Answering Web Queries Using Structured Data Sources

Stelios Paparizos, Alexandros Ntoulas, John Shafer, Rakesh Agrawal
Microsoft Research, Search Labs
1065 La Avenida, Mountain View, CA 94043, USA
{steliosp, antoulas, jshafer, rakesha}@microsoft.com

ABSTRACT

In web search today, a user types a few keywords which are then matched against a large collection of unstructured web pages. This leaves a lot to be desired for when the best answer to a query is contained in structured data stores and/or when the user includes some structural semantics in the query.

In our work, we include information from structured data sources into web results. Such sources can vary from fully relational DBs, to flat tables and XML files. In addition, we take advantage of information in such sources to automatically extract corresponding semantics from the query and use them appropriately in improving the overall relevance of results.

For this demonstration, we show how we effectively capture, annotate and translate web user queries such as 'popular digital camera around $425' returning results from a shopping-like DB.

Categories and Subject Descriptors: H.2.8 Database Applications **General Terms:** Performance.

1. INTRODUCTION

Web search has successfully evolved over the past few years, from a carefully selected hierarchy of web page bookmarks, to a huge collection of crawled documents requiring sophisticated algorithms to identify the few most relevant results. Yet today's prominent solutions still leave a lot to be desired for certain classes of queries representing real user needs.

Consider for example a query such as 'popular digital camera around $425'. If we consider this query as a matching of words over unstructured text content, then only web pages containing these words will be returned to the user. However, in this case, cameras that are priced at, say, $410 or $430 will not be returned.

Performing such queries over free text is not necessarily straightforward, yet there are available appropriate structured data sources that contain information such as the camera and price pairs. Additionally, sorting the results based on IR-like textual relevance is not the best approach in this particular scenario. Here, the user is implying that he is interested in some notion of camera popularity. In such case, we need to perform a deeper analysis of the query in order to understand its semantics and return better results.

We should note that our camera example is not a rare hand-picked query applicable only to one domain. On the contrary, similar trends appear in other real user scenarios for a variety of domains. For example, users are looking for 'flights from san jose to seattle for march 3rd', 'movies with Pacino and De Niro', 'indiana jones 4 near san francisco', 'sauce recipes with tomato, basil but not cilantro', 'used compact cars 3-5 years old' and so forth. Such user queries tend to produce poor results in today's search engines.

In the past, there have been efforts of natural language processing or keyword search over databases that tried to addressed similar issues. But such approaches have problems due to some characteristics inherent in web search. Web users do not issue full language queries that can benefit from a linguistic analysis – for example, quite often, no verb is present. In addition, web users are not used in waiting long periods of time for results to appear – typical web response is under half a second. Finally, web users are not users sitting on one box with some partial knowledge of the underlying data, they simply ask questions about anything. These characteristics motivate and differentiate our work.

In this paper we present our approach, called Helix, that supports semantically rich web queries, incorporates results from multiple structured data sources and is efficient and easy to parallelize. We start with defining basic concepts, in Section 2, and then summarize our approach in Section 3. We give a walk-through of our proposed demo, in Section 4, and close with related efforts, in Section 5.

2. TOKENS AND PATTERNS

We start our discussion on our approach with establishing our basic primitives of *Token*, *Token Classes* and *Patterns*. **Token** is a sequence of characters. **Token Class** (TC) is a set of tokens described by a deterministic function. **Pattern** is a sequence of TCs.

Example *Tokens* are 'blue', 'Michael Jordan' and 'eos350'. Note that *Tokens* can contain white space characters. A *Token Class* can be described by a set of *Tokens*, i.e. $\langle basketballplayers \rangle$ = {'Michael Jordan', 'Magic Johnson', 'Larry Bird'} or it can be described by a regular expression, i.e. $\langle$ model$\rangle$ = 'eos'$\backslash d+$, where 'eos' is the matching string, $\backslash d$ a digit and $+$ denotes the matching of at least one digit. Finally, a *Pattern* example is: pPlayerScored = $\langle basketballplayers \rangle$ $\langle points \rangle$.

Furthermore, we classify *Token Classes* into four categories. **Universal:** generic mechanism describes them deterministically, i.e. number, date, time, location. **DataDriven:** generated from values of a given database column. **Inconsequential:** do not affect query meaning, i.e. for query 'what is the weather in Seattle', TC {'what', 'is', 'the'} is inconsequential for this context. **Modifiers:** alter how other *Token Classes* are processed.

Example of this classification, for 'popular digital camera around $425', 'digital camera' maps to a $\langle product \rangle$ *DataDriven* TC, '$425' to a $\langle price \rangle$ *Universal* TC, 'popular' and 'around' are *Modifiers*.

3. HELIX APPROACH

We built a system named Helix that utilizes *Tokens*, *Token Classes* and *Patterns* to effectively handle web queries over structured data. It uses an online component that is responsible for query annotation and handling and an offline that mines *Patterns* from query logs.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
ACM 978-1-60558-551-2/09/06.

3.1 Online Query Processing

Patterns, *Token Classes* and *Tokens* are given as input to the online phase and are utilized to do query annotation, routing and translation on the user queries as they were entered.

Query Annotation is achieved by first tokenizing each query and then performing segmentation using pattern matching.

Tokenization: The goal is to associate query words with *Tokens* in a meaningful way. To this end, all *Tokens* are combined into one large dictionary structure allowing fast lookups – a trie representation is used. We match words to the max possible *Token* size going left to right in a single pass.

Segmentation (or Pattern Matching): The goal is to break the query into meaningful pieces, annotating them with TCs. For each candidate *Pattern*, we map *Tokens* using an LR(1) parsing process – single lookahead, matching maximum sub-pattern left to right. This process is parallelized and all *Patterns* are kept in memory.

Note that due to the multiplicity of the TCs, a single *Pattern* captures a large number of queries during query annotation. The major advantages in using *Patterns* are the compact representation, small memory footprint and fast query analysis that we obtain. For example, ⟨brand⟩⟨productClass⟩ captures 'canon digital camera', 'sony digital camera', 'panasonic HDTV', 'HP printer' and so forth.

Routing aims to forward the query to data sources that can generate meaningful results. Since web search engines receive millions of queries daily, it would not be computationally efficient to simply send all queries to all data sources and perform a keyword match. So routing acts as a very selective filtering step that enhances overall performance. To achieve this, we maintain for each DataDriven TCs the corresponding DB. After the pattern match, we do a single lookup and route the query. Essentially pattern matching provides the routing capability 'for free', as no additional steps are required.

Translation takes place on the machine where the data lives and converts the user query to a system friendly evaluation expression. One way to perform the translation is to implement SQL rules for each of the *Patterns* used in the annotation. However, this is a cumbersome process as a few TCs can result in a big number of *Patterns*– factorial in the number of TCs. Instead, we have simplified the process using only a limited set of mappings into very few operations that we deemed necessary for our purposes. These operations are: i) Select(*column*), accesses *column* from specific data store, ii) Filter(*column*, *operand*, *value*), removes rows not satisfying the *operand* & *value* condition on the *column* entries and iii) iSort(*column*), indicates a sort intention on *column*.

Given such operations we can create mappings to perform generic translation rules for all patterns. DataDriven TCs map to a Select on a corresponding column, and so do Universal TCs. All we require is an operator entered mapping for every Modifier TC to a corresponding Filter, iSort or Select operation. This way, 'popular digital camera around $425' could be captured by a sequence of {Select(productClass), Filter(productClass, =, 'digital camera'), Filter(price, $>$, 375), Filter(price, $<$, 475), iSort(numOfReviews)}.

At a high-level, one can make the observation that only very few set of mappings from TCs to operations are necessary to capture a significantly large number of patterns as most patterns would have repeated combinations of the same TCs. Such mappings can be entered manually for a given domain.

3.2 Offline Pattern Mining

A key element of our approach is the use of *Patterns*, TCs and *Tokens* to perform the online query processing. One important question, however, is how we can obtain all the *Patterns* for a given domain. For this, we have developed a solution that takes as input the DataDriven and Universal *Token Classes* (TCs) and mines

```
Input: Set of queries and DataDriven & Universal TCs
Output: A set of patterns
Procedure
(1)  Tokenize queries using input TCs.
(2)  Parse remaining unknown words in query.
     Create singleton Token Classes by clustering multiple words
     based on their inter-query co-occurrence frequency.
(3)  Create primitive patterns by rewriting each query using TCs.
(4)  Break each primitive pattern into elementary sub-patterns.
     Use input TCs and special begin/end TCs as stop points.
(5)  Consider merging unknown tokens into single TC.
(6)  Use structural similarity amongst patterns to identify
     intra-query clusters.
(7)  Merge TCs according to frequency-based similarity
     (e.g. Jaccard distance of candidate TCs)
```

Figure 1: Outline for Pattern Mining

Patterns by analyzing millions of samples coming from query logs. We provide an outline of our algorithm in Figure 1.

We follow a bottom-up approach starting with the assumption that we operate on a given structured data source. Based on the data, we can identify the DataDriven (TCs) by simply selecting all entries on a DB column and removing duplicate values. Universal TCs are already available within the system as they are generic TCs applicable across domains (i.e. number, date, location). Using the DataDriven and Universal TCs we can process a number of queries, annotating only the known *Tokens* and creating new TCs for the unknown *Tokens*, essentially converting everything into primitive *Patterns*. Subsequent steps use both structural and frequency-based similarity functions to group *Patterns* while merging TCs by calculating the union of their *Tokens*. The end result is a series of *Patterns* rich in structural variations containing both given TCs as well as newly-learned ones. The overall process can be generalized, allowing us to learn *Patterns* from a limited number of query samples, and subsequently use them to capture a significantly larger number of queries during the online processing.

4. THE DEMONSTRATION

For our demonstration we will present both offline and online components of Helix. As the basis, we will use a structured data source containing information on consumer products described by a schema of products(productName, productClass, reviewRating, numOfReviews) – such data can be found publicly available in [10]). From this table we create the DataDriven TC ⟨productClass⟩ containing the set of duplicate free values of the corresponding DB column, an example instance is ⟨productClass⟩ = {'digital camera', 'HDTV', 'handbag', 'evening dress'}. We also use the Universal TC ⟨price⟩ described by the regular expression $'\$'\backslash d + ('.'\backslash d+)?$.

First, the off-line pattern generation process takes place, as described in Section 3.2. For simplicity we look at a small fraction of the overall process, using as input the limited set of queries shown in Figure 2(a) and the two aforementioned TCs ⟨productClass⟩, ⟨price⟩. The first step is to tokenize the queries using the input TCs and assign new TCs to the unknown words like 'near', 'for', 'popular'. The produced primitive *Patterns* are shown in Figure 2(b). Next, we group *Patterns* by using structural and frequency similarity functions producing the compressed representation shown in Figure 2(d). The unknown-word TCs got merged during this process into ⟨tcFAN⟩ and ⟨tcPMP⟩, as shown in Figure 2(c). The number of *Patterns* is reduced producing a more compact representation, whereas at the same time, the merged *Token Classes* (TCs) allow for an increase in online query coverage.

Having learned the *Patterns*, all we need are a few translation rules before we can proceed with online processing as described

digital camera for $200
handbag around $400
popular HDTV near $2000
most popular evening dress

(a) Sample queries

⟨productClass⟩ ⟨tcFor⟩ ⟨price⟩
⟨productClass⟩ ⟨tcAround⟩ ⟨price⟩
⟨tcPop⟩ ⟨productClass⟩ ⟨tcNear⟩ ⟨price⟩
⟨tcMostPop⟩ ⟨productClass⟩

(b) Primitive Patterns

⟨tcFAN⟩ = { ⟨tcFor⟩ ∪ ⟨tcAround⟩ ∪ ⟨tcNear⟩ }
⟨tcPMP⟩ = { ⟨tcPop⟩ ∪ ⟨tcMostPop⟩ }

(c) Merged Token Classes

⟨productClass⟩ ⟨tcFAN⟩ ⟨price⟩
⟨tcPMP⟩ ⟨productClass⟩ ⟨tcFAN⟩ ⟨price⟩
⟨tcPMP⟩ ⟨productClass⟩

(d) Compressed Patterns

Figure 2: Mining Patterns from Queries, an Example.

in Section 3.1. The following 3 rules are manually entered: 1) ⟨tcPMP⟩ → iSort(products:numOfReviews), 2) ⟨tcFAN⟩ ⟨price⟩ → Filter(products:price, <, 1.15*tokenOf(⟨price⟩)) AND Filter(products:price, >, 0.85*tokenOf(⟨price⟩)), 3)⟨productClass⟩ → Filter(products:productClass, =, tokenOf(⟨productClass⟩)).

We use all this information to execute the online query processing component as described in Section 3.2. We consider the web user query 'popular digital camera around $425'. First, tokenization occurs mapping 'popular' to ⟨tcPMP⟩, 'digital camera' to ⟨productClass⟩, 'around' to ⟨tcFAN⟩ and '$425' to ⟨price⟩. Then pattern matching associates the query to ⟨tcPMP⟩ ⟨productClass⟩ ⟨tcFAN⟩ ⟨price⟩. As determined from this match the query is routed where the *products* data store resides. Translation of the annotated query takes place following the given rules, resulting to the set of operations in {Select(products:productClass), Filter(products: productClass, =, 'digital camera'), Filter(products:price, <, 1.15*$425), Filter(products:price, >, 0.85*$425), iSort(products: numOfReviews)}.

Note that although for clarity we only use Filter and iSort operations here, our underlying engine is more sophisticated. Instead of a simple sort, there is a ranking function that combines weighted scores for price proximity to $425 in combination with number of reviews for a given camera.

The returned results are seen in Figure 3. We return some information about the camera model, the price and a review rating. Each product links to the corresponding detailed page. Although only three results are shown here, we provide the users with the capability to see multiple results on the same page with a simple click of a button, yet still showing everything within the familiar web search ecosystem.

As part of our live demonstration we will allow users to enter any web query they want using the familiar web query box interface. As the structured data source, we will connect with a shopping-like database (as in [10]). For the user queries, if data exists, our system will return results from the structured data store and integrate them with the results returned from a major web search engine. Pattern matching will occur with a large pre-generated lists of patterns, yet we will also allow users to give sample queries and built patterns locally as in Figure 2.

5. RELATED WORK

There has been previous work in the area of keyword search over structured and semi-structured data. The approaches in [2, 4, 8] present systems that allow users to issue keyword queries to DBMSs. In [2] the authors investigate different approaches in searching by joining tuples through the use of a keyword index (called symbol table) that points to the rows or columns containing a given keyword. In [4] the database tuples are modeled as a graph and keyword search is performed by locating Steiner trees in the graph. The work in [8] focuses on returning the top-k matches for a keyword query instead of computing all matches.

In [3] the authors discuss natural language interfaces that were aiming in proving alternate access to relational data bases via a user friendly text to SQL approach. More recently, there has been similar work that enabled text-based search ([5, 9]) over XML semi-structured data. In a separate line of work, [1] presents different storage schemes that facilitate keyword search for RDF/XML data with the column-oriented approach showing as the most promising.

For the cases where we need to deal with unstructured data [7] presents ways of incrementally extracting structured information from text and storing it in an DBMS for later querying. Similarly, [6] discusses the problem of identifying the most prominent entities within a set of Web pages and searching over them.

Our work differs from previous ones in that we focus on web queries and their correlation with structured data. In this setting, keyword search is not enough, as semantic knowledge of words is very important. On the other hand a linguistic analysis is problematic as web queries rarely look like full sentences. We attack this problem with a clever use of patterns mined via query logs.

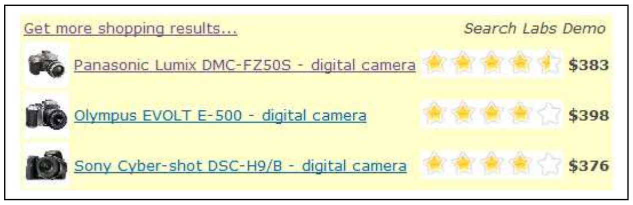
Get more shopping results... Search Labs Demo
Panasonic Lumix DMC-FZ50S - digital camera $383
Olympus EVOLT E-500 - digital camera $398
Sony Cyber-shot DSC-H9/B - digital camera $376

Figure 3: Results for 'popular digital camera around $425'

6. CONCLUSION

We have presented a system, called HELIX, that incorporates responses from structured data for web queries by analyzing and translating them using *Patterns*. Such *Patterns* are generated offline via query log mining and are continuously updated. *Patterns* give us a power to understand query semantics beyond that of keyword search without the complexity and fragility of NLP.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K.J.Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. VLDB Conf.*, 2007.
[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. *ICDE*'02
[3] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.
[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. ICDE Conf.*, 2002.
[5] C. Botev, J. Shanmugasundaram, and S. A. Yahia. A texquery-based xml full-text search engine. In *SIGMOD*'04.
[6] T. Cheng, X. Yan, K. Chen, and C. Chang. Entityrank: Searching entities directly and holistically. In *VLDB*, 2007.
[7] E. Chu, A. Baid, T. Chen, A. Doan, and J. F. Naughton. A relational approach to incrementally extracting and querying structure in unstructured data. In *Proc. VLDB Conf.*, 2007.
[8] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. *VLDB*'03.
[9] Y. Li, H. Yang, and H. V. Jagadish. Nalix: an interactive natural language interface for querying xml. *SIGMOD*'05.
[10] MSN Shopping. Public XML data api. http://shopping.msn.com/xml/v1/getresults.aspx?text=camera.