Assisted Querying using Instant-Response Interfaces

Arnab Nandi
Dept. of EECS
University of Michigan, Ann Arbor
arnab@umich.edu

H. V. Jagadish
Dept. of EECS
University of Michigan, Ann Arbor
jag@umich.edu

ABSTRACT

We demonstrate a novel query interface that enables users to construct a rich search query without any prior knowledge of the underlying schema or data. The interface, which is in the form of a single text input box, interacts in real-time with the users as they type, guiding them through the query construction. We discuss the issues of schema and data complexity, result size estimation, and query validity; and provide novel approaches to solving these problems. We demonstrate our query interface on two popular applications; an enterprise-wide personnel search, and a biological information database.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Human Factors, Design, Languages

Keywords

keyword, query, interface, autocompletion

1. INTRODUCTION

The problem of searching for information in large databases has always been a daunting task. In current database systems, the user has to overcome a multitude of challenges. To illustrate these problems, we take the example of a user searching for the employee record of an American *"Emily Davies"* in an enterprise database. The first major challenge is that of schema complexity: large organizations may have employee records in varied schema, typical to each department. Second, the user may not be aware of the exact values of the selection predicates, and may provide only a partial or misspelled attribute value (as is the case with our example, where the correct spelling is *"Davis"*). Third, we would like the user to issue queries that are meaningful in terms of result size a query listing all employees in the organization would not be helpful to the user, and could be expensive for the system to compute. Lastly, we do not expect the user to be proficient in any complex database query language to access the database.

Work supported in part by NIH grant R01 LM008106 and NSF grant IIS 0438909.

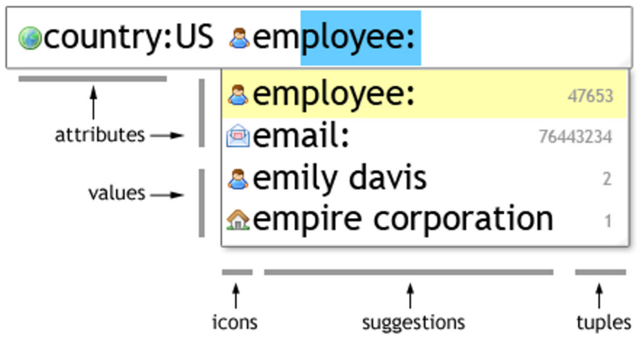

Figure 1: The instant-response interface

2. THE INSTANT-RESPONSE INTERFACE

The instant-response interface is similar in interaction to various services such as auto-completion in word processors and mobile phones, and keyword query suggestion in search engines. The user is presented with a text box, and is expected to type in keywords, a simple interaction paradigm that is considered the standard in search systems. As soon as the user starts typing text into the textbox, the system instantly begins to provide suggestions, as shown in Figure. 1. The first set of suggestions is a set of schema level parameters; i.e. a ranked list of keys, or entity names; which are simplified names for attributes and table names in a relational database, or element tags in an XML database. This allows the user to discover the schema as an ongoing implicit process. The next set of suggestions is a ranked list of text fragments from the data, which provides the user with an insight into valid predicate values. Each text fragment has an associated icon corresponding to the entity type it belongs to, which are expanded to a full *key:value* pair on selection. This allows the user to disambiguate between text values based on their type. Clicking on the icons in the textbox will wrap the text selection around the associated key name and hence trigger the drop down to modify the key, if needed. In addition to these suggestion sets that are integrated into a single list, the estimated number of tuples, or result objects from the resultant constructed query is provided on the right of each suggestion. If the estimated number of results becomes zero due to a value string that never occurs, or a query that violates the schema, further suggestions are stopped and the background of the textbox changes color, depicting an invalid query. The user is then expected to reformulate the query by changing the terms entered.

2.1 User Experience

In the case of our example, the user first types in *"US"*, to specify that he is considering only the United States offices of the organization. Since *"US"* came up as a valid suggestion, the user selected it from the drop down box, which expands the query to *country:US.* This is beneficial for the database, since it can now do an exact attribute match instead of looking through multiple attributes. Then, the user types in *"em"*; the suggestion mechanism returns a list of suggestions, with two key values, and two data values. The user notices that there are two employee records which have *"Emily Davis"* in them, and selects the respective suggestion. Since the query so far will still return a result set of cardinality two, the user decides to refine the query even further, typing in *"department:"*, which provides the suggestions *"Sales"*, and *"HR"*, one for each result tuple. By selecting *"HR"*, the user is able to locate the single employee record of interest, and directly navigate to it. It should be noted that the complete interaction occurs over just a few seconds, as all suggestions are provided in real time as the user is typing out the query.

3. PROBLEM DEFINITION

We adopt the standard conjunctive attribute-value query language, where queries are of the form

$$Q = key_1 : value_1 \cdot key_2 : value_2 \cdot key_3 : value_3 \cdot \ldots$$

Where key_i denotes the *key* or *attribute* of a particular object, and $value_i$ denotes a specific value for that attribute or key. This form of querying can be considered an annotated form of basic keyword querying that is simple enough to be effective and yet rich enough to efficiently query the database. It is used widely in mainstream search engines, and the biomedical community. Keyword identifiers are considered optional, so that a pure keyword query with no attribute name specifications is also acceptable. Any such value is expected to match any attribute of the selected object. Conversely, a single key identifier without a data value is also acceptable, and would be interpreted as the parent type of the expected results. Depending on the schema, the key attributes specified may not belong to the object of interest, but rather to a "closely related" object. In such a case the query is interpreted using the techniques suggested in Schema-free XQuery[5].

3.1 Challenges

In contrast to visual query systems[8, 2], the idea of incrementally and instantaneously formulating queries using database schema and data brings forth its own set of problems. Unlike form-based querying where the users information need is explicit, we need to *infer* what data the user is expecting to find *while* the user is typing in his query. Also, while recent developments in completion based search systems[4] attempt to apply efficient index structures for basic keyword querying; this is only a small part of the challenges faced in our schema and data based language query system.

Suggestion Quality

There is a subtle element of distraction introduced while proposing suggestions to the user, which may overwhelm the user[1]. Hence, we have to keep in mind that the quality of the suggestions need to be high enough such that it is clear that the suggestion mechanism is worth the negative effects of UI-based distraction. In addition to this, we acknowledge the well known human mind's limit[6] to perceive and consider choices, taken to be approximately seven. Given this fixed limit, we use a ranking strategy based on acceptance likelihood, to prioritize suggestions and display only a *top-k* set of suggestions.

Performance

The notion of suggesting information to the user at the time of input also depends critically on the "instantaneous" appearance of the suggestions. It has been shown[7] that a maximum delay of 100ms can be considered for user interfaces to seem instantaneous; clearly all our suggestions need to generated and displayed to the user in less time than this. This poses challenges for our system, which is required to be efficient enough to afford extremely fast querying even for large databases with diverse schema.

These two challenges must each be addressed in the context of attribute name suggestions, data value suggestions, and size estimation. We describe our solutions to these challenges in the following section.

4. IMPLEMENTATION

4.1 Schema based suggestions

The first few entries in the suggestion list are a ranked list of suggestions for possible keys. This is done by using a suffix tree to generate all possible attributes that are a completion of the current word typed in. All candidate keys are then ranked according to distances in the schema graph, between the candidate keys and the currently selected keys. Since the user may not know actual attribute names, or the attribute names may be too cryptic to be used as keywords, we maintain a metadata map that stores all the convenient words used for the field name. We augment this metadata with a subset of the WordNet dataset for further unsupervised synonym-based expansion.

4.2 Schema based validation

We use the schema to construct a reachability index for detecting validity of the query. An $n \times n$ matrix is constructed, where *entry(i,j)* is true if element name i occurs at least as an n^{th} level descendent of element name j in any data object in the database, where n is a upper bound parameter set during database set up. For example, an attribute *"employee"* could contain attributes *"name"* and *"country"* in its data, while *"designation"*(an employee attribute) and *"building-type"*(an office building's attribute) will not have any common 1^{st} or 2^{nd} level ancestors in a personnel database. The query is considered valid with respect to the schema if there exists at least one row in the matrix that contains true entries for each of the attributes in the query. To define descendents in relational data, we develop a hierarchy with attributes at the leaves, tuples as their parents, table as the grandparent, and joins amongst tables as higher levels of ancestry.

4.3 Data fragment suggestions

All human-readable data in the database is first fragmented into simple chunks of phrases using statistical methods. We then construct two tries; in the first, each phrase is entered with its attribute as a prefix; in the second the attribute is entered as a suffix. The first trie is used for attribute-specific completion, where the attribute name has been provided by the user. The second trie is used for generic completions, where the attributes are stored as suffixes of the phrase string, allowing us to deduce the attribute name, and hence the icon for the suggestion. In the case of attributes with a large number of distinct phrases, we prefer to reuse the generic completion trie in place of the attribute-specific trie, for efficiency reasons. The leaf node of each trie is also appended with the count information, which is used to rank the suggestions in descending order of popularity.

4.4 Size estimation

Each item in the database is provided a unique ID based on its ancestry, in the form $x.y.z$, where x is the parent of y, which in turn is the parent of z. Ancestry is easy to understand for XML data. For relational data, we consider the same hierarchy concepts as Sec. 4.2. We use a fielded inverted index to store these unique identifiers. At query time, we merge the query-generated attribute lists using a simple ancestry check; which allows us to come up with the number of actual items that match the query. Since the query process is incremental, both attribute lists and merged lists are cached. In addition, we do not consider large lists for the sake of efficiency and time-constraints in our query processing. This is done by ignoring merges of any lists that are beyond a certain threshold count, and reporting an estimate of size based on an incomplete independence assumption.

5. SYSTEM ARCHITECTURE

In contrast to currently available iterative query building systems, our system employs a continuously updating interface to provide real-time query responses, allowing the user to incorporate suggestions and modify the query in real time, as opposed to going through a time-consuming cycle of iterative query building. Beyond the visible user interface, our system involves a backend server that handles the continuous query requests from the interface, and replies to it. At the core of the system lie the three modules that provide suggestion inputs, as described in the previous section. We use Lucene[3] as our inverted index for size estimation purposes. For the fragment suggestions, we implement in-memory trie-based data structures. The third module provides all schema-related functions, such as query validation and attribute-name suggestion. The information from these modules is then aggregated and combined to be presented as suggestions, which are served off the UI backend server. We implement a caching layer as an integral part of our system, caching not only requests for suggestions, but also the underlying size estimates, etc. Beyond this, the results are then passed on to the suggestion server, which serves the update calls of the user interface.

6. DEMONSTRATION

In our demonstration, we present to the user an interface visible as a single text box in a web browser. This javascript based interface communicates with the main Java-based backend suggestion server, which can be run either locally or on another system. We offer users a chance to pose queries to the database loaded in the server. The users are guided through the search process by the instantaneous suggestions, and are then presented with results upon query

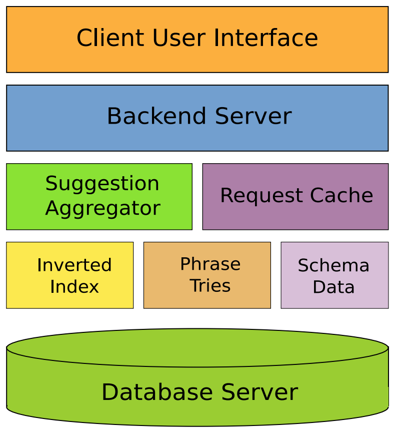

Figure 2: System architecture

execution. We also demonstrate the “plug and play” nature of our system, where we allow any existing relational or XML database to be loaded quickly for use with the instant-response interface, by simply pointing to the database location. The schema and other metadata information are automatically read from the database, and are used to load the database contents, creating the inverted index, phrase tries and schema graph. An optional step is provided to the user for annotating the schema with application-specific metadata (e.g. attribute “EMPZ” could be annotated with “Employees”).

We demonstrate our application on two datasets. The first dataset is based on the IMDB dataset. This comprises of movie information with over 100,000 records, including actors, movie titles and other film related information. For our second dataset, we use the MiMI molecular interaction database, which is an XML dataset over 1 gigabyte in size, containing over 48 million entities including data collected from external databases.

7. REFERENCES

[1] B. Bailey, J. Konstan, and J. Carlis. The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface. *Proc. INTERACT*, 2001.

[2] T. Catarci, M. Costabile, S. Levialdi, and C. Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Languages and Computing*, 1997.

[3] E. Hatcher and O. Gospodnetic. *Lucene in Action*. Manning Publications, 2004.

[4] I. W. Holger Bast. Type Less, Find More: Fast Autocompletion Search with a Succinct Index. *Proc. SIGIR*, 2006.

[5] Y. Li, C. Yu, and H. Jagadish. Schema-Free XQuery. *Proc. Conf. Very Large Databases*, 2004.

[6] G. Miller et al. *The Magical Number Seven, Plus Or Minus Two: Some Limits on Our Capacity for Processing Information.* Bobbs-Merrill, 1956.

[7] R. Miller. Response time in man-computer conversational transactions. *Proceedings of the AFIPS Fall Joint Computer Conference*, 1968.

[8] A. Sengupta and A. Dillon. Query by templates: A generalized approach for visual query formulation for text dominated databases. *Proc. Symposium on Advanced Digital Libraries*, 1997.