Proactive Natural Language Search Engine: Tapping into Structured Data on the Web

Wensheng Wu
University of North Carolina at Charlotte
w.wu@uncc.edu

ABSTRACT

In this era of "big data", a key challenge facing the database community is to help average users tap into the huge amounts of structured data on the Web. To address this challenge, we propose a novel proactive template-based engine for searching structured data on the Web using natural language. Departing from conventional search engines, the proposed engine organizes questions it can answer using templates and figures out ahead of time which sources can answer which templates and how. Then, at query time, the engine can simply match queries with the templates and retrieve answers using the pre-compiled evaluation plans. While attractive, building such an engine requires innovations in template creation, query evaluation, and system evolution. In this paper, we propose novel techniques to address these challenges.

Categories and Subject Descriptors

H.2.5 [**Database Management**]: Heterogeneous Databases

General Terms

Design, Language, Algorithms

Keywords

Web data, proactive search engine, natural language queries

1. INTRODUCTION

Huge amounts of structured data are becoming available on the Web [10]. They come from a large number of diverse sources and cover a great variety of subjects, from business, science, government, to entertainment. Indeed, the number of Web databases with form-based query interfaces (i.e., the *Deep Web*) alone has already exceeded 25 millions [10]. In addition, recent advances in information extraction have greatly facilitated the extraction of structured data from web pages and texts [15], which further contributes to the explosion of structured data on the Web.

Current web search engines have popularized the keyword-style search interface, where users can simply enter keywords in a single box to search for relevant documents all over the Web. So a natural question is: *is it feasible to use the same keyword interface to search for the information in the structured data?* The *key* issue to be addressed is the *ambiguity* of keyword queries [5]. For example, "movies clint eastwood" could mean "movies that Clint Eastwood played in" or "movies directed by Clint Eastwood" or both.

Traditionally, structured query languages such as SQL and XQuery are used to formulate *precise* queries against structured data such as relational tables and XML documents. However, writing queries in such languages requires substantial skills and great familiarity with the schema of data, which are unrealistic for *average* web users.

Instead, many sources provide more user-friendly form-based query interfaces [17], where users can specify query conditions using a select set of attributes. However, in order to accommodate query needs from different users, these interfaces can become very complex. For example, NSF [12] award search interface has over 30 fields, covering varied questions on PIs, programs, and projects.

Furthermore, users often need to combine the information from multiple sources, e.g., for comparison shopping. However, due to the autonomous nature of sources, their interfaces are designed independently based on projected query needs and the content of sources. As a result, query interfaces of different sources, even in the same domain, may be very different. This in effect means that users would need to learn a new query language for every source they visit.

A possible solution is to build a global query interface for each domain of interest by merging query interfaces of sources in the domain [17]. However, such a global interface is likely even more complex, with a large number of query attributes combined from different sources. As a result, it might only be suitable for sophisticated users.

So the question remains whether there is a universal query language for structured data that is easy to learn for average web users and at the same time reduces the ambiguity of keyword queries. An obvious choice is the natural language: human being is the master of natural language, so there is no new language to learn. Meanwhile, natural language queries can be much more precise than keyword queries [9]. For example, a natural language query "what movies were directed by Clint Eastwood" clearly states that user is looking for movies that Eastwood directed. Being able to pose queries in natural language also removes from users the hassle of deciphering form-based source interfaces.

Challenges: However, there are several *serious* challenges to be addressed when building a search engine to support natural language queries over the structured data on the

EDBT 2013, March 18–22, 2013, Genoa, Italy.

Web. First, *query parsing:* Before a query can be processed, it needs to be parsed to identify its intent [13]. Natural language parsing is an extremely difficult problem, largely due to the richness of the language [8]. Many parsers have been developed, but they are still too brittle to reliably perform a full parse of arbitrary natural language texts [7]. Furthermore, parser should also be able to recognize variations in query expressions. For example, both "movies clint eastwood has directed" and "list films directed by clint eastwood" search for movies directed by Clint Eastwood.[1]

Second, *query processing:* After a user issues a query, the engine must quickly obtain answers from sources. Here the challenge arises from the sheer *scale* and *heterogeneity* of sources [6]. For example, consider the above query on the movies directed by Eastwood. The system must quickly determine which sources contain the information on movies and directors, how to transform the query into the format understood by the sources (e.g., query form at Internet Movie Database or IMDb), what is the best plan for obtaining answers from sources (e.g., retrieving from authoritative ones first), and how to combine the answers.

Third, *keyword search:* Although natural language is a more natural and precise way of expressing queries, research indicates that web users tend to pose short keyword queries (2-3 words) [8]. This is largely due to the lack of support from search engines. In fact, natural language queries are popular among novice users before they learn the limitations of search engines [8]. Nevertheless, since keyword queries may be regarded as an abbreviated (and often ambiguous) version of their natural language correspondents, the engine should also be able to interpret and disambiguate keyword queries. For example, it may interpret "star wars director" as "who is the director of the movie Star Wars".

Solution: To address these challenges, we propose a novel *proactive template-based* approach to building a natural language search engine for the structured data on the Web. Departing from existing search engines, the proposed engine organizes questions it can answer using templates and plans ahead on which sources may be used to answer which templates and how. Then, at query time, it matches queries with the templates, instantiates parameters in the templates with values extracted from the queries, and retrieves answers using the pre-compiled evaluation plans for the templates.

To illustrate the approach, consider a template T: "what movies has director directed", where the parameter director takes director names as values. Further consider two movie sources: FilmsAndTV.com, which has only movies up to 2010; and IMDb, which has more recent movies but does not have movies before 1950. Then one way of evaluating the queries in T is to retrieve movies directed by the given director from both sources and combine the results. Based on this plan, a query "what movies has Clint Eastwood directed" can then be answered by simply executing an instantiated plan whose director is set to "Clint Eastwood".

Advantages: The proposed approach has several key *advantages*. First, recent study [1] shows that web queries often follow certain patterns and a large number of queries in a given domain may be captured using just a few templates. Thus, a proactive engine as proposed, even with only a few popular templates, may already be able to serve a large number of web users, providing far more accurate answers than the current keyword-based search engines.

Second, the approach turns complex query parsing into a simpler template matching problem: instead of parsing queries using the entire natural language grammar, queries are matched with the templates, each being a small grammar with *well-defined* syntax and semantics. Since template grammars are much simpler, the approach effectively circumvents the difficulties in query parsing.

Third, answering queries over a large number of diverse sources is an extremely challenging problem and has been extensively studied in data integration [6]. Distinct from existing online solutions, the proposed proactive engine does most of the hard work offline on answering template queries, which helps greatly reduce response time at query time.

Finally, templates may be used to help disambiguate keyword queries. For example, consider a keyword query "movies clint eastwood" and two templates: "what movies has director directed" and "what movies has actor starred in". Knowing Clint Eastwood has been both an actor and a director, the engine may use the templates to suggest possible natural language refinements to the original query, e.g., do you mean "what movies has Clint Eastwood starred in"?

Contributions: While attractive, building a proposed engine requires innovations in template creation, query evaluation, and system evolution. In the rest of this paper, we propose novel techniques to address these challenges.

- **Template creation:** For the engine to succeed, it must have templates that can capture frequently asked queries. *Where can it obtain such templates? How can it predict frequent queries?* To address these challenges, we propose techniques to automatically generate templates and to discover templates from query forms and logs.
- **Template-driven query planning:** Each template corresponds to a possibly infinite number of queries, whose answers may lie in a large number of diverse sources. *How can the engine quickly locate relevant sources? To what extent can it plan ahead to answer queries in the template?* To address these challenges, we introduce a novel problem of template-driven query planning, propose a best-effort strategy for query planning, and present novel algorithms for plan construction and optimization.
- **Query processing & system evolution:** *How can the engine quickly find templates matching a query? What if no matching templates can be found? How can the engine improve to address its deficiency?* To address these challenges, we propose a novel multi-level approach to query parsing and present techniques to evolve the engine by learning from its interaction with users and sources.

2. TEMPLATE CREATION

Manually building templates to support a large number of queries is likely to be a labor-extensive process. So the first challenge is: *can we automate this process as much as possible?* To address this challenge, we propose a solution to systematically generate templates using domain models.

A domain model essentially captures the knowledge that the engine has about a particular domain. A domain model may contain the following components: (1) *concepts*, e.g., movie, director, and actor for the movie domain; (2) *attributes* of concepts, e.g., the movie concept might have attributes such as title, genre, and release year, while director

[1]It is informative to note that search engines such as Google return very different results for the two queries.

and actor might have attributes such as name and address; (3) *relationships* of concepts, e.g., "direct" relationship between movie and director; (4) *instances* of concepts and relationships, e.g., director "Victor Fleming" directed movie "gone with the wind"; (5) *synonyms*, e.g., film and picture are synonymous with movie; and (6) *statistics*, e.g., movie is more frequently used than film and title is the most popular attribute of movie. Note that a concept typically corresponds to a set of real-world entities, e.g., the movie concept represents all movies. Note also that relationships are often denoted by verbs or verb phrases, e.g., direct and star in. An important kind of relationship between concepts is the taxonomic or is-a relationship, e.g., release year is a year, director or actor is a person, and address is a location.

To generate templates using a domain model, we first determine main target types (i.e., entity, attribute, and relationship) of queries in the templates, and then employ type-specific rules to generate the templates. Specifically, *entity queries* look for entities with certain characteristics. A possible rule is: start with "list", followed by the plural form of the concept name for the entities, and then a whose-clause specifying the characteristics. For example, a template "list movies whose genre is genre and release year is release_year" asks for movies with specific genre and release year. A template may have many variants. For example, in the above template, the characteristics of entities may be stated using pre-modifiers as: "list release_year genre movies". For another example, since release year is a (kind of) year, an equivalent template is "list genre movies in release_year".

In contrast, *attribute queries* ask for attribute values of known entities. These queries may start with "what is (are)". For example, "what is the genre of movie_title" asks for the genre of a specific movie with known title.

Finally, *relationship queries* ask for entities that are related to some other entities. For example, "which movies were directed by director" asks for movies directed by specific director. In this template, "movie" is the concept name and "directed by" comes from the relationship "direct" between movie and director.

However, the above solution does not address two key issues: *Where does the domain model come from? How can the templates capture most frequently asked questions?*

Learning from query forms: To address these issues, we propose a novel approach to learning domain models from query forms of Web databases. Although there have been many works on learning domain models [4], past efforts are mostly focused on learning from texts and structured data. Little attention has been given to learning from query forms. However, query forms are invaluable resources for learning domain models, especially in our context, for several reasons. First, query forms of databases capture salient aspects of the databases that designers of forms expect users to be interested in. Second, there are plenty of databases with query forms available on the Web and through examining a large number of forms, we can gather important statistics about the domain such as popular concepts and instances. For example, we may learn that make and brand (of cars) are synonyms, and make is the most popular query attribute for cars. Using this information, we can then direct the template generation process described above to generate templates for frequent attributes on the query forms.

One approach to learning a domain model from a set of query forms in a domain is to first extract query attributes from each form and then group attributes using effective clustering algorithms (e.g., [17]) that were developed *specifically* to address the unique challenges (e.g., the lack of attribute values for text fields) in analyzing query forms. Each cluster contains a set of semantically similar attributes, e.g., make and brand, and the importance of an attribute may be measured by the number of attributes in the cluster where the attribute belongs. Furthermore, by observing how attributes are located over a large number of query forms, we may discover attributes that describe the same concept. For example, make and model of cars are often located close to each other on query forms, while it is unusual that attributes of dealership would be placed between them.

Learning from query logs: Besides query forms, another important resource for learning domain models and frequent asked queries is query logs of keyword-based search engines. Compared to query forms, keyword queries are much closer in syntax to natural language queries. For example, fields for director names on query forms of movies databases are more likely to be labeled as "director" than "directed by". On the other hand, "movies directed by clint eastwood" might be more natural than "movies director clint eastwood".

Due to its importance in learning user intent, the problem of analyzing query logs has received a lot of attention [1]. Recent study [1] indicates that web search queries often follow certain patterns, e.g., "flights from origin to destination" for flight search and "make model year" for car search. However, we note that while these patterns are commonly used in web queries now, they tend to be very ambiguous. For example, it is not clear from the query "flights from chicago to new york" whether users are searching for round-trip or one-way flights or flights at preferred times. In fact, some more precise queries found at Google are "round trip flights from chicago to new york", "flights from chicago to new york today", and "all flights from chicago to new york".

There have been several works on mining query patterns from query logs (e.g., [1]). Clearly such patterns could provide much insight into the common interest among web search users, and thus may suggest good templates for the proactive engine. However, these works assume that query attributes (e.g., origin and destination of flights) and their instances are already known. This assumption is too strong in our context, since our goal is to learn not only query templates, but also domain model. To address this limitation, we propose an approach that can simultaneously discover domain model and query templates. The *key* idea is to leverage variations among similar queries to learn the structure of queries. For example, consider a query "flights from chicago to new york". Comparing it to a similar query "flights from chicago to los angeles", we may infer that "new york" is likely similar to "los angeles", and that "to" likely represents a new attribute. Furthermore, after seeing "flights from chicago", we may further infer that "to new york" expresses an *optional* query condition over the attribute "to".

Finally, we note that yet another interesting resource for learning query patterns is query logs of form-based query interfaces. For example, such a log may reveal that title is the most commonly *used* field on an interface.

3. TEMPLATE-DRIVEN PLANNING

After templates have been created, the engine must determine how to obtain answers for the queries in the templates. This gives rise to the novel problem of *template-*

driven query planning. As described earlier, a key advantage of a template-based engine is that it can plan ahead offline on query answering. However, there are several serious challenges to be addressed. First, query answers may be found in a *large* number of *diverse* sources whose querying capabilities, schemas, content coverage, and computational resources may vary greatly. Second, it is often impossible to generate complete plans for some templates or some queries in the templates. For example, consider a template "find books written by author". Since there may be a large number of authors, the engine might not know ahead of time all authors and which source has books by which author.

To address these challenges, we adopt a *best-effort* strategy for query planning. In other words, the engine will try its best to build query plans for templates. But due to limitations in its knowledge and resources, it might not be able to always generate complete plans. For example, in the above template, the engine might generate a complete plan ahead of time for best-seller authors, e.g., Bill Clinton. Suppose that only two sources A and B have his books, and A has more books than B. Then a possible plan would be: access A first; if no results, then retrieve from B. For other authors, the engine might generate more abstract plans. Such a plan might indicate that answers can be found in a set of data sources but leave out details such as the order of accessing these sources and how to combine their results.

Similar best-effort strategy has also been employed in information extraction [15] and data integration [10]. It helps avoid the huge upfront efforts in building a complex system and quickly bootstrap the system.

The problem of planning queries for the execution over a set of heterogeneous data sources has been extensively studied in data integration [6]. However, query planning in the traditional data integration is largely *schema-driven*. First, a mediated schema needs to be created to capture *all* query needs. Next, schema mappings need to be discovered to capture the relationships among all elements in the mediated schema and source schemas. Finally, query plans are generated by rewriting queries using schema mappings. A *drawback* to this approach is the huge upfront efforts needed for creating the mediated schema and schema mappings. In contrast, the proposed template-driven approach focuses on planning only the queries in a template. It does not require a comprehensive mediated schema: in a sense, each template is a small mediated schema. Furthermore, the domain model only needs to be rich enough to cover the templates in the domain. Thus this *template-at-a-time* pay-as-you-go strategy greatly reduces the complexity of query planning and helps quickly bootstrap the engine.

Based on the above motivations, we now describe a *two-phase* compositional approach to template-driven query planning: phase one determines how to obtain answers for a template from individual sources and outputs *source-specific access paths*; then phase two assembles these paths into query plans and optimizes the plans. A key advantage of the approach is that it greatly facilitates the incremental development of query plans. For example, to add a new source to the plans, we just need to find its access path and reinvoke the plan assembler in the phase two.

Finding source-specific access paths: In RDBMS, access paths are used to describe methods of retrieving tuples from tables, e.g., scanning the entire table or using an index to directly look up the desired tuples. Here access paths are ways of obtaining answers to the queries in a template from data sources. To build such paths, we first need to locate among a huge number of sources ones that might provide answers to the template. We observe that similar challenge arises when search engines try to answer keyword queries using structured data sources [10].

To address this challenge, there have been some works on developing schema and value indexes to help locate sources whose schema elements and data values match attributes and values extracted from queries [10]. However, our focus is very *different*. Instead of finding sources to answer specific queries, we are searching relevant sources for a query template and might not know all possible values of its parameters and hence the exact queries. As a result, while schema indexes may be adapted to our context, the value-based indexes developed in these works become less useful.

To address this limitation, we propose several *new* types of indexes for searching schema elements (e.g., relational attributes and XML elements) of sources based on the characteristics of their values instead of actual values. These indexes include type index for searching elements by their data types (e.g., string or numeric), pattern index for searching by value patterns (e.g., the number of digits or the presence of special characters such as '-'), and distribution index for searching by the characteristics of value distributions (e.g., the average and standard deviation of numeric values).

Once relevant sources for a template have been located, we may then proceed to find source-specific access paths. There are two *key issues* to be addressed. (1) Sources may vary greatly in their querying capabilities and the ways that their data may be accessed. As a result, access paths to different sources may be very different. (2) Some sources might not be able to provide a complete answer to the template. As a result, we need a mechanism to describe the content of sources with respect to the template. Such descriptions will be *critical* to query planning.

To systematically address the *first* issue, we divide sources into two categories: one that engine has direct access to their data and the other not. Data sources in the first category include files and databases that store data extracted from the Web, e.g., from tables and lists on web pages [3]. Access paths to these sources largely depend on how the data are stored. For example, consider a table extracted from Wikipedia listing U.S. presidents (e.g., Barack Obama) and their terms (e.g., 56-th). Suppose the table has two columns name and term, and is stored in a relational database. Consider a template "who is the term president of united states". Then the access path is: execute a query "select name from presidents where term = term" against the database.

Sources in the second category are the ones that only provide limited access to their data, e.g., via query forms or Web services. For these sources, the engine may not pose arbitrary queries and might need to extract answers embedded in query results. For example, consider a template "list NSF grants of PI". To answer this template using the NSF award database [12], the engine needs to construct an access path which consists of: (1) fill out search form on NSF web site, with its PI field set to the value obtained from the template; (2) submit the form and obtain result in HTML pages; and (3) extract grant details from the pages.

To address the *second* issue, we need a *formalism* for describing the content of sources with respect to a particular template. Similar formalisms, such as global-as-view and

local-as-view, have been developed in data integration [6]. We adapt the *local-as-view* method to our context, due to its flexibility in handling new data sources. In the conventional local-as-view method, data sources are described using queries (views) expressed over a mediated schema. In contrast, we use views to relate sources to the template.

To illustrate this, consider a template T for querying showtimes of top-rated movies in given theaters. Following the Datalog notations [6], we may represent T using a predicate T(title, rating, location, time) whose arguments represent title and rating of movie, location of theater, and showtime respectively. Consider a source S_1 listing movies and their ratings and another source S_2 listing movies and their showtimes. Note that neither S_1 nor S_2 provides a complete answer to T. In local-as-view, S_1 may be expressed as: S_1(title, rating) :- T(title, rating, location, time), and S_2 as: S_2(title, location, time) :- T(title, rating, location, time). Note that views are not restricted to simple projections as above. For example, if S_2 has only evening movies, its view would be: S_2(title, location, time) :- T(title, rating, location, time), time = 'evening'.

Plan construction and optimization: In phase two, query planner constructs plans based on source descriptions and access paths built in phase one. It faces two *challenges*: How to combine partial answers from individual sources into a more complete answer? How to select the best query plans among a possibly large number of alternative plans?

To address the *first* challenge, we may use source descriptions to determine how to combine sources to answer the template. We note that the general problem of answering queries using views is *NP-complete* [6]. However, our problem is much simpler: we focus on answering particular templates instead of arbitrary queries and thus may use template-specific views rather than the complex views defined over the mediated schema.

To illustrate the approach, consider the template T and sources S_1 and S_2 as described above. From the source descriptions, the planner may determine that neither source provides all attributes in T. However, two sources share a common attribute title and thus may be joined to provide a more complete answer to T. (Note that finding common attributes may require the resolution of semantic heterogeneity among different sources [6].) Based on this reasoning, the planner may then produce a plan for T as: S_1(title, rating) $\bowtie$ S_2(title, location, time).

To address the *second* challenge, we propose a model for measuring the quality of query plans. The model incorporates several key factors. (1) *Coverage*, e.g., measured by the number of results generated by the plan. For example, consider a template "books on subject". Suppose the engine has a list of known subjects, e.g., American history, and Amazon has more American history books than Barnes & Noble. Then a plan that obtains answers from Amazon may be ranked higher than that from Barnes & Noble. (2) *Authority of sources*, e.g., measured by their PageRanks. Thus plans that obtain answers from more authoritative sources are preferred. (3) *Communication cost*, e.g., measured by the number of queries that need to be posed to the sources for executing the plan [6]. For example, consider the above plan that joins S_1 and S_2 on title to answer the template T. One method of executing the plan is to first obtain titles and ratings from S_1, and then for each title, pose a query to S_2 to obtain its theaters and showtimes. Another method is to obtain all tuples from S_1 and S_2 and then perform the join in the engine. The first method would need to pose one query to S_1 plus as many queries to S_2 as the number of movies, while the second method only needs two queries. Thus, the first method will likely have a higher cost.

Note that coverage is often at odds with communication cost: to achieve greater coverage, we may need to retrieve from more sources, which in turn leads to higher communication cost. To address this, we may consider *time-constrained* query planning: find plans with the highest coverage under the constraint on targeted query response time (e.g., .5 sec).

4. QUERY PROCESSING & SYSTEM EVOLUTION

With the offline planning, the engine needs to do much less work at query time. However, it still needs to address several key issues when processing queries: (1) How to quickly locate among a large number of templates ones that (partially) match queries? (2) How to continue to improve itself?

The problem of locating templates that match queries is similar to that of finding pre-compiled questions (e.g., FAQ's) that are similar to user questions in question answering [2]. Previous research has employed information retrieval techniques to locate similar questions by treating questions as (short) documents [2]. However, instead of actual questions, we are searching for templates with parameters. As a result, these techniques are not directly applicable.

Parsing queries: To address the first issue, we propose a novel *multi-level* approach to parsing queries. The key idea is to first use cheap methods to eliminate irrelevant templates and then perform in-depth parsing on the remaining ones.

The first level, *domain identification*, determines the domain of a query. At this level, a domain is simply represented as a document containing all tokens in the domain model and templates. The similarity between a query and a domain may be measured using the TF*IDF function from information retrieval. The domain with the highest similarity score will be chosen to be the domain for the query.

The second level, *shallow semantic tagging*, determines components in the query and their *possible* semantic types. For example, in a query Q: "list clint eastwood directed films 2012", list may be recognized as a question word, clint eastwood as the name of director or actor, directed as the relationship direct, films as the concept movie, and 2012 as the attribute release year. After the query has been tagged, templates with highly similar tags (judged by the TF*IDF of tags) are identified and retained for further considerations.

The last level, *template-based parsing*, removes ambiguous tags in the query by comparing it with top-ranked templates from the second level. For example, comparing Q with a template T_1 "list movies in year directed by director", we may infer that clint eastwood in Q refers to his role as a director. Besides the disambiguation, the engine also needs to handle partial matches. For example, compared to another template T_2 "list movies directed by director", Q is *overspecified* in that it asks for movies in specific year that T_2 cannot answer. On the other hand, a query Q' "movies clint eastwood" is *underspecified* with respect to T_1. In this case, we are not sure whether Q' asks for movies directed by Clint Eastwood or movies starring him or both. In both cases, the engine should inform users of how it interprets the query, when it returns the answer.

Note that when there are multiple templates that par-

tially match a query, the engine needs to properly *merge* the results given by different templates. Such merging can be either shallow, e.g., a simple concatenation of results, or deep, e.g., a ranked list of results with duplicates removed.

Evolving the system: There are several ways that the engine can learn from its interaction with users and sources. First, it may learn new templates from queries that existing templates fail to capture. For example, from an overspecified query "toyota camry 2012" and the corresponding template "make model", it may learn a new attribute year. Furthermore, if similar queries are frequently asked, it may generate a new template "make model year".

Second, it may learn new variations for existing templates from user clickthrough data. To illustrate this, consider a similar query "toyota 2012 camry". If users frequently click on the answer produced by the template T "make model year", it may infer that the order of model and year in T may be interchanged without affecting its semantics. As another example, consider a query "movies clint eastwood director". If many users click on the answer given by the template "list movies directed by director", it may infer that "directed by" may be replaced by "director", and "list" may be optional.

Finally, it may also improve query plans by learning from their past execution history. For example, suppose that in planning for a template T, it only knows that sources A, B, and C can provide answer to T. After several queries have been answered using T, the engine may learn more about these sources, e.g., A always returns answer faster than B, and C has the best coverage. The engine may then rerun the plan optimization algorithm to generate a better plan.

5. RELATED WORK

A key novelty of the proposed template-based query parser is to use template-based grammars to perform targeted parsing of user queries. However, templates might not be able to capture all queries and their variations. In such cases, shallow syntactic and semantic parsing techniques [7] may be employed to tag the words in the queries by their syntactic and semantic roles (e.g., part-of-speech or verb-argument relationships). The similarity of queries with the templates may then be measured based on the similarity of their tags. These parsing techniques may also be utilized to discover query patterns. For example, we may discover noun-phrase query patterns (e.g., "books written by author") if there are a large number of such queries in a query log.

Natural language interface to databases remains an overarching goal despite decades of research. Such an interface is also highly desirable for end users to query XML data [9]. Recently, [16] conducted a user study on replacing complex query forms with free-text query interface. Our work largely departs from these efforts by using templates to capture user interests and parse user queries.

Wrappers are key components in a traditional data integration system [6]. Wrapper construction is a well-known challenging task due to the difficulties in automatic understanding of source query interfaces. The proposed template-driven query planning enables an *incremental* template-at-a-time wrapping of sources, thereby eliminating the huge upfront efforts in building full-fledged wrappers.

[14] proposed an approach to annotating search engine queries using attributes in a relational table. This annotation process is similar to the shallow semantic tagging in our proposed query parser. However, annotation is the final parsing step in [14], while we leverage templates to refine the annotations to generate more accurate query parses.

Recent work [11] proposed to create views in a *single* database to capture user interests and permit keyword search over the views to improve the search efficiency. In contrast, we use templates to *directly* capture user query needs and focus on planning queries in the templates for the execution over a *multitude* of heterogeneous data sources.

6. CONCLUDING REMARKS

We have presented a proactive search engine for tapping into structured data on the Web using natural language queries. The engine is *unique* in several aspects. (1) It addresses the key limitations of current keyword-based engines and the "impedance mismatch" between unstructured keyword queries and structured data. (2) It adopts a novel template-based parsing paradigm to address fundamental challenges in parsing natural language queries. (3) It employs a novel template-driven approach to offline query planning, which may greatly reduce query response time and help quickly bootstrap the engine.

Besides search engines, we expect that the proposed techniques may also be employed in many other user-facing applications, such as data warehousing, content management, and mobile data management, to revolutionize the way that users interact with the systems.

Acknowledgment: We thank anonymous reviewers for their invaluable comments. This work is supported in part by the Faculty Research Grant of UNC Charlotte.

7. REFERENCES

[1] G. Agarwal et al. Towards rich query interpretation: walking back and forth for mining query templates. In *WWW*, 2010.
[2] R. Burke et al. Natural language processing in the faq finder system. In *AAAI spring symposium*, 1997.
[3] M. J. Cafarella et al. Webtables: exploring the power of tables on the web. *PVLDB*, 1(1), 2008.
[4] M. J. Carman et al. Learning semantic descriptions of web information sources. In *IJCAI*, 2007.
[5] Y. Chen et al. Keyword search on structured and semi-structured data. In *SIGMOD Conference*, 2009.
[6] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.
[7] D. Gildea et al. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3), 2002.
[8] M. A. Hearst. *Search User Interfaces*. Cambridge University Press, 2009.
[9] Y. Li et al. A domain-adaptive natural language interface for querying xml. In *SIGMOD*, 2007.
[10] J. Madhavan et al. Web-scale data integration: You can afford to pay as you go. In *CIDR*, 2007.
[11] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR*, 2009.
[12] National Science Foundation. NSF Award Search. http://www.nsf.gov/awardsearch/.
[13] E. Sadikov et al. Clustering query refinements by user intent. In *WWW*, 2010.
[14] N. Sarkas et al. Structured annotations of web queries. In *SIGMOD*, 2010.
[15] W. Shen et al. Toward best-effort information extraction. In *SIGMOD*, 2008.
[16] K. Tjin-Kam-Jet et al. Free-text search versus complex web forms. In *ECIR*, 2011.
[17] W. Wu et al. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD*, 2004.