Combining Keyword Search and Forms for Ad Hoc Querying of Databases

Eric Chu   Akanksha Baid   Xiaoyong Chai   AnHai Doan   Jeffrey Naughton
Computer Sciences Department
University of Wisconsin-Madison
{ericc, baid, xchai, anhai, naughton} @cs.wisc.edu

ABSTRACT

A common criticism of database systems is that they are hard to query for users uncomfortable with a formal query language. To address this problem, form-based interfaces and keyword search have been proposed; while both have benefits, both also have limitations. In this paper, we investigate combining the two with the hopes of creating an approach that provides the best of both. Specifically, we propose to take as input a target database and then generate and index a set of query forms offline. At query time, a user with a question to be answered issues standard keyword search queries; but instead of returning tuples, the system returns forms relevant to the question. The user may then build a structured query with one of these forms and submit it back to the system for evaluation. In this paper, we address challenges that arise in form generation, keyword search over forms, and ranking and displaying these forms. We explore techniques to tackle these challenges, and present experimental results suggesting that the approach of combining keyword search and form-based interfaces is promising.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications

General Terms

Design, Experimentation

Keywords

Keyword search, query forms, relational databases

1. INTRODUCTION

As the success of Internet search engines makes abundantly clear, when faced with discovering documents of interest, the general public is successful at using keyword search to accomplish the task. However, it is much more difficult to pose structured queries to satisfy information requests over structured databases, since users need to know the query language (e.g., SQL) and the schema. Our goal in this paper is to explore techniques that assist users who do not want to use SQL in posing ad hoc structured queries over relational databases.

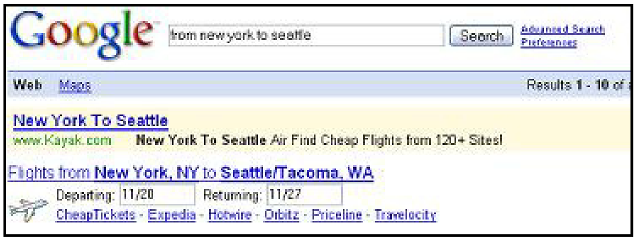
Our basic idea is to exploit the observation that for many tasks, it is easier to recognize a solution when presented with one than it is to construct the solution from scratch. A user with a question to be answered may find it easier to recognize a form that can be used to express a relevant query than it is for the user to generate that query from scratch. This observation suggests the approach of, given a structured database, generating enough forms to cover a wide variety of potential user queries, and then allowing the user to browse this set of forms when he or she wishes to pose a query. In non-trivial applications, there will be many forms to consider, and browsing this set of forms will itself be a non-trivial endeavour. Therefore, we propose the use of keyword search to help the user find a manageably small set of relevant forms – the user submits a keyword query; in response, the system returns a ranked list of relevant forms, from which the user selects and uses one to build a structured query.

The approach of keyword search leading to a form has already been used in an ad hoc manner by search engines such as Google and Yahoo!. For example, as shown in Figure 1, querying Google with "from new york to seattle" brings up a query interface to buy plane tickets as the first result. However, on the Web, the primary task of keyword search is to lead users exploring the Web to documents relevant to their search. While the search engine may occasionally return a form relevant to a query, there is no desire to support a wide range of possible structured queries. By contrast, we seek to develop a comprehensive approach that allows users to answer a wide variety of questions over a single structured data set.

Although the approach is straightforward in concept, when one actually attempts to implement such a facility, one is faced with myriad options and difficult decisions every step of the way. For example, how can one automatically generate a set of forms to support a wide range of queries? How specific or general should these forms be? How effective is keyword search in exploring this set of forms? What challenges arise in ranking the results of these keyword searches? And finally, can users really use the result of a keyword search to identify forms useful in satisfying their information requests?

Our main contributions in this paper are to 1) identify and elucidate these challenges, and explain how they arise, 2) give initial solutions to these challenges, 3) implement these solutions and conduct a user study to evaluate our approach of keyword search leading to forms leading to structured queries. Even with our initial solutions to the challenges inherent in this approach, the results are already encouraging – given a real-life data set and a number of information requests, without any manual intervention on our part, users were able to perform keyword search over our automatically generated set of forms,

SIGMOD'09, June 29–July 2, 2009, Providence, RI, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06...$5.00.



Google | from new york to seattle | Search

New York To Seattle
www.Kayak.com New York To Seattle Air Find Cheap Flights from 120+ Sites!

Flights from New York, NY to Seattle/Tacoma, WA
Departing: 11/20 Returning: 11/27
CheapTickets - Expedia - Hotwire - Orbitz - Priceline - Travelocity

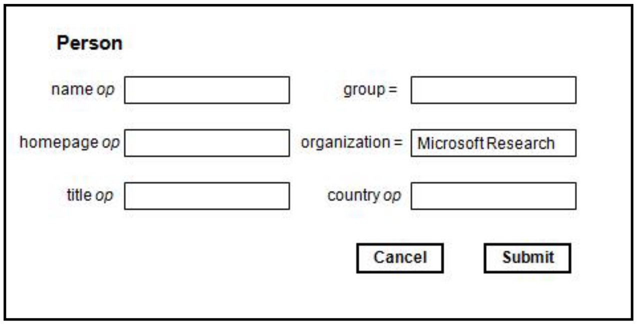
Figure 1. For the query "from new york to seattle," Google returns a simple form for finding flight tickets besides links to websites.

Relational Schema of DBLife	
Entity tables:	**# rows**
person(id, name, homepage, title, group, organization, country)	68459
publication(id, name, booktitle, year, pages, cites, clink, link)	108972
topic(id, name)	736
organization(id, name)	163
conference(id, name)	170
Relationship tables:	
// records two related persons and strength of this pair	
related_people(rid, pid1, pid2, strength)	115436
// records related person-topic pair and strength	
related_topic(rid, pid, tid, strength)	114196
// records related person-organization pair and strength	
related_organization(rid, pid, oid, strength)	2436
// records a person giving a tutorial in a conference	
give_tutorial(rid, pid, cid)	132
// records a person giving a talk in a conference	
give_conf_talk(rid, pid, cid)	131
//records a person giving a talk at an organization	
give_org_talk(rid, pid, oid)	913
// records a person serving in a conference and the // assignment	
serve_conf(rid, pid, cid, assignment)	3591
// records a person as an author of a publication and the // position of the person's name on the list of authors	
write_pub(rid, pid, pub_id, position)	328410
// records a pair of co-authors and strength	
co_author(rid, pid1, pid2, strength)	56370

Figure 2. The relational schema of the DBLife data set. Primary keys are underlined. The total number of tuples is 801,189.

and were successful in finding and filling out the right forms to satisfy the information requests, all done in under two minutes. This success with our end-to-end complete prototype suggests that the approach is promising and warrants further exploration.

Throughout this paper, to provide context and specificity to our presentation, we refer to the DBLife [5] data set in our discussion. DBLife manages information for the database research community. It monitors about 1,000 data sources and downloads about 9,500 pages daily to track mentions and entities. We use a snapshot of the data set as of June 2007. Its relational schema, shown in Figure 2, comprises five entity tables and nine relationship tables, which reference the entity tables. Figure 2 also shows the number of tuples in each table. The whole data set is about 40 MB. The rest of this paper is organized as follows. Section 2 addresses form generation. Section 3 describes our approaches to map keyword queries to forms and eliminate forms that do not produce answers with respect to a given keyword query. Section 4 considers ranking and grouping forms. Section 5 presents our experiments and the user study. Section 6 discusses related work. Section 7 concludes the paper.



Person

name *op* [ ] group = [ ]
homepage *op* [ ] organization = Microsoft Research
title *op* [ ] country *op* [ ]

Figure 3. An example of a query form for the DBLife data set with one of the predicates specified.

2. QUERY FORMS

A popular approach to query databases is to use forms, as they allow users who have no knowledge of the database query language or the schema to build structured queries. Each form is essentially an interface for a query template – an incomplete SQL query in which some components are parameters whose values are unknown until a user provides them when he or she fills out the form. Figure 3 shows an example of a completed form over the person relation of DBLife. When the form is empty, it maps to the template

```
SELECT   *
FROM     person
WHERE    name op value AND homepage op value
         AND title op value AND group op value AND
         organization op value AND country op value
```

where *op* and *value* are parameters representing an operator and a constant respectively. Figure 3, then, represents the query

```
SELECT   *
FROM     person
WHERE    organization = 'Microsoft Research'
```

In other words, a template with user-specified parameters corresponds to a SQL query. Predicates for which users have not specified parameters, such as those for name, homepage, title, and country in this example, are excluded from query evaluation.

To let general users build ad hoc database queries with forms, we want to generate forms that are easy to use and that, as a set, support a wide range of queries. In this section, we present a systematic approach to do so, and address challenges that arise in the process. Let *D* be a database instance and S_D be the schema of *D*. We can describe form generation as a four-step procedure:

1) Specify a subset of SQL as the target language to implement the queries supported by forms.
2) Determine a set of "skeleton" templates specifying the main clauses and join conditions based on the chosen subset of SQL and S_D.

SQL':

```
Let B = (SELECT       select-list
         FROM         from-list
         WHERE        qualification
         [GROUP BY    grouping-list
         HAVING       group-qualification])
```

where

- *select-list* comprises a list of column names, and, if applicable, a list of terms having the form *aggop(column-name)*, with *aggop* being one of {MIN, MAX, COUNT, SUM, and AVG}.
- *from-list* is a list of tables.
- *qualification* is a conjunction of the conditions of the form *expression op expression*. An expression is a column name or a constant, and *op* is one of the comparison operators {<, <=, =, <>, >=, >, LIKE}.
  Note: we do not allow nested queries in FROM and WHERE clauses.
- *grouping-list* and *group-qualification* are as defined in SQL-92 (i.e., no *every* or *any* in *group-qualification*).

We consider queries of the form B [UNION|INTERSECT B].

Figure 4. SQL', the subset of SQL we consider for form generation in this paper.

3) Finalize templates by modifying skeleton templates based on the desired form specificity.
4) Map each template to a form.

We elaborate on each of these steps in the following.

2.1 SQL'

We first specify a subset of SQL to be the target language implementing the queries supported by the query forms considered in this paper. This subset, which we term SQL', is shown in Figure 4. In principle, many different subsets of SQL can be considered. Since the goal of this paper is to explore the challenges and the promise of the approach of using keyword search to identify relevant forms for posing structured queries, the SQL' we use is intended to be simple enough to allow us to explore the challenges in depth and build an end-to-end prototype solution to evaluate the approach, while being expressive enough to cover a useful fraction of potential user queries that cannot be expressed with keyword search alone.

2.2 Schema-based Query Templates

The next question to address is which of the infinitely many possible query templates we should generate as candidates to return to users in response to their keyword queries. Clearly, the set of forms to generate depends on SQL' and S_D, the schema for the data set being queried. However, it is unclear what a reasonable scope for a template should be.

A related design issue is which part of a query template should be made parameters on the form. At one extreme, we could parameterize almost everything – relations, attributes, operators, and values – tantamount to a general graphical query-building interface, such as QBE [17]. However, many users use query forms because they are unfamiliar with the data model and the query language; expecting them to manage the full generality of such an interface seems unrealistic. Therefore, for each template, we parameterize only the attributes and the operators, and fix the SQL' clauses, the relations, and the join conditions. To do so, we first generate a set of *skeleton templates*, a preliminary sketch that we later modify to obtain the final templates.

Given S_D, we first create a skeleton template for each relation. Let R_i be a relation following a relation schema $S_i \in S_D$. If R_i does not reference other relations with foreign-keys, its skeleton template would be

Ex_{basic}: SELECT *
FROM R_i
WHERE *predicate-list*

where *predicate-list* is a conjunction of the predicates "attr *op value*," in which attr is a non-key attribute of R_i. We use Ex_{basic} for the entity tables in S_D as they do not reference other tables.

If R_i references other relations with foreign-keys, its skeleton template would support foreign-key joins as follows. In the FROM clause, we include R_i and the relations it references; in the WHERE clause, we join all these relations, and have an "attr *op value*" predicate for each attribute from these relations in the predicate-list. For example, consider the relation give_tutorial in Figure 2. Since it references the person relation and the conference relation, its skeleton template looks like:

Ex_{FK}: SELECT *
FROM give_tutorial t, person p, conference c
WHERE t.pid = p.id AND t.cid = c.id AND p.name *op expr* AND … AND c.name *op expr*

In addition, we create templates that support non-foreign-key equijoins on attributes that relations have in common. In DBLife, all nine relationship tables have the attribute pid, so we can do equijoins on pid for any non-empty subset of the nine relationship tables. These equijoins are equivalent to searching for people who have participated in some of the nine relationships. For example, to support queries about people who have given a tutorial, given a talk in a conference, and given a talk in an organization, the following skeleton template does an equijoin on pid in give_tutorial, give_conf_talk, give_org_talk, and person, and has predicates only from person in the WHERE clause:

Ex_{EQ}: SELECT non-key attributes from p
FROM give_tutorial t, give_conf_talk c, give_org_talk o, person p
WHERE t.pid = c.pid AND c.pid = o.oid AND o.pid = p.id AND p.name *op expr* AND … AND p.name *op expr*

In the DBLife schema, there are 502 possible equijoins over two or more of the nine relationship tables. For our experiments involving equijoins, we generated 36 skeleton templates for all equijoins on two different relationship tables and the table person (see the set of forms *F2* in Section 5.5). We chose this set of equijoin templates to facilitate tracking and analyzing the results. We can certainly choose another set based on considerations for the complexity and the usability of templates, storage constraints, and workload information (if any).

In practice, queries are often run against read-only views instead of the base tables, to keep querying separate from updates, and to have a more logical view of real-world entities that are stored in multiple tables because of normalization. The same principle – fixing tables, join conditions, and predicates – can be used to decide on the views (which are essentially queries) and to create templates on these views.

2.3 Form Specificity

As we use skeleton templates as a foundation to generate query templates that meet the specification of SQL', we need to determine how specific or general we want the forms to be. Intuitively, we say that a form that can be customized only in minor ways is very specific, whereas a form that is highly customizable is more general. Given a fixed set of queries S_Q, the more specific the forms are, the more forms we need to cover S_Q; likewise, the more general the forms are, the fewer forms we need to cover S_Q.

It is not obvious which point in this continuum of form specificity is the best choice. When there are fewer, more general forms, it is easier to find a form that supports the query a user has loosely in his or her mind. However, the user may have difficulty in understanding and using this form, especially when he or she is not familiar with the data model and the query language. Conversely, when there are a larger number of more specific forms, it may be harder to find a form that matches the user's specific information need, but when one is found, the necessary customization to express the query is minor.

To facilitate our analysis, we consider form specificity as a function of two quantifiable measures: form complexity, which refers to the number of parameters on a form, and data specificity, which refers to the number of parameters with fixed values on a form. Suppose the snapshot in Figure 3 is the initial state of a form. We can adjust form specificity in four ways:

1) Increase its complexity by adding more parameters (e.g., allows the counting of the number of tuples that satisfy existing predicates)
2) Decrease its complexity by removing existing parameters (e.g., the predicate on country)
3) Increase data-specificity by binding more existing parameters to constants (e.g., group = 'Databases')
4) Decrease data-specificity by unbinding parameters with fixed values (e.g., organization can be anything).

Our definition of form specificity gives us a natural approach to generate one or more query templates from each skeleton template – we first determine the desired form complexity based on SQL' and the skeleton template itself; then, we consider binding certain parameters to their data values.

We could map each skeleton template, which has only a SELECT-FROM-WHERE construct, to one large template supporting aggregation, GROUP BY and HAVING, and UNION and INTERSECT, as specified in SQL'; however, such a multi-purpose query template could be too complex for our target users. Therefore, we reduce form complexity by dividing SQL' into subsets, each supported by a separate template. For our experiments, we decided to divide SQL' into four query classes, each with a different intent:

1) *SELECT*: the basic SELECT-FROM-WHERE construct
2) *AGGR*: *SELECT* with aggregation
3) *GROUP*: *AGGR* with GROUP BY and HAVING clauses
4) *UNION-INTERSECT*: a UNION or INTERSECT of two *SELECT*

To adjust data specificity of a form, we bind only the "*value*" fields of the "attr *op value*" predicates in the WHERE clause to data values. Given a template *t* with a set of attributes *Z*, we first determine a subset of attributes *A* from *Z* whose values we want to bind; then, for each combination of data values for the attributes in *A* that exists in the database, we create a form for *t* in which the attributes in *A* are bound to these data values. We call this form-generation approach "data-aware" as it uses data values in addition to S_D and SQL'.

An important issue is to which extent we should make forms data-specific. While data-specific forms may enhance usability, an indiscriminate application of the data-aware approach could lead to a huge number of templates. Given a table with *n* columns and *r* rows, if we generate a "SELECT *" query template for every distinct combination of the "attr = *value*" predicates in the WHERE clause, we will already have $(\sum_{i=1}^{n} nCi) * r$ templates. Making data-specific templates that involve equijoins is also problematic because of cross products. Consider Ex_{EQ}, which has equijoins for the relations give_tutorial, give_conf_talk, and give_org_talk. Suppose that *A* includes the names of the tutorial, the conference, and the organization, and that a person *p* has *n1* tutorials, *n2* conference talks, and *n3* organization talks. The number of templates generated for Ex_{EQ} alone is already *n1*n2*n3*.

Another problem is maintaining the set of forms as the data set changes over time. A data-specific form could become invalid when the data set, after some updates, no longer has tuples satisfying the fixed predicates of that form. When updates are frequent, many data-specific forms could become invalid quickly. Keeping track of which forms have become invalid can be costly. In view of these disadvantages, we think data-specific forms are unlikely to be useful in general, so in this paper we do not consider data-specific forms.

2.4 Mapping Query Templates to Forms

Once we have generated a set of query templates, we can map each of them to a form. To build a form for each query template, we use the following standard form components:

- Label: for displaying text such as description for the form, the name of an attribute, a database constant, etc.
- Drop-down list: for displaying a list of parameter values from which users can choose one. For example, we use a drop-down list to allow users to choose the target attribute for an aggregation.
- Input box: for specifying a parameter value on the form.
- Button: for functions such as submit, cancel, and reset.

Form layout (i.e., where to put the labels, drop-down lists, input boxes, and buttons) is an interesting problem but one that is orthogonal to the issues we study in this paper. Accordingly, we did not focus on layout in this paper, and simply chose a basic layout for our experiments.

Each form has a brief English description based on its skeleton template. Ideally, a form description would be a user-intelligible summary of which queries are supported by the form. Much as snippets of documents are used to label the documents returned by an Internet search, these descriptions would be used to label the forms returned by a keyword query.

Unfortunately, generating natural-language descriptions for forms is a difficult challenge. Even when it is done manually, it is often unclear what a good description is, especially when a form supports a wide variety of queries. Also, a manual approach quickly becomes inefficient when there are more forms, but automatically generating good form descriptions is yet another challenge. For the forms for the DBLife data set, we first manually determined descriptions for the database relations, then automatically used them to label forms. For example, the description for the *SELECT* template over the coauthor relation is "Which two people are coauthors." The approach is simple and was effective in our experiments, although finding more sophisticated techniques for generating form labels is certainly an important area for future research.

2.5 Automating Form Generation

We built a template generator to facilitate form generation. The template generator uses the aforementioned specification for SQL' and query classes. It takes as input a data set and its schema. Form designers can specify the desired form complexity (i.e., a fixed number or all of the parameters) and data specificity (i.e., which attributes have fixed values). The output is a set of templates based on these configurations. We wrote scripts to transform these templates into forms and to add a form description to each form.

3. KEYWORD SEARCH FOR FORMS

3.1 Overview

The basic idea here is simple – treat a set of forms as a set of documents, then let users use keyword search to find relevant forms (which in turn are used to pose structured queries). However, this task differs from the standard document search problem in several key ways. In this section, we consider a number of approaches motivated by these differences.

Perhaps the most important difference is that a form contains parameters, which are undefined until users fill out the form at query time. Furthermore, possible data values for these parameters often do not appear on the forms. A keyword search approach that ignores this difference can yield undesirable results. Consider a first approach, which we call Naïve, which simply retrieves a form if the form contains at least one (OR semantics) or all (AND semantics) of the terms from a keyword query. If a user specifies a data value and we use Naïve-AND, we will get no answers. If we use Naïve-OR, some forms would be returned if the user includes in the query at least one *schema term* (i.e., a term that matches a table or attribute name). However, the *data terms* (i.e., terms that match data values), if any, would be completely ignored, which is not very satisfying.

We can solve this problem by putting data values on query forms. However, recall from Section 2.3 that generating data-specific forms for all interesting combinations of data values is impractical because it leads to a combinatorial explosion of forms. An alternative that uses a drop-down list of all possible values for each parameter would require many fewer forms, but it suffers the same problem of high storage and maintenance costs, and is impractical when an attribute has many possible values.

Moving beyond these approaches, we transform a user's keyword query by checking to see whether the terms from the query appear in the database, and if so, modifying the query with relevant schema terms. That is, if the keyword query contains a data value *d*, and *d* appears in table *R*, we rewrite the original user query to contain *R*. Because our forms include either all the attributes of a table (if the table appears on a form) or none (if it does not), using the table name is equivalent to using the name of the attribute that contains *d*.

A moment's thought shows we need to take some care in doing so. For example, what if a user-provided keyword appears both as a schema term and as a data term? What if the keyword appears in multiple attributes, possibly of different tables? Should we add the schema terms to the original user query, or replace the user-provided data terms with the corresponding schema terms?

We consider two basic approaches to resolving this issue. The first is to add all schema terms corresponding to the data terms in a keyword query from a user, and to evaluate it using OR semantics. We call this approach Double-Index OR (DI-OR), for reasons that will become clear when we describe how the approach is implemented. The second approach is to use AND semantics, but we cannot simply add all schema terms and use AND semantics, because we may generate empty results if we add two or more schema terms such that there is no form that contains all of them. Therefore, with AND semantics, we augment the original query by generating all possible queries that result from replacing user-supplied data terms with schema terms, use AND semantics for each query, and return the union of the query results to the user. We call this approach Double-Index AND (DI-AND).

As a simple example, return to our DBLife example and suppose that some user would like to know for which conferences a researcher named "Widom" has served on the program committee. The user might issue the keyword query "Widom conference," where "Widom" is a data term and "conference" is a schema term. Using Naïve-AND, we would get no forms, since "Widom" does not appear on any forms. Using Naïve-OR, we would ignore "Widom" and get all forms that contain "conference." Using DI-OR, we would find that "Widom" appears in the person table, so the resulting rewritten keyword query would be "Widom person conference," evaluated with OR semantics. Using DI-AND, we would generate two queries: "person conference" and "Widom conference," evaluate each with AND semantics, and return the union of the results. In this case, "Widom conference" would lead to an empty result, but this would not be the case if "Widom" were both a database term and a schema term.

Finally, with our form-generation approach, one scenario is potentially problematic with DI-AND. When a search involves a table referenced by many other tables, DI-AND returns all the forms for all these tables, even though some may return no answer with respect to the user query. Returning forms that can never produce results with respect to the user query can be annoying to users. We consider an additional optimization to identify and filter these "dead" forms from the results.

3.2 Double-Index

We now describe the double-index approaches in more detail. Given a keyword query, we augment user queries with *form terms* (i.e., terms that appear on a form, such as schema terms, SQL keywords, and natural-language descriptions), and retrieve forms containing the form terms. To implement this strategy, we use two inverted indexes, one on the data set and the other on the set of forms. The first index, called DataIndex, takes in a term and returns a set of <*tuple-id, table*> pairs. Each pair describes a tuple in the data set that contains the term: *tuple-id* is the primary key of the tuple, and *table* is the name of the

Double-Index OR (DI-OR)

```
Input:    A keyword query $Q = [q_1 q_2.... q_n]$
Output:   A set of form-ids F'
Algorithm:
  FormTerms = {}, F' = {}
  // Replace any data terms with table names
  for each $q_i \in Q$
    if DataIndex($q_i$) returns <table, tuple-id> pairs
      Add each table to FormTerms
    Add $q_i$ to FormTerms            // $q_i$ could be a form term
  // Get form-ids based on FormTerms
  FormIndex(FormTerms) => F'    // OR semantics
  return F'          // Ordered by ranking scores (Section 4)
```

Figure 5. DI-OR augments the query with form terms and evaluates the query with OR semantics.

table containing the tuple. The second index, called FormIndex, takes in a term and returns a set of *form-ids*, or identifiers of the forms containing the term.

Figure 5 shows our first approach, called Double-Index OR (DI-OR). DI-OR comprises two basic steps. The first step is the query rewrite. We probe DataIndex with each query term q_i in a query Q. If q_i is a data term, DataIndex will return a set of <*tuple-id, table*> pairs, and we will add each *table* to the set *FormTerms*, which is initially empty. We also add q_i itself to *FormTerms*, because q_i could be a form term, regardless of whether it is a data term. In the second step, we simply probe FormIndex with the terms in *FormTerms*, and return any form that has at least one of these terms.

Although this approach satisfies the new semantics, using OR produces results that are often too inclusive. We want to use an approach similar to DI-OR but with AND semantics. However, as mentioned before, it would be wrong to simply do one AND-query with all the terms in *FormTerms*. The reason is that a data term may appear in multiple unrelated tables such that no form would contain all these tables. For a query "q_1 *AND* q_2," the correct interpretation after the query rewrite should be "$a \in Sq_1$ AND $b \in Sq_2$," where Sq_i is a "bucket" containing the form terms associated with q_i, and a and b are two form terms from Sq_1 and Sq_2 correspondingly. Therefore, if a bucket has multiple terms, we will have multiple queries.

Figure 6 shows this bucket-based DI using AND semantics. In the query rewrite step, for each q_i, we add each new *table*, and q_i itself, to Sq_i. In the second step, we generate and add to $S_{Q'}$ all distinct queries, each of which taking one term from each Sq_i; then, for each query in $S_{Q'}$, we probe FormIndex and retrieve forms that have *all* terms in the query. Finally, we return these forms ordered by their ranking scores. Notice that if q_i is both a data term and a form term, we will have queries to search for both types of forms, one having a table that has q_i as a database constant, and one in which q_i appears as a schema term. If q_i is a false term, Sq_i will be empty. Also, although we choose AND semantics here, the bucket-based approach does allow OR semantics – for each query in $S_{Q'}$, we simply retrieve a form containing *any* term from the query.

Double-Index AND (DI-AND)

```
Input:    A keyword query $Q = [q_1 q_2.... q_n]$
Output:   A set of form-ids F'
Algorithm:
  FormTerms = {}, F' = {}
  // Replace any data terms with table names
  for each $q_i \in Q$
    $Sq_i$ = {}                              // Bucket for $q_i$
    if DataIndex($q_i$) returns <table, tuple-id> pairs
      for each table
        if table $\notin$ FormTerms
          Add table to $Sq_i$ and FormTerms
    if $q_i \notin$ FormTerms
      Add $q_i$ to $Sq_i$ and FormTerms
  // Get form-ids based on $Sq_i$
  $S_{Q'}$ = EnumQueries($\forall$ $Sq_i$)  // Enumerate all unique queries,
                                 // each having one term from each $Sq_i$
  for each $Q' \in S_{Q'}$
    FormIndex($Q'$) => F'     // AND semantics on FormIndex
  return F'          // Ordered by ranking scores (Section 4)
```

Figure 6. DI-AND generates new queries by taking a term from each bucket of form terms, evaluates each new query with AND semantics, and then union the results.

A concern about DI-AND is that, in the second step, the number of queries we generate is $\prod_{i=1}^{n}(|Sq_i|)$. For a query with n terms and each query term leading to m form terms, we would generate m^n queries. However, since we replace each data term with the name of the table containing the term, many of these queries will be duplicates and can be ignored (in our experiments, the maximum number of queries generated by this approach is 12). To reduce the number of distinct queries, we record the form terms included so far in *FormTerms*, and only add a form term to Sq_i if it is not already in *FormTerms*.

Another issue related to query terms being a mix of data terms and form terms is that users may enter synonyms of schema elements or SQL keywords (e.g., "people" for "person," "how many" for "count," etc.) when they do keyword search. One solution is to add synonyms to a query based on a thesaurus during query evaluation, perhaps at the same stage as we replace data terms with form terms. An alternative is to add a set of synonyms to each form during form generation, and not do any thesaurus lookup at query time.

For our prototype, we chose the second approach. We selected and added a set of keywords, such as synonyms of the schema elements and SQL keywords and other related terms, to what we call a form profile for each form. The form profile is invisible to users since its function is to improve the chance of returning the form when a user enters terms that are related but that do not appear on the form. We used this approach mainly because we want to keep our algorithms and implementations simple, so that we could more clearly evaluate the approach of using keyword search to find relevant forms. Also, since we only consider schema-based related terms, the set of terms is quite small (i.e., about 10 additional terms per form), so redundancy and storage are not an issue. However, as future work, it is certainly interesting to consider recognizing synonyms of both data terms and form terms, and more generally, extending our implementation to handle more sophisticated query expansion. In that case, thesaurus look up at query time may be more suitable.

3.3 Double-Index-Join (DIJ)

Recall from Section 2.2 that if a table *T1* is referenced by another table *T2* in a foreign-key relationship, all of *T1*'s attributes will be in the form for *T2*. As a result, if a query has a data term in *T1* and we use DI to evaluate the query, we will get *T2* as the corresponding form term, and return the form for *T2* (in addition to the form for T1). For example, suppose that a

```
Double-Index-Join
Input:     A keyword query $Q = [q_1 q_2 .... q_n]$
Output:    A set of form-ids $F'$
Algorithm:
  FormTerms = {}, F' = {}, X = {}
  // Replace any data terms with table names
  for each $q_i \in Q$
    $Sq_i$ = {}
    if DataIndex($q_i$) returns <table, tuple-id> pairs
      for each table T
        let I be the set of tuple-ids from T
        if $T \notin$ FormTerms
          Add T to $Sq_i$ and FormTerms
          // New "join" step
          SchemaGraph(T) returns refTables
          for each refTable
            if DataIndex(refTable:tid) is NULL for every $tid \in I$
              FormIndex(T AND refTable) => X
    if $q_i \notin$ FormTerms
      Add $q_i$ to $Sq_i$ and FormTerms
  // Get form-ids based on form terms
  $S_{Q'}$ = EnumQueries($\forall Sq_i$)
  for each $Q' \in S_{Q'}$
    FormIndex(Q') => F'
  return F' – X       // Filter "dead" forms
```

Figure 7. DIJ eliminates "dead" forms with extra probes to DataIndex.

user wants to search for a person named "John Doe" from DBLife, and that "John Doe" appears in the person table, but is not involved in any relationship (i.e., the "John Doe" tuple inperson is not referenced by any tuple in any relationship table). For the query "John Doe," in addition to returning forms for the person table, we would return forms for all the relationship tables that reference person.

At first glance, returning all these forms seems reasonable because every form is related to person. However, since "John Doe" appears only in person, if the user enters "John Doe" in the person.name field on any of these join forms, they will return empty results. We call forms that yield no answer with respect to the user query "dead" because they are not useful to users. As shown in Figure 7, we modify DI-AND to filter a common type of "dead" forms with the guarantee that no live forms will be eliminated. Intuitively, before returning a form, we check to see if the form will return an answer if instantiated with the data terms in the user query.

We perform this check as follows. Given a keyword query Q, we probe DataIndex with each query term q_i. When q_i is a data term that leads to a set of <*table, tuple-id* > pairs, we look up each *table T* in a schema graph for S_D and find the *refTables* that reference *T*. For each *refTable*, we check to see if it contains any *tid*, i.e., *tuple-ids* from *T*, by probing DataIndex again. If no *tid* appears in *refTable*, we retrieve the forms that contain both *T* and *refTable*, and record these "dead" forms in *X* so that we can remove them later. In the second step, we get *F'* the same way we do in DI-AND, but instead of returning *F'*, we subtract *X* from *F'* and return the difference as the final result.

There are other types of dead forms that DIJ does not eliminate. Consider the query "John XML." Suppose John has published some papers and there are papers about XML, but John has never worked on XML. In this case, DIJ will return a form joining Authors with Publications, but the form will return no result. Although it does not eliminate all types of dead forms, in our evaluation section we found that it does eliminate a sizable fraction of "dead" forms.

One concern about DIJ is that the extra probes to DataIndex with *refTable:tuple-id* can be slow when *tuple-id* appears in many tuples in *refTable*. Fortunately, since the purpose of the extra probe is to see whether there exists at least one tuple with the given *tuple-id*, we do not need to return all the tuples that have *tuple-id* in *refTable*. Instead, the index probe can stop as soon as it finds out whether *tuple-id* has an entry in *refTable*.

4. DISPLAYING RETURNED FORMS

4.1 Ranking Forms

Since our keyword search approach could still return a significant number of forms, it is important to rank the returned forms so that those that are more relevant according to some ranking function are placed higher in the result. Unlike keyword search over RDMBSs where tuple-tree results are generated on-the-fly, we generated and indexed the forms offline, so the forms can be interpreted as "documents," and sorting and finding top-*k* documents is very fast, as shown in our experiments (Tables 4 and 6). We ranked the forms based on the scoring function of a Lucene [15] index, which we used to implement our keyword search approaches. As documents are retrieved, a Lucene index assigns each query-document pair a normalized TF/IDF score with optional boosting. The Lucene score for a query Q and a document D is:

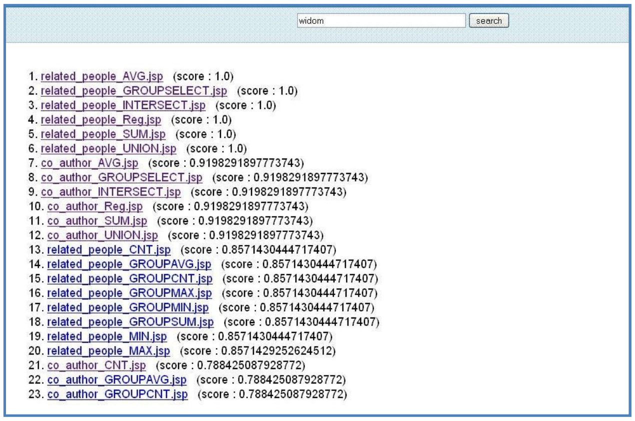
$$\text{score}(Q,D) = \text{coord}(Q,D) * \text{queryNorm}(Q) * \Sigma_{t \text{ in } Q}(\ \text{tf}(t \text{ in } D) * \text{idf}(t)^2 * t.\text{getBoost}() * \text{norm}(t,D)\ )$$

where coord(Q,D) is a score factor based on the number of query terms found in D, queryNorm(Q) is a normalizing factor, tf(t in D) is the term frequency of t in D, idf(t) is the inverse-document frequency of t, t.getBoost() is a search-time boost of t, and norm(t,D) is an index-time boost. In our implementations, we set both search-time boost and index-time boost to one.

We noticed that although our keyword search approach was effective in retrieving relevant forms while pruning irrelevant ones, it was not always easy to spot the right form from the result. The problem arises when we have very specific forms. As form specificity increases, the number of forms created from each skeleton template increases, and forms based on the same skeleton template – which we call "sister forms" (the skeleton template being a "family name") – become increasingly similar.

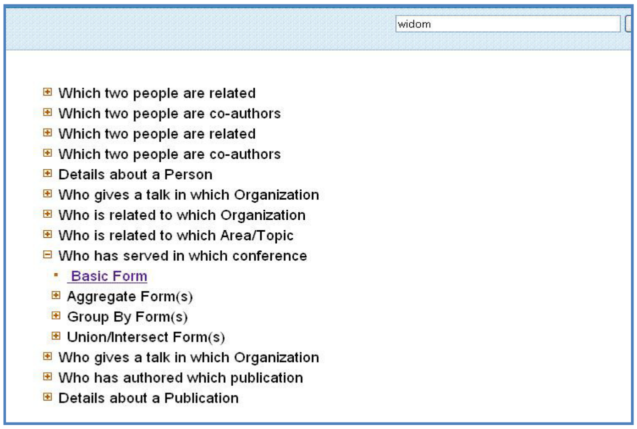
While usability is the motivation behind more specific forms in that a specific form requires less customization, having many sister forms per family interestingly presents a different usability issue. That is, sister forms from the same family get the same ranking score unless a query distinguishes among them with terms that are unique to some of them. We may see the first *k* forms from one family, the next *k* forms from another family, and so on. Moreover, when a query is relatively vague, there is not enough information to determine the user's intent. Therefore, the "right" form may not be in the first couple families, and could get pushed low in the result when each family has many sister forms.

To see these problems, consider the query "Widom." Figure 8 shows the result of this query as a flat list of forms. The sister forms have similar descriptions and the same scores. Also, with



1. related_people_AVG.jsp (score : 1.0)
2. related_people_GROUPSELECT.jsp (score : 1.0)
3. related_people_INTERSECT.jsp (score : 1.0)
4. related_people_Reg.jsp (score : 1.0)
5. related_people_SUM.jsp (score : 1.0)
6. related_people_UNION.jsp (score : 1.0)
7. co_author_AVG.jsp (score : 0.9198291897773743)
8. co_author_GROUPSELECT.jsp (score : 0.9198291897773743)
9. co_author_INTERSECT.jsp (score : 0.9198291897773743)
10. co_author_Reg.jsp (score : 0.9198291897773743)
11. co_author_SUM.jsp (score : 0.9198291897773743)
12. co_author_UNION.jsp (score : 0.9198291897773743)
13. related_people_CNT.jsp (score : 0.8571430444717407)
14. related_people_GROUPAVG.jsp (score : 0.8571430444717407)
15. related_people_GROUPCNT.jsp (score : 0.8571430444717407)
16. related_people_GROUPMAX.jsp (score : 0.8571430444717407)
17. related_people_GROUPMIN.jsp (score : 0.8571430444717407)
18. related_people_GROUPSUM.jsp (score : 0.8571430444717407)
19. related_people_MIN.jsp (score : 0.8571430444717407)
20. related_people_MAX.jsp (score : 0.8571429252624512)
21. co_author_CNT.jsp (score : 0.788425087928772)
22. co_author_GROUPAVG.jsp (score : 0.788425087928772)
23. co_author_GROUPCNT.jsp (score : 0.788425087928772)

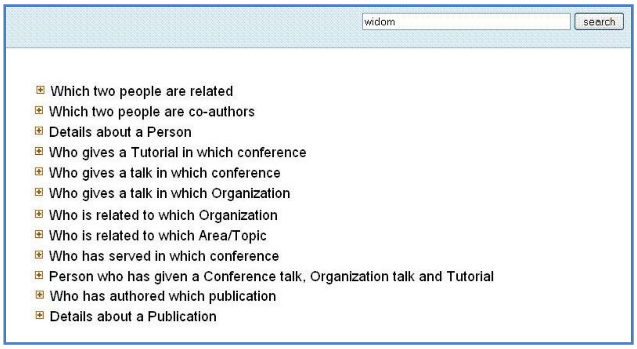
Figure 8. The result for the query "Widom" displayed as a flat list. The form for the information need "for which conference Widom has served" is ranked 127th (not shown in the picture).



⊞ Which two people are related
⊞ Which two people are co-authors
⊞ Which two people are related
⊞ Which two people are co-authors
⊞ Details about a Person
⊞ Who gives a talk in which Organization
⊞ Who is related to which Organization
⊞ Who is related to which Area/Topic
⊟ Who has served in which conference
- Basic Form
⊞ Aggregate Form(s)
⊞ Group By Form(s)
⊞ Union/Intersect Form(s)

⊞ Who gives a talk in which Organization
⊞ Who has authored which publication
⊞ Details about a Publication

Figure 9. Displaying the result in Figure 8 with the first grouping approach. The form for the same information need is in the 9th group and the first form within the group.



⊞ Which two people are related
⊞ Which two people are co-authors
⊞ Details about a Person
⊞ Who gives a Tutorial in which conference
⊞ Who gives a talk in which conference
⊞ Who gives a talk in which Organization
⊞ Who is related to which Organization
⊞ Who is related to which Area/Topic
⊞ Who has served in which conference
⊞ Person who has given a Conference talk, Organization talk and Tutorial
⊞ Who has authored which publication
⊞ Details about a Publication

Figure 10. Displaying the result in Figure 8 with the second grouping approach. The advantage of this approach is that each family is shown only once.

"Widom" as the only term, it is impossible to tell more specifically what the user is looking for. Suppose the user actually wants to find out in which conference "Widom" has served. The right form for this information need – the basic form (i.e., for the *SELECT* class) for serve_conf – is ranked 127th, lower than forms from other families such as related_people, coauthor, and person. However, the ranking makes sense for "Widom" as there are more tuples containing this term in those tables. To address this problem, we propose to group sister forms and present these groups to users, so that users know about all the groups at a high level and can "drill down" into a group if one looks relevant.

4.2 Grouping Forms

We explore collapsing similar forms into groups at query time, so that users can expand on a promising group and drill down to choose among these forms. The basic idea is similar to the automatic categorization of query results proposed by Chakrabarti et al. [4]. Given a list of forms ordered by each form's score, our first approach comprises two steps:

1) Form first-level groups by grouping consecutive sister forms with the same score.
2) In each first-level group, group forms by the four query classes described in Section 2.3, and display the classes in the order of *SELECT*, *AGGR*, *GROUP*, and *UNION-INTERSECT*. The order is fixed because forms in the same group have the same ranking scores.

Figure 9 shows the result for the "Widom" query with this approach. For the information need "for which conference Widom has served," the right form is the first one in the 9th group, which we think is easier to spot than being the 127th form in a flat list. Each first-level group has a description that applies to all the sister forms in that group. This description is the same as that of the form for the *SELECT* class in the family.

One problem with this approach is that when two sister forms have different ranking scores such that they are not consecutive, they join different first-level groups. However, these groups still have the same description and could confuse users. For example, in Figure 9, the first and third groups have the same description "which two people are related"; the second and fourth groups also have the same description.

To tackle this problem, we consider a second approach: we first group the returned forms by their table, then order the groups by the sum of their scores. Figure 10 illustrates this alternative for the same query. The advantage of this approach is that each group of forms based on the same skeleton template is only shown once. However, the forms within each first-level group are always shown in the same order. Therefore, for some queries, the right form could be last in a first-level group, even though this first-level group is ranked high. The first approach does not have this problem because we group the forms by the scores first. We used the first approach in our experiments.

5. EXPERIMENTS

5.1 Experimental Setup

We built an end-to-end prototype system to evaluate the combining of keyword search and forms for posing ad hoc structured queries over a relational database. We implemented the search interface with Perl CGI scripts, used MySQL as the back-end database, and used an Apache Web Server to host the service. We ran the experiments on a Red Hat Enterprise Linux 5 workstation with 2.33 GHz CPU and 3 GB RAM.

We used DBLife as the data set and generated a set of forms, *F1*. It had 14 skeleton templates, one for each of the 5 entity tables and the 9 relationship tables. From each skeleton template, we created, based on the four query classes in Section 2.3, 1 *SELECT* template, 5 *AGGR* templates (one for each

	SQL Query

T1	SELECT p.name
	FROM give_tutorial g, conf c, person p
	WHERE g.cid = c.id AND g.pid = p.id AND c.name like 'VLDB'

T2	SELECT t.name
	FROM related_topic t, person p
	WHERE t.pid = p.id AND p.name like 'jeff naughton'

T3	SELECT p.name
	FROM serve_conf s, person p, conf c
	WHERE s.pid = p.pid AND s.cid = c.id AND c.name = 'SIGMOD' AND s.assignment = "chair"

T4	SELECT per.name
	FROM pub, person per, write_pub w
	WHERE w.pub_id = pub.id AND w.pid = per.id and w.pos = 1 and pub.cites > 5

T5	SELECT count(*)
	FROM co_author c, person p1
	WHERE c.pid1 = p1.id AND p1.name like 'david dewitt'

T6	SELECT p2.name
	FROM co_author c, person p1, person p2
	WHERE c.pid1 = p1.id AND p1.name like 'david dewitt'
	UNION
	SELECT p2.name
	FROM co_author c, person p1, person p2
	WHERE c.pid1 = p1.id AND p1.name like 'jeff naughton'

Figure 11. The SQL queries for the information needs.

aggregate), 6 *GROUP* templates (one for each aggregate and one without aggregates), and 2 *UNION-INTERSECT* templates (for UNION and INTERSECT), so *F1* had 14 * 14 = 196 forms total.

Using *F1* and the 6 information needs presented in the following, we conducted a real-life user study with 7 graduate students in the Computer Sciences Department.

T1: Find all people who have given a tutorial at VLDB
T2: Find topics of areas related to Jeff Naughton.
T3: Find people who have served as the SIGMOD PC chair.
T4: Find the first author of all papers cited more than 5 times.
T5: Find the number of people who have co-authored a paper with David Dewitt.
T6: Find people who have published with David DeWitt or Jeff Naughton.

To find the answers, users submitted keyword queries to the system, browsed the results to identify the correct form (there was one such form for each query), and filled in values on the form. All users successfully found the correct answer for each information need. Figure 11 shows the equivalent SQL queries. We report and discuss the results in the following sections.

5.2 Comparing Naïve, Double-Index, and Double-Index-Join

We first compared Naïve-OR, Naïve-AND, DI-OR, DI-AND, and DIJ, with respect to the difference between AND and OR semantics, the effect of query rewrite, the importance of filtering dead forms, and performance. Because of space constraint and because users entered similar sets of keywords for the same information need, we consider the inputs of just one of the users. Below, Qi is a keyword query for the information need Ti:

F1	Number of Forms Returned				
	Naive-OR	Naive-And	DI-OR	DI-AND	DIJ
Q1	14	0	168	42	42
Q2	28	0	182	28	28
Q3	0	0	142	28	28
Q4	28	28	142	28	28
Q5	14	0	196	14	14
Q6	0	0	196	182	168

Table 1. Number of forms each approach returns for each query.

F1	Average Number of Forms Returned					
	T1	T2	T3	T4	T5	T6
DI-AND	44	48	38	28	129	64
DIJ	44	46	38	28	116	56

Table 2. Average number of forms returned by DI-AND and DIJ for the 6 information needs based on the inputs of 7 users.

F1	Flat Rank of the Correct Form				
	Naïve-OR	Naïve-And	DI-OR	DI-AND	DIJ
Q1	1	N/A	29	1	1
Q2	15	N/A	28	1	1
Q3	N/A	N/A	28	1	1
Q4	1	1	29	1	1
Q5	4	N/A	29	4	4
Q6	N/A	N/A	51	27	12

Table 3. Rank of correct form in a flat list of returned forms.

F1	Response Time (in milliseconds)				
	Naïve-OR	Naïve-AND	DI-OR	DI-AND	DIJ
Q1	5.0	5.0	43.3	34.5	79.8
Q2	8.0	6.0	40.0	24.0	102.7
Q3	10.0	6.0	52.0	52.0	60.4
Q4	16.0	7.0	42.9	35.0	33.6
Q5	13.0	5.0	44.6	42.0	61.8
Q6	12.0	5.0	44.3	32.1	41.9

Table 4. Response time by each approach for each query.

Q1: "tutorial vldb"
Q2: "jeff naughton research area"
Q3: "sigmod chair"
Q4: "paper citation"
Q5: "david dewitt coauthor"
Q6: "dewitt naughton"

Table 1 shows the number of forms returned by each approach for each query. The correct form was returned in all cases except when no forms were returned. Five queries contain data terms (that are not on any form): "vldb" in Q1, "jeff naughton" in Q2, "sigmod chair" in Q3, "david dewitt" in Q5, and "dewitt naughton" in Q6. Therefore, Naïve-AND returned no forms for these queries, and Naïve-OR returned no forms for Q3 and Q6, which contain only data terms. DI-OR was ineffective as a means to narrow one's search – it returned all 196 forms for Q5 and Q6, and a majority of forms for the remaining queries. Though disappointing, it makes sense that DI-OR returned the most forms – both the OR semantics and the query rewrite strategy add more forms to the result. By comparison, DI-AND returned significantly fewer forms for most queries. DIJ, based on DI-AND but eliminates "dead" forms, returned the same set of forms as DI-AND for all queries except Q6, for which DIJ reduced the number of returned forms from 196 to 168.

F1	Flat Rank				Group Rank			
	H	**M**	**L**	**#F**	**H**	**M**	**L**	**#G**
T1	1	1	1	44	1	1	1	3.14
T2	1	1	69	46	1	1	7	3.7
T3	1	1	1	38	1	1	1	2.7
T4	1	15	15	28	1	2	2	2
T5	4	21	21	116	1	2	4	11.57
T6	1	12	12	56	1	1	6	4

Table 5. The highest (H), median (M), and the lowest (L) flat and group ranks for each queries, and the average number of forms (#F) and groups (#G) returned, based on the results of 7 users.

F1	Pose query (sec)	Find the right form (sec)	Fill out the form (sec)	Total average time (sec)	Standard Deviation (sec)	Median (sec)
T1	7.0	12.3	5.3	24.6	13.1	23.0
T2	7.5	23.9	14.8	46.1	48.0	26.0
T3	7.5	18.0	25.6	51.1	31.4	36.0
T4	12.0	79.7	15.2	106.9	56.6	123.0
T5	19.0	46.9	7.7	73.6	29.9	80.0
T6	14.0	64.0	15.2	93.2	47.8	78.0

Table 6. The breakdown of the time of using DIJ by 7 users.

Extending this comparison, Table 2 shows the average number of forms returned by the two approaches for the 6 information needs based on the inputs of all users. Returning an average of 27.9% of the 196 forms, DIJ detected and filtered dead forms for T2, T5, and T6. The number of dead forms depends on the schema and the specific query for an information need. In our study, the user queries were generally specific enough that there were not too many dead forms.

Table 3 shows the rank of the correct form in a flat list of forms returned by each approach for each query. This metric is not applicable when Naïve-OR and Naïve-AND did not return any forms; however, when they did, the rankings were mostly on par with DI-AND and DIJ. The only exception is Q2, for which the correct form was ranked 15th by Naïve-OR and 1st by DI-AND and DIJ – with Naïve-OR, the top 14 forms are for the topic table, whereas the right form is for related_topic, which joins person and topic. This example demonstrates how ranking can differ when data terms are ignored in query evaluation, and that when the right form is not in the first family of sister forms, its position in a flat list of forms could be low.

Compared to other approaches, DI-OR consistently gave the correct form the worst rank. Lastly, comparing DI-AND and DIJ on Q6, we see that after eliminating dead forms, not only did DIJ return fewer forms, but also it ranked the correct form significantly higher (from 27th to 12th).

Table 4 shows the response time (in milliseconds) of each approach for each query. As expected, Naïve-OR and Naïve-AND took much less time than approaches using the query-rewrite strategy. Naïve-AND was faster than Naïve-OR because more terms tend to lead to fewer forms returned. DI-AND was faster than DI-OR for the same reason. DIJ took the most time, but the average of the absolute time of the six queries was still only 68.2 ms, hardly noticeable by humans.

We conclude that DIJ is the best approach because it can handle a mix of data terms and form terms in the query, filter "dead" forms, and give the best flat rank, with minor additional overhead in response time. We use DIJ in the following discussion.

5.3 Ranking and Displaying Forms

The effectiveness of keyword search largely depends on whether users can quickly spot the right form from the list of returned forms. In general, we assume that users look at the list top-down, so the higher the right form appears on a list, the better. In Section 4.1, we have discussed that sister forms often receive the same score and appear together. Therefore, when the returned forms are presented as a flat list, the right form could appear low on the list, and users may have difficulty spotting it. With our prototype, we explored using the first grouping approach described in Section 4.2 to group together sister forms with the same ranking score.

In the user study, we recorded for each information need, the keyword query by each user and the corresponding list of forms. From these results, in addition to recording the *flat rank* – the position of the right form on the flat list, we recorded a *group rank* – the position of the first-level group that contains the right form. We do not report the position of the sub-groups within the first-level group that contained the right form because all the forms under the same first-level group have the same scores and the sub-groups are ordered the same by their query classes.

Table 5 shows the highest (H), the median (M), and the lowest (L) flat and group ranks, along with the average number of forms (#F) and groups (#G) returned, for each information need by DIJ. Under "Flat Rank," we can see that the highest ranks are near or at the top, which is encouraging, especially when the total number of returned forms is quite large for some queries. However, some of the median and the lowest ranks are significantly worse. In other words, for the same information need, the correct form could still have very different rankings even with similar keyword queries. Also, we notice that the rankings are worse for the last three queries. One possible reason is that while the last three information needs are more complicated, the users did not necessarily enter more specific keywords. For example, for T6, Q6 is simply "dewitt naughton," which could be just about anything related to David DeWitt and Jeff Naughton.

In contrast, we can see that the group ranks are much more consistent – the correct form for each query belonged to a top-7 group. With the grouping approach, a user needs to expand a group and drill down to find the right form, so one concern is that the user needs to drill down many levels before finding the correct form. This issue depends on how the query classes are organized. In the study, there were at most three levels, including the first-level group, and for most queries the right form was right under the first-level. Therefore, overall, we believe that grouping forms helps users find the right form more easily, especially when a keyword query for a complicated information need is relatively vague.

5.4 User Interaction with Keyword Search and Forms

In our user study, we measured the time (in seconds) spent by each user over three segments: 1) posing the initial keyword query after learning the information need, 2) finding the right form from the result of the last keyword query, and 3) filling in values on the right form. Since six users used just one keyword query in the process, and the other one used two, the sum of

F2	Average Number of Forms Returned						
	T1	T2	T3	T4	T5	T6	T7
DI-AND	187	174	174	127	94	462	214
DIJ	187	174	167	127	94	427	187

Table 7. With respect to the user queries for these information needs, DIJ returned on average 27.8% of the 700 forms.

F2	Flat Rank				Group Rank			
	H	M	L	#F	H	M	L	#G
T1	1	15	15	187	1	2	2	35.9
T2	15	15	272	174	2	2	20	15.4
T3	1	1	1	167	1	1	1	12.3
T4	1	1	15	127	1	1	2	8.3
T5	4	18	18	94	1	2	2	18
T6	13	27	27	427	1	2	2	23.4
T7	1	1	57	187	1	1	4	1.6

Table 8. The flat and group ranks were still very good even though *F2* had a lot more forms than *F1*.

F2	Response Time (in milliseconds)						
	Q1	Q2	Q3	Q4	Q5	Q6	Q7
DIJ	102	109.2	70.4	43.2	56.2	114.2	96.8

Table 9. Response times by DIJ. Though longer than those with *F1* (see Table 4), humans probably would not notice the difference.

these three segments is roughly equal to the end-to-end time. Table 6 shows each component, the averaged sum of the components, the standard deviation, and the median.

The total time ranged from 24.6 to 106.9 seconds. Of the three segments, the time to find the right form from the results of a keyword query is of most interest to us because it reflected the usability of our "keyword search to forms" approach. For the first three information needs, this segment took about 30 seconds or less, whereas for the last three, it took between 73 and 107 seconds. The reason is likely that the last three information needs were more complicated. Overall, for real information needs on a real-world data set, real users in our experiment found the right form in a reasonable amount of time.

5.5 Impact of Adding Forms

Although the results of the queries against *F1*, which has 196 forms, are encouraging, they beg the question: how good is the keyword search approach when there are a lot more forms? To gain insight to this question, we created a new set of forms, *F2*, which comprised *F1* and forms for all combinations of equijoins involving 2 different relationship tables and the person table on the attribute pid. There were 9 choose 2, or 36 skeleton templates for these equijoins (which were similar to Ex_{EQ} except here we used only 2 relationship tables), leading to 36*14 = 504 forms. As a result, *F2* had 196 + 504 = 700 forms total. Using F2, we re-evaluated the keyword queries of the 7 users, and considered a new information need.

T7: Find people who have given a conference talk and given a tutorial.

T7 corresponds to a SQL query doing an equijoin on give_conf_talk, give_tutorial, and person:

```
SELECT   p.name
FROM     give_conf_talk c, give_tutorial t, give_org_talk
         o, person p
WHERE    c.pid = t.pid AND t.pid = o.pid and o.pid = p.id
```

Table 7 shows the average number of forms returned by DI-AND and DIJ with respect to our user queries against *F2*. Compared to Table 2, we can see that, as expected, a lot more forms were returned with *F2*. However, DIJ returned essentially the same fraction of the total number of forms with *F1* (27.9%) and *F2* (27.8%). Also, comparing DI-AND and DIJ, we can see that DIJ filtered more dead forms from *F2*. This result shows that filtering dead forms is even more important when the set of forms being considered is large.

The counterpart of Table 5, Table 8 shows the flat and group ranks when DIJ was used to evaluate the user queries against *F2*. Comparing the two tables, we can see that even though *F2* has a lot more forms, the flat and group ranks with respect to these queries were still very good. While the flat ranks may be outside top 10 for some queries, the median group ranks were consistently in the top 2. The result demonstrates the advantage of grouping sister forms when a large set of forms is used.

Finally, Table 9 shows the response time by DIJ for the queries Q1 to Q6, and Q7, which is "conference tutorial," against *F2*. The times were significantly longer than those for Q1 to Q6 against *F1* (see DIJ column in Table 4), since more forms were retrieved and returned. However, with the longest of those being 114.2 ms, users most likely would not notice a difference.

To summarize, when we evaluated the same queries against a much larger set of forms, DIJ returned roughly the same fraction of the total number of forms. More importantly, the rankings of the correct forms remained very good, and response times were still relatively insignificant for humans to notice.

6. RELATED WORK

QBE (Query-By-Example) [17] and NFQL (Natural Forms Query Language) [6] are languages for non-programmers to query and update a relational database. Skeleton tables of a database or customized forms are presented to users, who can fill in the blanks with "examples" to specify query constraints. Though much simpler than SQL, they still require an understanding of the relational model, and could give users trouble when the schema is complicated. These works do not consider the problem of choosing from a set of forms, probably because only a few (but very general) forms are used.

Jayapandian et al. described an approach that automatically generates forms for a database based on a sample query workload [11], and more recently, an approach to automatically create a form-based interface, with the goal of maximizing expressivity while respecting specified bounds on interface complexity [10]. Since their goal is to create a small set of forms, they also do not consider the problem of choosing from a set of forms. Instead, when the forms do not support a user query, they allow users to modify an existing form [12].

An alternative to writing structured queries is keyword search over databases [e.g., 1, 3, 8, 9, 14]; however, it has limited ability to exploit structured data. For example, aggregations, projections, range queries, and queries that specify which attribute must contain a desired constant, are all outside the scope of "basic" keyword search. Liu et al. [14] proposed to automatically distinguish between schema terms and value terms in a keyword query, and adopted a new ranking strategy for handling keyword queries with schema terms. Compared to using forms, this approach has little support for structured queries. Indeed, a major motivation for our approach is to allow users who do not want to use SQL to still be able to leverage the advantage of querying structured data.

To extend basic keyword search over databases to be more expressive, BANKS [3] proposed supporting the "attribute = value" construct in keyword queries. For this approach to be effective, users either need to know the schema elements, or the system needs to be able to map user-specified attributes to system attributes. By contrast, with our approach, users do not use any operators in keyword search, and the schema elements are already presented on the forms.

The idea of incorporating simple support for structured querying in keyword search falls under the more general question: just how structured users prefer a query language to be? To gain some insight, Bernstein et al. [2] introduced four query interfaces, each representing a different degree of "structuredness," and conducted a user study to evaluate which one the users prefer and how they perform. The results showed that while the users disliked the constraints of a fully structured formal query language, they also seemed lost with the freedom of a full Natural Language Processing (NLP) approach. The authors suggested that a restricted query language is better than natural language because of its "guidance effect."

AVATAR [13] supports precision-oriented search tasks, in which users are looking for specific information buried within a few documents in a large collection (e.g., a certain phone number in emails). Its approach – extracting concepts from text, and given a keyword query, generating structured queries over the concepts – is similar to our approach; however, there are two major differences. First, its motivation is to exploit structured data to improve search over unstructured data, whereas ours is to help users query a structured database. Second, at query time, Avatar generates structured queries dynamically, whereas we simply identify relevant forms from existing ones.

A nuisance in query processing is waiting for a long time before realizing the query result is empty. Luo proposed to efficiently detect empty-result queries by "remembering" results from previously executed, empty-result queries [16]. While this work is in the same spirit as DIJ, the two are very different. DIJ filters forms that lead to empty results based on users' keyword queries, whereas Luo's work identifies an empty-result structured query when one arrives at the database.

Noting that unstructured data is easier to create, query, share, and less sensitive to change than structured data, and that structured data supports much richer queries, Halevy et al. [7] wanted to "import" the advantages of unstructured data to structured data management. The bulk of this work described mechanisms to facilitate the creation and sharing of structured data, but not querying structured data.

7. CONCLUSION

In this paper, we investigate the approach of using keyword search to lead users to forms for ad hoc querying of databases. We consider a number of issues that arise in the implementation for this approach: designing and generating forms in a systematic fashion, handling keyword queries that are a mix of data terms and schema terms, filtering out forms that would produce no results with respect to a user's query, and ranking and displaying forms in a way that help users find useful forms more quickly. Our experience suggests several conclusions. One is that a query rewrite by mapping data values to schema values during keyword search, coupled with filtering forms that would lead to empty results, is an attractive approach. Another conclusion is that simply displaying the returned forms as a flat list may not be desirable – some way of grouping and presenting similar forms to users is necessary.

Substantial scope for further work remains. In particular, developing automated techniques for generating better form descriptions, especially in the presence of grouping of forms, appears to be a challenging and important problem. Also, exploring the tradeoffs between keyword search directly over the relational database and our approach is an intriguing topic. Certainly forms can express queries not expressible in basic keyword search; however, it is possible to ameliorate this somewhat by augmenting basic keyword search with some structured constructs. Discovering which approach is most useful to users is an open question. Even on queries that are expressible by both approaches, there is a basic philosophical difference: our approach returns a ranked list of relevant queries (expressed in what are hopefully user-friendly forms), whereas the traditional keyword search approach returns ranked lists of relevant answers. Determining under which circumstances each is most appropriate is an important task.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. ICDE, 2002.

[2] A. Bernstein and E. Kaufmann. Making the semantic web accessible to the casual user: Empirical evidence on the usefulness of semiformal query languages. IEEE Transactions on Knowledge and Data Engineering, under review.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. ICDE, 2002.

[4] K. Chakrabarti, S. Chaudhuri, S. Hwang. Automatic Categorization of Query Results. SIGMOD 2004.

[5] P. DeRose et al. DBLife: A Community Information Management Platform for the Database Research Community. CIDR 2007 Demo.

[6] D. W. Embley. NFQL: The Natural Forms Query Language. ACM Transaction Database System, 1989.

[7] A. Halevy et al. Crossing the Structure Chasm. CIDR 2003.

[8] V. Hristidis, L. Gravano, Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. VLDB 2003.

[9] V. Hristidis, Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. VLDB, 2002.

[10] M. Jayapandian, H. V. Jagadish. Automated Creation of a Form-based Database Query Interface. VLDB 2008.

[11] M. Jayapandian, H. V. Jagadish. Automating the Design and Construction of Query Forms. ICDE 2006.

[12] M. Jayapandian, H. V. Jagadish. Expressive Query Specification through Form Customization. EDBT 2008.

[13] R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, H. Zhu. Using Structured Queries for Keyword Information Retrieval. IBM Technical Report RJ 10413.

[14] F. Liu, C. Yu, W. Meng, A. Chowdhury. Effective Keyword Search in Relational Databases. SIGMOD 2006.

[15] Lucene. http://lucene.apache.org/

[16] G. Luo. Efficient Detection of Empty-Result Queries. VLDB 2006.

[17] M.M. Zloof. Query-by-Example: the Invocation and Definition of Tables and Forms. VLDB 1975.