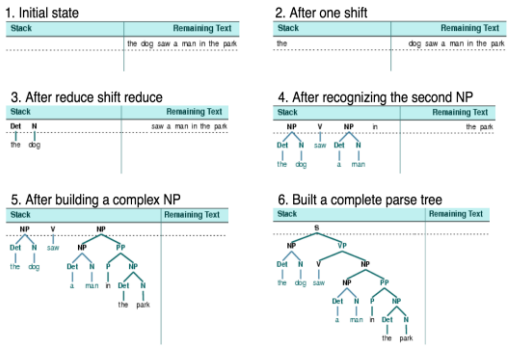

Figure 4: Stanford accessing data tree construction.

The dependencies are all binary relations: a grammatical relation holds between a governor (also known as a regent or a head) and a dependent. The grammatical relations are defined below, in alphabetical order according to the dependency's abbreviated name (which appears in the parser output). The definitions make use of the Penn Treebank part-of-speech tags and phrasal labels. abbrev: abbreviation modifier An abbreviation modifier of an NP is a parenthesized NP that serves to abbreviate the NP (or to define an abbreviation). "The Australian Broadcasting Corporation (ABC)" abbrev(Corporation, ABC)acomp: adjectival complement An adjectival complement of a verb is an adjectival phrase which functions as the complement (like an object of the verb).

V. PERFORMANCE RESULTS

Five variants of the typed dependency representation are available in the dependency extraction system provided with the Stanford parser. The representations follow the same format. In the plain text format, a dependency is written as abbreviated relation name(governor, dependent) where the governor and the dependent are words in the sentence to which a number indicating the position of the word in the sentence is appended.1 The parser also provides an XML format which captures the same information.

Figure 5: Stanford Dependencies results

The lexicalized probabilistic parser implements a factored product model, with separate PCFG phrase structure and lexical dependency experts, whose preferences are combined by efficient exact inference, using an A* algorithm. Or the software can be used simply as an accurate unlexicalized stochastic context-free grammar parser. Either of these yields a good performance statistical parsing system. A GUI is provided for viewing the phrase structure tree output of the parser.

VI. CONCLUSION

In this paper we presented a system for efficient IR-style keyword search over relational databases. A query in our model is simply a list of keywords, and does not need to specify any relation or attribute names. The answer to such a query consists of a rank of "tuple trees," which potentially include tuples from multiple relations that are combined via joins. A natural language parser is a program that works out the grammatical **structure of sentences**, for instance, which groups of words go together (as "phrases") and which words are the **subject** or **object** of a verb. Probabilistic parsers use knowledge of language gained from hand-parsed sentences to try to produce the *most likely* analysis of new sentences

VII.REFERENCES

[1] D. Ferrucci and A. Lally, "UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment," Natural Language Eng., vol. 10, nos. 3/4, pp. 327-348, 2004.

[2] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan, "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications," Proc. 40th Ann. Meeting of the ACL, 2002.

[3] A. Doan, J.F. Naughton, R. Ramakrishnan, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong, "Information Extraction Challenges in Managing Unstructured Data," ACM SIGMOD Record, vol. 37, no. 4, pp. 14-20, 2008.

[4] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan,and H. Zhu, "SystemT: A System for Declarative Information Extraction," ACM SIGMOD Record, vol. 37, no. 4, pp. 7-13, 2009.

[5] M. Huang, S. Ding, H. Wang, and X. Zhu, "Mining Physical Protein-Protein Interactions by Exploiting Abundant Features,"Proc. Second BioCreative Challenge, pp. 237-245, 2007.

[6] J. Hakenberg, C. Plake, L. Royer, H. Strobelt, U. Leser, and M. Schroeder, "Gene Mention Normalization and Interaction Extraction with Context Models and Sentence Motifs," Genome Biology, vol. 9, Suppl 2, p. S14, 2008.s