Keyword Search on Form Results

Aditya Ramesh*
Stanford University
aramesh1@stanford.edu

S. Sudarshan
IIT Bombay
sudarsha@cse.iitb.ac.in

Purva Joshi
IIT Bombay
jpurva@cse.iitb.ac.in

ABSTRACT

In recent years there has been a good deal of research in the area of keyword search on structured and semi-structured data. Most of this body of work has a significant limitation in the context of enterprise data since it ignores the application code that has often been carefully designed to present data in a meaningful fashion to users. In this work, we consider how to perform keyword search on enterprise applications, which provide a number of forms that can take parameters; parameters may be explicit, or implicit such as the identifier of the user. In the context of such applications, the goal of keyword search is, given a set of keywords, to retrieve forms, along with parameter values, such that result of each retrieved form executed on the corresponding retrieved parameter values will contain the specified keywords. Some earlier work in this area was based on creating keyword indices on form results, but there are problems in maintaining such indices in the face of updates. In contrast, we propose techniques based on creating inverted SQL queries from the SQL queries in the forms. Unlike earlier work, our techniques do not require any special purpose indices, and instead make use of standard text indices supported by most database systems. We have implemented our techniques and show that keyword search can run at reasonable speeds even on large databases with a significant number of forms.

1. INTRODUCTION

Keyword search has been extremely successful in the context of Web search. There has been a good deal of research on applying keyword search to structured data over the past decade, for example [3], [11] and [1], with a number of systems built to support such keyword search. However, these systems have thus far not seen wide adoption. A primary reason is that they expose the underlying schema to users, which is not appropriate for lay users. Even expert users would find it hard to deal with the complexity of the schema in large ERP systems. Thus, users of database-backed applications typically only interact with the database through (Web) form interfaces, where they can fill in parameter values (with some values, such as the current user's identifier, automatically filled in) and view the result of executing the form. Such form based interfaces are ubiquitous, with ERP systems being a classic example of a mission critical system based on form interfaces.

Form-based interfaces allow users to retrieve required information in a convenient manner. However, enterprise applications today typically have a very large number of forms, and it is not trivial for a user to even find out what forms exist and what information they provide. Even if the user knows what forms are available, it is not possible in general to know what parameter values would lead to results containing the desired keywords; as an example, a form may take a student ID and return the name and other information, but given the name of a student, there is no way for a user to find the student ID, unless a search form is created for that purpose. Even if such a search form were available, to get information about a student, the user would have to first find the student ID using the search form, then navigate to another form that provides desired information, and paste the student ID in that form, which can be rather tedious.

Keyword search is a promising alternative for retrieving information from such form-based applications. In a form-based setting, the goal of keyword search is to retrieve forms, along with associated parameter values, such that executing the retrieved form on the retrieved parameter values would return a result containing the specified keywords. When there can be multiple answers (whether multiple forms, or multiple different parameter values for a form), there is an associated need to rank answers, and present the highest ranked ones to users.

Consider the query 'Silberschatz course', whose goal is to find what courses Silberschatz teaches. A form that takes an instructor ID as parameter and returns the name of the instructor and the courses taught by the instructor may return a result with the above keywords, given the ID of instructor Silberschatz (we assume the keyword "course" matches metadata, or static words in a form). Even if there is no form as above in the system, a form that takes a department ID as a parameter and returns the names of instructors and courses they teach may return the above answer, with the CS department as the parameter value. Our goal is to retrieve such (form, parameter-value) pairs. There may be other more specific forms that return instructor/course information for specified semesters, or less restrictive forms that return instructor/course information for all courses in the university. It is important to be able to rank such forms; for example, ranking could be based on the length of the form result.

As another example, given a query 'Programming Languages Database Systems' if there is a form that returns the courses taken by a specified student (identified by ID), a form search system can return such a form along with the IDs of students for whom the form result contains all the keywords, i.e. they have taken both

*Work done while visiting IIT Bombay

Proceedings of the VLDB Endowment, Vol. 4, No. 11

courses. The query would also return other results as well, such as professors who have taught both courses, departments that offer both courses, and so on.

Some enterprise applications provide keyword based search to retrieve relevant forms, but these are restricted to search on text that describes the form, rather than on text in form results. For example, if we search for 'professor course', the system should return the form that describes which professors teach which courses, provided that the form's metadata contains the terms "professor" and "course". However, parameter values, such as the IDs of instructor or department names, which are required to execute the forms, are not returned by such systems. Moreover, if we phrase our query as "Silberschatz course", with the goal of finding what courses are taught by Silberschatz, or as "Silberschatz database", with the goal of finding what form results related Silberschatz and database, then no form will be returned if (as expected) the keyword terms "Silberschatz" and "database" do not appear in the descriptive text of any form.

An approach that provides the functionality we desire is to materialize form results for each possible parameter value, and building an index on the materialized results, treating each result as a document. An optimized version of this approach is described in [8]. This approach can be expensive in a setting where there are a large number of forms, each of which can take a large number of different parameter values, resulting in a large number of materialized form results. Although disk size is no longer a limitation for many systems these days, the bigger problem lies in maintaining the materialized results in the face of updates.

Ideally, materialized results should be maintained incrementally; in this case, in the face of an update, the system must identify which (form, parameter-value) combinations are affected. This problem is not addressed in [8], but the keyword-independent query inversion techniques we describe can in fact be used to create a materialized views that help in the above task. However, our experimental results show that even such incremental maintenance can be very expensive given a large number of materialized form queries. For systems where keyword queries are used less frequently than normal form interfaces, the overhead of view maintenance is imposed on every update, for the benefit of the occasional keyword query, which is not a reasonable tradeoff.

Moreover, support for incremental view maintenance on most database systems is restricted to simple types of queries. Queries used in forms are often more complicated, and cannot be maintained incrementally; recomputation requires executing the form queries on a very large number of parameter values, and would be unreasonably expensive.

To avoid the above problems, we provide solutions to the problem of keyword search on forms, based on executing queries generated by a rewriting of queries used in the forms.

We address the keyword search on forms problem in the following setting. Each form contains one or more underlying queries (hereby defined as form queries) which are executed when the form is submitted; the form result consists of a static textual part, and a dynamic part based on the results of the queries. For simplicity, we assume initially that each form contains only one query although Sections 6.4 and Appendix C.3 deal with forms having more than one query. Each form has an associated set of (zero or more) parameters for which values must be provided; we assume that all parameters are mandatory, and do not consider the case of optional parameters. We assume that these values are directly provided to the queries, and the results of the queries are returned directly to the form result. Thus, technically speaking, we address the problem of keyword search on parameterized queries.

Unlike earlier work based on materializing form results and indexing the materialized results, our approach works directly on the queries and the underlying database. As a result, there is no need to create and maintain form results. However, we face the challenge of "inverting" parametrized queries; normally the query is executed, with the given values for its parameters, to get a result. However, for a given query, and a given set of keywords, we need to find parameter values that would generate a result containing the specified keywords.

The contributions of this paper are as follows:

- After addressing the issue of safety (in Section 4), we present (in Sections 5 and 6) a two-step algorithm for inverting parametrized SQL queries, which takes a set of keywords, and generates parameter values such that the query executed on the parameter values would generate a result containing the keywords. Step 1 of our algorithm can also be used to create a materialized view which would be useful for identifying parameters for which a form result is affected by a given relation update.
- We describe several optimizations to the basic algorithm, which can improve performance significantly.
- With certain keywords, there may be large number of results (a result is a ⟨form-id,parameter(s)⟩ pair). We discuss (in Section 7) how to rank results in a meaningful fashion.
- We have implemented our algorithms on two different database systems (PostgreSQL and Microsoft SQL Server), using two different datasets (including a real academic database application from IIT Bombay, and an application based on TPCE). We present (in Section 8) results of a performance study showing the practicality of our proposed techniques, and the benefits over the alternative of materializing form results.

2. RELATED WORK

The problem of keyword search on form interfaces was addressed earlier by [8]; their approach is based on indexing materialized form results, but with an optimization called predicate-based indexing. They do not provide details on how to incrementally maintain the index in the face of updates. There has been work on database search in the enterprise search industry; however we are not aware of any publicly revealed approaches other than crawling the application forms and applying text indexing on the crawled result.

Combining keyword search with databases has been a very active area of research, including systems such as BANKS [3], DBXPlorer [1], DISCOVER [11], and algorithms proposed by [7], [13] and [12]. However, the goal of these papers is fundamentally different from our application in two major aspects. First, the above body of work deals directly with the database data and schema, and does not have any concept of forms, or form queries. Some of the above work actually generates SQL queries from the given keywords; however, the generated SQL queries are basically join queries that help to find connections between tuples containing the keywords, and there is no notion of parameters. In contrast, form queries can be quite different; for example, a form may contain a query that selects all courses taught by an instructor, without a join. A keyword query on forms which returns the above form can also be executed as a keyword query on the underlying data, and could be satisfied by a self join query, with the instructor identifier as the join attribute. However, the results would not be presented in a manner that is intuitive to users, and the keyword query may return connections that are meaningless to lay users.

The notion of QUnits was proposed by [14] to make keyword query results more relevant by defining (parametrized) queries that gather related information, and which can be subsequently queried; however [14] do not provide algorithms for answering keyword queries. QUnits can in fact be considered as forms, and our techniques can be applied to perform keyword queries on QUnits.

A somewhat different problem of form search is addressed by [6, 2]; in contrast to our work, they assume that a schema is given, but forms do not exist a-priori, and have to be generated by the system. They generate a space of forms based on SQL queries. The main contribution is to find a form that may be relevant to a given set of keywords; however, they do not generate parameter values, and further do not even guarantee that there exists a parameter value for a retrieved form, whose result would contain the given keywords. As an example of the limitation of that approach, if a form takes a employee ID and returns the name, a keyword search on name would retrieve the form, but not provide the employee ID; without that value, the user would have no idea how to use the form. (Technically, in the approach of [6], parameter values are optional, but if the employee ID is omitted, the form would return names of all employees.)

3. SYSTEM MODEL

We assume that the system at hand has a set of forms $F = \{f_1, f_2, \cdots f_n\}$, and each form $f_i \in F$ takes a set of parameters P_i. Formally, the goal of our application is as follows: given a set of keywords $K = \{k_1, k_2, \ldots, k_m\}$, to return a ranked list of (form, parameter-value) pairs (f_j, p_j) such that form f_j when executed on parameter values p_j returns a result that contains all keywords in K; the metrics for ranking answers are discussed later.

We assume initially that each form is defined by a single parameterized query, which uses all the form parameters; later, in Section 6.4, we discuss how to handle forms with multiple queries which may each use a subset of the parameter values.

We also assume initially that the result of a form executed with a specified value for the parameters contains exactly the result of the query executed on the given parameter values; extensions to allow static text in the form result are discussed later in Appendix C.2. Some applications construct form queries dynamically, based on which several optional parameter values are provided by a user; we do not handle such dynamically constructed queries, and require that queries in a form be statically fixed.

We use the term *query inversion* to refer to the following task: given a query and a set of keywords, retrieve all possible tuples of parameter values such that the query result with each tuple of parameter values contains the given keywords.[1] Although we present our techniques using relational algebra, our actual implementation is based on SQL.

We use the following university schema as a running example in this paper.

prof(ID,*name*, *dept*)
course(CID,*title,dept*)
teaches(ID,CID,*year*, *sem*)

Here teaches(ID) and teaches(CID) are foreign keys referencing prof(ID) and course(CID) respectively.

4. UNSAFE QUERIES

There are certain queries for which the solution set for the parameters is infinite. As an example, assume that the relation prof contains two records: (1,'John','CS') and (2,'Bob','EE') and consider the following parametrized query Q:

$$\Pi_{name}(\sigma_{dept<>\$DeptID}(prof))$$

If the keyword query is 'John', then the possible parameter values for Q are all strings except 'CS'. This even includes values that are not valid departments because the clause "$dept <> \$DeptID$" will always evaluate to true as long as DeptID is not 'CS'. Thus, the solution set for this query is unbounded.

Formally a parametrized query is considered to be *unsafe* if it returns a non-empty result for an unbounded number of parameter values, including values that are not legal database values. This principle is similar to the concept of domains and safety in tuple relational calculus (see, e.g. [15]). There are a number of reasons why a query may be unsafe, including parameters used only in inequality conditions (e.g. $P.name < \$N1$); parameters used only in disjunctive conditions (e.g. "$P.dept = \$D1 \vee P.ID = \$D2$"); parameters used only in the right hand input of an antijoin $\overline{\ltimes}$ (corresponding to not-in or not-exists subqueries in SQL); parameters used only in the right input of a left outer join $\exists\bowtie$, and symmetrically, left input of a right outer join, and either input of a full outer join, and parameters used only in one input of a union ($\Cup$) operation. See Section A of the appendix for a more detailed explanation.

A *sufficient syntactic condition for safety* of a parametrized query is that every parameter is equated to a relation attribute values in some selection condition, no selection condition has a disjunctive condition, and the only operators above that parameter occurrence are $\sigma, \Pi, \bowtie$ and the aggregation/grouping operator γ; $\exists\bowtie$/$\bowtie\sqsubset$ can also be allowed, provided the parameter occurs in the input that is preserved, i.e. left input of $\exists\bowtie$ and right input of $\bowtie\sqsubset$.

Note that the above condition rules out parameters that only occur as arguments to a function, for example $fn(\$DeptId)$, since arbitrary functions cannot be inverted (and may be unsafe).

A *sufficient semantic condition for safety* is that the query can be rewritten to a form that satisfies the above syntactic conditions. Such a semantic condition is particularly useful for SQL queries containing subqueries; such queries can be represented in relational algebra using semijoin/antijoin, or in some more complex cases, by using an "apply" operator [9]. A variety of decorrelation techniques are available for such queries, which can be used to remove subqueries, and syntactic safety can be checked on the rewritten query. To handle such queries, we assume they have been rewritten to satisfy the syntactic conditions.

Another sufficient semantic condition, which exploits knowledge of the application that uses the parametrized query, is that every parameter can only take on only values from a finite domain; in such cases, for a parameter $\$P1$ the query Q can be rewritten as $Q \times \sigma_{P=\$P1} D_P$, where D_P is a relation with a single attribute P, containing all values that parameter $P1$ can take. It should be clear that the rewritten query is syntactically safe with respect to parameter $\$P1$.

Although we primarily address safe queries, we show later how to handle inversion of certain cases of unsafe queries by using a special value ('$*$') which represents the set of all possible values.

5. INVERTING SIMPLE QUERIES

In this section we consider how to handle simple queries containing σ, Π and $\times$. We consider more complex queries later, in Section 6.

In general, inversion is done in two steps:

1. The first step is independent of the keyword query; it takes as input the given query, and gives as output another query

[1] The idea of query inversion arose out of conversations with Surajit Chaudhuri.

which we call the keyword independent inverted query. This step can be done as part of preprocessing, before any keyword queries are submitted to the system.

2. The second step uses the keyword independent inverted query, along with the given keywords, to form a query that gives the keyword search result, i.e. parameter values, corresponding to the original query.

Given multiple forms (for now, assuming each has only a single query) the same process is applied to each query.

5.1 Keyword Independent Inversion

Suppose query Q is of the following form, with k parameters:

$$\Pi_{A1,...,An}(\sigma_{pp}(r_1 \times r_2 \times \ldots \times r_n))$$

where parameterized predicate pp is of the form $B1 = \$B1 \wedge B2 = \$B2 \wedge ... \wedge Bk = \$Bk) \wedge p$, where p is a predicate that does not contain any parameters, and Bj is the column to which the parameter $\$Bj$ is bound. Then the keyword-independent inverted query KIQ is as follows:

$$\Pi_{B1,B2,...,Bk,A1,...,An}(\sigma_{p}(r_1 \times r_2 \times \ldots \times r_n))$$

The idea is that the query generates all possible parameter values that could have given a non-empty result. As of now, there is no restriction on the keywords, these restrictions will be added subsequently as selections on the query. We keep track of the source of each attribute in the output of KIQ, i.e. whether it is a parameter or an original projection attribute.
Example: Suppose we are given the query

$$\Pi_{name,title}(\sigma_{dept=\$DeptID \wedge sem=\$Sem}(prof \bowtie teaches \bowtie course))$$

Although this query uses natural join, the rewriting is identical to the case described above, where the relations have a Cartesian product. The resultant keyword independent inverted query is

$$\Pi_{dept,sem,name,title}(prof \bowtie teaches \bowtie course)$$

Note that the selection condition in this rewritten query is empty, i.e. true, so the selection has been omitted.

Observe that as a condition to ensure safety, we required that every parameter is equated to an attribute of some relation; we call such attributes that are equated to query parameters as *parameter attributes*.

The basic idea of keyword-independent inversion is to create a query that outputs the parameter attributes along with the values that result from query execution on each parameter binding. This task is trivial for queries that involve only joins (including Cartesian products) and selections, since all attributes are available in the query result. If the query additionally includes projection, it is straightforward to rewrite the query to pull the projection to the top of the query, to bring it to the form discussed above. We discuss other relational algebra operations later in Section 6.

It is worth noting that the keyword-independent inverted query can be stored as a materialized view, which can be used to find which (form, parameter) values are affected by a database update. Specifically, if a particular (form, parameter) value is affected, then one of the rows in the corresponding materialized view, with that parameter value, will be affected by the update (i.e., inserted, deleted, or updated). Thus, this materialized view can be used to maintain a materialized form index such as the one proposed in [8]. Note that [8] do not address how to maintain the index.

5.2 Inverting Single Keyword Queries

To process a given keyword query on a given form query, we first invert the form query, and then add selections based on the given keywords to the WHERE clause of the inverted query. The selections ensure that the given keywords occur in the result. We first handle the case of a single keyword, and then address the more general case of multiple keywords.

Given a query Q, a keyword independent inverted form KIQ of Q, and a single keyword $K1$, the resulting inverted query is as follows:

$$\Pi_{B1,B2,...,Bk}(\sigma_{Contains((B1,...,Bk,A1,...,An),K1)}(KIQ))$$

where $Contains((B1, \ldots, Bk, A1, \ldots, An), K1)$ denotes that $K1$ is contained in at least one of the attributes of $B1, \ldots, Bk, A1, \ldots, An$.

Note that we need to add the parameter attributes $B1, \ldots Bk$ to the *Contains* clause even if they are not part of the original query result, since many applications output parameter values directly to the form result, without (redundantly) retrieving the value of the corresponding parameter attribute Bi in the query. Adding the Bi's ensures that parameter binding results from such forms are included in the inverted query result.

For the case where $B1, \ldots, Bk, A1, \ldots, An$ are attributes of a single relation r, the above $Contains$ predicate can be efficiently evaluated almost exactly as shown, provided a text index has been built on all attributes of relation r (or at least those that appear in the query result); the syntax shown is modeled on SQL Server, where the predicate can be written as
"$contains((B1, \ldots, Bk, A1, \ldots, An), K1) > 0$",
but other databases such as PostgreSQL offer equivalent features.[2]

As an example, given the preceding example query with the projection result containing only the attribute *name*, and a keyword 'John', the inverted query taking the keyword into account is

$$\Pi_{dept,sem,name}(\sigma_{Contains((dept,sem,name),'John')}(J))$$

where J denotes $prof \bowtie teaches \bowtie course$.

In case $A1, \ldots, An$ contain attributes from more than one relation, the predicate can be split into one contains predicate per relation, which are combined disjunctively. For example, of $A1, A2$ are from $r1$ and $A3, A4$ are from $r2$, the predicate can be written as where $Contains((A1, A2), K1) \vee Contains((A3, A4), K1)$. Equivalently, the inverted query can be shown as the union of two queries

$$\Pi_{B1,B2,...,Bk}(\sigma_{contains((A1,A2),K1)}(KIQ)) \cup \Pi_{B1,B2,...,Bk}(\sigma_{contains((A3,A4),K1)}(KIQ))$$

In the preceding example, with the projection list containing both *name* and *title*, which come from relations *prof* and *course*, respectively, the inverted query can be expressed either as

$$\Pi_{dept,sem}(\sigma_{P1 \vee P2 \vee P3}(J))$$

where $P1$, $P2$, and $P3$ denote, respectively, *Contains((name),'John')*, *Contains((dept,sem),'John')* and *Contains((title),'John')*, or as

$$\Pi_{dept,sem}(\sigma_{P1}(J)) \cup \Pi_{dept,sem}(\sigma_{P2}(J)) \cup \Pi_{dept,sem}(\sigma_{P3}(J))$$

In practise, we found the formulation using union was faster on both PostgreSQL and SQL Server, and we use this version in our performance study.

[2]Values from non-text types, such as integer or date, can be included in a full-text index by casting to text type (in PostgreSQL), or by adding a (persisted) computed column of text/varchar type containing the textual representation of the value of the column.

5.3 Inverting Multiple-Keyword Queries

Keyword queries using multiple keywords can be handled using the AND semantics, in a straightforward manner, as follows. Given a query Q, and keywords $K1, \ldots, Kn$, let the inverted query using Ki be denoted by IQi. Then the overall inverted query is

$$IQ1 \cap IQ2 \cap \ldots IQn$$

Continuing with our earlier form query example, if the keyword query were $\{Avi, database\}$, the inverted query would be as follows:

$$\Pi_{dept,sem}(\sigma_{C1 \vee C2 \vee C3}(J)) \cap \Pi_{dept,sem}(\sigma_{C4 \vee C5 \vee C6}(J))$$

where J denotes *prof* $\bowtie$ *teaches* $\bowtie$ *course*, and $C1, C2, \ldots C6$ denote, respectively, *Contains((name),'Avi')*, *Contains((dept, sem), 'Avi')*, *Contains((title),'Avi')*, *Contains((dept, sem),'database')*, *Contains((name),'database')* and *Contains((title),'database')*. Alternatively, the query can be expressed as:

$$(\Pi_{dept,sem}(\sigma_{C1}(J) \cup \Pi_{dept,sem}(\sigma_{C2}(J) \cup \Pi_{dept,sem}(\sigma_{C3}(J))$$
$$\cap (\Pi_{dept,sem}(\sigma_{C4}(J) \cup \Pi_{dept,sem}(\sigma_{C5}(J) \cup \Pi_{dept,sem}(\sigma_{C6}(J))$$

As a special case, if the query result is guaranteed to have at most one result for a parameter binding, instead of intersecting two queries, we use a conjunction of the Contains predicates. For example if (unrealistically) each (*dept*,*sem*) combination had exactly one result above, the query would be

$$\Pi_{dept,sem}(\sigma_{(C1 \vee C2 \vee C3) \wedge (C4 \vee C5 \vee C6)}(J))$$

where the Ci are as defined earlier. We call the above optimization the *primary key* optimization.

6. COMPLEX QUERIES

We now consider the case of inverting more complex queries containing other relational operations. For the case of queries using only the basic operations σ, Π and $\bowtie$, it was easy to rewrite the queries to get the parameter attribute in the result of the inverted query. This task is more complicated with other relational operations, and we consider those operations in this section; a few cases such as set difference are covered in Appendix C.

Once the keyword-independent inverted query has been generated, the task of adding the keyword conditions can be done as described earlier in Sections 5.2 and 5.3, since that step does not depend on the structure of either the original query or the keyword-independent inverted query.

6.1 Aggregation Operations

As discussed earlier, for the case of queries with projections at the top, we added the parameter attributes to the projection list. However, if there is an aggregate operation on top of the projection, adding parameter attributes to a projection can change the number of duplicates. But the more important question is, how to make parameter attribute values available above an aggregation operation. We solve both problems as outlined below.

Consider an aggregation operation ${}_G\gamma_{aggfns(A)}(E)$ where G denotes the group-by attributes, and $aggfns(A)$ denotes the aggregation functions and the attributes they are applied on. Suppose that the set of parameter attributes from expression E are $B1, \ldots, Bn$. We then rewrite the expression as

$${}_{G,B1,\ldots,Bn}\gamma_{aggfns(A)}(\mathit{INVQ}(E))$$

where $\mathit{INVQ}(E)$ denotes the inverted query generated from E. (As should be clear, we are informally defining a recursive procedure to perform the inversion, but omit formal details for lack of space.)

Observe that by adding the parameter attributes to the group-by list, the rewritten query returns the same aggregate result for any particular binding of values to the parameter attributes as the original query with the specific parameter binding. This property holds even if some of the parameter attributes are used in the aggregation operation (for a specific parameter binding, these would be constants).

6.2 Intersection Operation

Keyword-independent inversion can be done for the intersection operations $\cap$ as discussed below. The case of union queries are more complicated, and is discussed later in Section 6.3.

A first incorrect attempt to handle queries with intersection would be to independently invert the inputs to the intersection, and then to compute their intersection. Unfortunately, however, different parameters may be used in each of the inputs, leading to different parameter attributes being present in the different inverted inputs, and a direct intersection is not possible.

When the intersection is a set intersection (the default in SQL) an alternative is to turn intersection into a join. Specifically, given a query $Q = Q1 \cap Q2$ where the attribute names of $Q1$ and $Q2$ are identical, the inverted query is simply

$$\mathit{INVQ}(Q1) \bowtie \mathit{INVQ}(Q2)$$

where each of $Q1$ and $Q2$ is inverted with respect to just the parameters that occur in it. Note that if each parameter occurs in only one of $Q1$ or $Q2$, the natural join above would equate only the original attributes of $Q1$ and $Q2$, but if any parameter appears in both $Q1$ and $Q2$, the natural join would ensure that both have the same value.

For example, given the query

$$\Pi_{ID}(\sigma_{name='Mike'}(\mathit{prof})) \cap \Pi_{ID}(\sigma_{sem=\$Sem}(\mathit{teaches}))$$

the inverted query is

$$\Pi_{ID}(\sigma_{name='Mike'}(\mathit{prof})) \bowtie \Pi_{ID,sem}(\mathit{teaches})$$

For the case of multiset union, the duplicate count matters only if the result is used in some aggregation or set operation above. If it is not, the multiset union can be replaced by a set union; but if it is used, the conversion of intersection to a join would change the number of duplicates. To handle such a case, we can add extra parameter attributes to each of $\mathit{INVQ}(Q1)$ and $\mathit{INVQ}(Q2)$, so they have the same schema. The value of the parameter attributes added thus is set to a special constant "top" value "*", which matches with every concrete value. The rewritten query is then

$$\mathit{INVQ}(Q1) \Cap \mathit{INVQ}(Q2)$$

where $\Cap$ denotes intersection taking the special semantics of the "*" value.

For example, suppose parameter $P1$ occurs in $Q1$ and $P2$ in $Q2$, and if we have tuples $t1$ and $t2$ from the two inverted inputs, that are identical on all other attributes, but have extra parameter attributes $B1$ (with value $v1$) and $B2$ (with value $v2$) respectively. To make their schemas identical, we add parameter attributes $B2$ and $B1$, with value $*$, to $t1$ and $t2$ respectively. Now the result of intersecting these two tuples has the values $(v1, v2)$ for attributes $(B1, B2)$. If both tuples have a value $*$ for a common parameter attribute, the result also has the value $*$ for that attribute. For the example above, this form of the inverted query is as follows:

$$\Pi_{ID,*}(\sigma_{name='Mike'}(\mathit{prof})) \Cap \Pi_{ID,sem}(\mathit{teaches})$$

6.3 Union Operation

The case of union operations is more complicated, since for a particular parameter binding some of the keywords may be present only in one input to the union, and others may be present only in the other input. Additionally, as in the case of intersection, some parameters may be used in one input and others in the other input, complicating the task of inversion.

Suppose Q = Q1 ∪ Q2. If each subquery Qi has the same set of parameters, the result of the inverting Q is simply *INVQ*$(Q1) \cup$ *INVQ*$(Q2)$, where *INVQ*(Qi) is the keyword-independent inverted query corresponding to Qi. However, if the subqueries have different sets of parameters, the problem is more complicated.

Suppose that Q = Q1 ∪ Q2 has parameters $B1, $B2 and $B3, and suppose that Q1 has parameters $B1 and $B2, while Q2 has parameters $B2 and $B3. The output of the inverted form of Q1 would contain two attributes B1 and B2, while that of Q2 would contain B2 and B3.

6.3.1 Single-Keyword with Union

Consider first the case of a single keyword query, with the keyword K1. The query in the above example is in fact unsafe with respect to a single keyword query; if K1 is contained in an answer for Q1, then the value of $B3 is irrelevant, and it can take any possible value, while if K1 is contained in an answer for Q2, then the value of $B1 is similarly irrelevant. However, it is possible that for specific pairs of keywords K1, K2, the query is not unsafe. However, the lack of safety can be handled specially here, by using the special "top" value * in keyword query answers. If parameter $Bi is not present in subquery Qj, we add a column corresponding to Bi in the result of inverting Qj, but with the constant value *. Thus, the schema of all inverted subqueries of a union becomes the same, and the inversion of Q is simply the union of the inversions of Qi.

Processing a single keyword query is straightforward, except that the result may contain the special value * for one or more parameters. In this case, the query is actually unsafe, but the special value * provides us a finite representation of an infinite number of values. Here, * can be taken to represent "don't care".

6.3.2 Multiple-Keywords with Union

To handle a multi-keyword query, intersection of the results for each keyword can be done as described earlier. However the intersection is made more complicated by the presence of the special * value. For example, given two tuples (*, 'B') and ('A', *) respectively from two relations being intersected, their result on intersection would be ('A', 'B'). This illustrates how it is possible for the overall query result to be safe even though the results for each keyword may be unsafe. Similarly intersecting (*, 'B') and (*,'B') would result in (*,'B').

Unfortunately, database systems do not support the special "don't care" value * when performing intersection (for union, the value * can be treated as a normal value). Intersection taking * values into account can be done in application code, but efficiency remains an issue, since standard techniques for intersection, such as sorting, cannot be applied in a straightforward manner in the presence of the * value. However, there is a implementation technique, which we call the *Keyword-at-a-time (KAT)* implementation, described in Appendix B.1, which allows the query to be processed efficiently, and entirely in the database. The basic idea is to partition the parameter bindings based on which subset of attributes have a * value. Intersection of each pair of partitions can then be implemented by a join on the partitions, equating only non * attributes, followed by projection of appropriate non-* attributes. Finally, the results of the joins are combined by a regular union operation.

In addition, there is an alternative approach, which in effect finds the results (sets of parameter bindings) of inverting each input to the union separately, and then combines the results. This approach, called the *Query-at-a-time (QAT)* approach, is described in Appendix B.2. The QAT approach solves each query with all the keywords, but allowing bindings for which the query result contains only a subset of keywords, using a bitmap to record which keywords are present. It then merges the intermediate results of all queries to find the answer. This is in contrast to the keyword-at-a-time approach, which computes parameter values that satisfy one keyword, across the union of subqueries, and merges the intermediate results from each of keywords to get the final solution.

6.4 Multiple Queries in a Form

So far all of our examples have dealt with forms containing only a single query. However, there are instances of forms that contain multiple queries. For example, a form can be used to access information about a particular student (query $S1$) as well as a list of courses that the student has taken (query $C1$); such a form can be represented by two separate SQL queries.

In the simplest and most common case, the different queries are independent, that is they can be evaluated independently using the form parameter values. Such a form can be handled by considering it as having a single query which is defined as the union of the two separate queries. Of course the attributes of the queries may be different, but we can follow the SQL outer union approach of using the union of the attribute sets, with null values used for attributes that are present in one query but not the other. Thus, any solution for the union operation can also be used to handle forms with multiple independent queries as above.

In the above example, the form can be considered to have a single query $S1 \cup C1$. We discuss the case where the queries are not independent, later, in Appendix C.3.

6.5 Semijoin and Antijoin

Uncorrelated where clause subqueries in SQL lead to semijoins and antijoins in the relational algebra representation. Correlated subqueries can be modeled using the apply operator [9]; but as shown in [9], decorrelation techniques can be used to replace the apply operator by a join, semijoin, or antijoin.

Inversion of queries with a semijoin/antijoin is straightforward for parameters that appear in the left input, since the corresponding parameter attributes are already present in the result.

However, inversion is harder if parameters appear in the right hand side input, since the corresponding attributes do not appear in the result, and cannot be directly added. The solution for the case of semijoins is to use decorrelation techniques such as those in [9] to replace semijoins by joins. The decorrelation technique can in fact be simplified in the absence of aggregation, since we do not care about the number of duplicates in the inverted query result. We omit details for lack of space.

Parameters appearing in the right input of antijoins (corresponding to not-in or not-exists subqueries) are harder to handle. The safety requirement in this case requires that any such parameter must also be equated to an attribute of some other (finite) relation. We can use techniques similar to those described in Appendix C.1 to handle this case; we omit details for lack of space.

7. RANKING AND PRESENTING RESULTS

In general, a keyword query can have multiple answers, and ranking the answers is an important task. For a given form, we display the set of parameter bindings together, rather than mixing up results corresponding to different forms. Thus, the ranking prob-

lem is broken up into two problems: ranking forms, and ranking parameter values within each form.

We experimented with two variants of form ranking. The first variant is based on form result length, favoring forms with short results since they tend to contain more specific information. For example, given a form F1 which retrieves course/instructor information for a specified department, and a form F2 that retrieves courses of a specified instructor, form F2 is likely to have a much smaller size on average. Given a keyword query such as 'Silberschatz database', the form F2 would rank higher and the inverted query for F2 would be executed first. The second variant multiples the average form result length with the number of different parameter values returned as answers to the given query.

In cases where the higher ranked forms provide sufficient answers, inverted queries may not even need to be executed for lower ranked forms.

The exact length of a form result depends on the specific parameter values, which can again be expensive to compute, so we instead use statistics on average form result size. Form result size is in turn estimated as the sum of the average result size of the queries contained in the form; average query result sizes can be precomputed and stored in the database, and need only periodic maintenance. Computing the average query result size can be done either by executing the query on a sample of parameter bindings, or by executing the keyword-independent inverted query, and aggregating on its result to find the number of tuples for each binding (by grouping on the parameter attributes), and then taking the average. We used the latter approach.

Many forms contain more than one query; for example a form displaying student information may first show the name and other key information about the student, and then show the grades obtained by the student. We give higher importance to the occurrence of a keyword in the first query than in the second, when ranking the form. For details of this scheme, and of ranking of parameter values within a form, see Appendix C.5. Extensions to support access control are discussed in Appendix C.7.

8. EXPERIMENTAL RESULTS

In this section we present the results of a study of the performance of our techniques.

8.1 Experimental Setup

The code for our system is written in Java. For our main performance tests, we used PostgreSQL 8.4 as the database, on a machine with an Intel Core 2, 1.86 GHz processor, with 3GB of RAM, running Ubuntu with a Linux 2.6.24 kernel. The application and the database ran on the same machine. We report numbers using a 80 GB 7200 rpm hard disk (Seagate ST380211AS), as well as with a Transcend SSD18M 64 GB solid state disk (SSD) with an eSATA interface.

We report numbers using a real database application, used to handle all academic information at IIT Bombay, with about 1 GB of data, and 90 form interfaces. In Appendix D, we also report numbers using a (fake) application with 4.3 GB of TPC-E data and 16 forms with queries based on the TPC-E queries.

Other than the full-text indices, we used exactly the same set of indices as were present in the live database, which included primary/foreign-key indices and a few more manually chosen indices. Full-text indexes, needed by the Contains operator, are built on all attributes of all tables. We present numbers for cold cache (CC) and warm cache (WC). Details of how we used the full-text indices, as well as details of how we enforced cold cache, are described in Section D.

In response to a keyword query, the system outputs up to 25 relevant forms. All timing numbers that we present are elapsed time for computing the keyword query results. To account for some random variations in measured time, by default we ran each keyword query multiple times, and report average numbers.

8.2 Basic Results and Effect of Optimizations

We implemented a pruning optimization, which does the following. Many of the keywords are present in only some of the relations, and are absent in others; before executing inverted queries, for each keyword we first find which relations contain the keyword, by accessing the corresponding text indices. Using this information, we prune out a form if the set of relations whose attributes appear in the SELECT clauses of the queries in a form do not together contain all the query keywords. Similarly, we prune out subqueries containing a conjunctive selection condition *Contains((Ri.A1,. . ., Ri.An),Kj)*, if we have found that Ri does not contain Kj. Pruning is particularly important as the number of forms increases, since it can potentially help keep the number of inverted query executions under control. The pruning optimization was turned on by default.

The first set of experiments studied the effectiveness of keyword querying in retrieving desired forms. We used a set of keyword queries (described in Appendix D) reflecting common tasks such as finding student information given the name or the student ID, finding course information given keywords describing the course or a course ID, finding grades of student, finding students registered for a course, and so on. The number of keywords ranged from 1 to 4. We compared the following form ranking methods: (a) ordered by average form result size (AVG), and (b) ordered by average form result size multiplied by the number of parameter values in the result for that form (AVGMULT) (we stopped once we found 500 parameter values).

We measure the quality of the results returned as follows. For each task, we identified a particular form as the desired result. We then manually examined the results of the corresponding query, and found the position at which the desired form was present. Across all the queries, the average position at which the desired form was present was 2.42 for AVG and 1.83 for AVGMULT. The maximum positions of the desired form were 6 and 3 for AVG and AVGMULT.

For the above set of queries, we measured the execution time using the QAT inversion method for forms with multiple queries, on cold cache. With flash disk, the execution time ranged from 0.8 to 10.8 seconds, with an average of 4.8 seconds, while for hard disk the time ranged from 3.5 to 49.5 with an average of 18.9 seconds, with cold cache. With warm cache, the time for hard disk as well as flash ranged from 0.6 to 10.9 seconds, with an average of 3.6 seconds; the timings were almost identical since there was almost no disk access on these queries. The KAT inversion method generated the same set of results as the QAT method, but took marginally more time, with an average of 5.1 seconds on flash cold cache, 3.8 seconds on flash warm cache, 19.5 seconds on hard disk cold cache, and 3.9 seconds on hard disk warm cache.

Overall, the results show that keyword search runs with reasonable speed with flash disk, even with cold cache, and given current trends it is quite reasonable to assume that enterprise application data will fit on flash disk for all but the largest enterprises. Execution times can be reduced by optimizations sketched in Section 9, and by using a faster CPU/disk. For example, we could reduce cold cache query execution times for some sample inverted queries by up to 40% by running them on a system with a higher end hard disk (WD5000AACS, with 9 msec. avg. seek time versus 13 msec. for the ST380211AS), and a faster processor (Intel Core i5, 3.2 GHz).

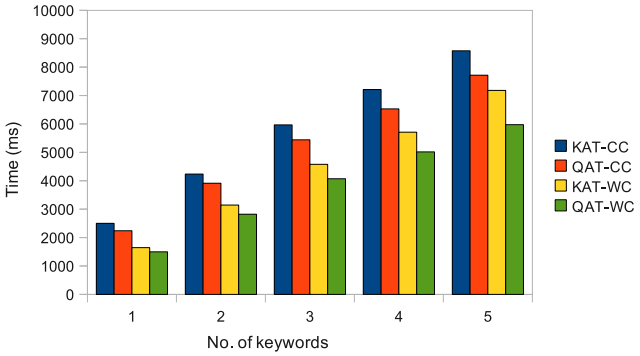

Figure 1: Performance on Academic database .

We also compared QAT and KAT methods, with the number of keywords varying from 1 to 5. We used a single 5 keyword query, and used each of its size k subsets as the set of queries using k keywords. The results for the case of forms on the academic database, running on PostgreSQL, using flash disk, are shown in Figure 1, with separate numbers for cold cache (CC) and warm cache (WC). As can also be seen from the figure, both KAT and QAT scale slightly sub-linearly with the number of keywords, for both cold and warm cache, and QAT performs slightly better than KAT.

Next we studied how the time taken for processing a keyword query scales with an increasing number of forms. The results, described in detail in Appendix D, show that the time increases sub-linearly with increasing number of forms.

8.3 Performance of Form Result Indexing

The approach of [8], which is an alternative to ours, is to materialize form results, and build a text index on the materialized results. For queries that can be incrementally maintained, we can implement indexing and view maintenance as described in Appendix C.8, by creating a materialized view for each form query. To test the overheads, we implemented a simplified form of materialization and view maintenance, which materializes and maintains inverted form queries, along with text indices on the materialized relations. For the academic database, the total size of the resultant materialized views along with indices was 1431 MB, on a 1GB database.

We measured the view maintenance performance on an update that added 9 course registrations for one student, measured on a cold cache. View maintenance took 3.6 seconds with a hard disk, and 1 second with a flash disk, for an update that takes a few tens of milliseconds, which is an unacceptable overhead for the academic application. We note that the time is actually an underestimate of the actual cost, since some of the form queries were too complex for the simple view maintenance algorithm we used, so we did not maintain them. Further, the view maintenance overhead increases with the number of form queries, and has to be paid for every update even if keyword queries are used only occassionally. It is also worth noting that some updates may cause a very large number of form results to be recomputed. Even worse, many queries cannot even be maintained incrementally (most databases which support view maintenance have significant restrictions on the queries supported) and may require full recomputation.

9. CONCLUSION

The problem of keyword search on the results of form interfaces is of increasing importance, since such interfaces provide information in a form fit for human consumption. We have presented an approach to keyword search on form results, based on inverting database queries, to return parameter bindings for which the form result contains the given keywords. We have proposed several optimizations of our basic technique and presented a performance study which shows that the proposed techniques are effective and practical for gigabyte sized databases.

As part of future work, we plan to improve the efficiency of query processing by caching inverted queries, creating a merged text index which will avoid the need for separate keyword lookups on each table, and caching mappings of which keywords are present in which tables. We also plan to extend our implementation to work with a larger class of SQL queries, and to handle complex application code with conditional execution of queries, and loops containing queries. We also plan to work on integrating search with access control, which is typically implemented at the form level. We also plan to address the form ranking problem in more detail.

Acknowledgment: We thank Surajit Chaudhuri for discussions leading to the idea of inverting form queries.

10. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. *ICDE*, pages 5–16, 2002.

[2] A. Baid, I. Rae, J. Li, A. Doan, and J. F. Naughton. Toward scalable keyword search over relational data. *PVLDB*, 3(1):140–149, 2010.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. *ICDE*, pages 431–440, 2002.

[4] I. T. Bowman and K. Salem. Semantic prefetching of correlated query sequences. In *ICDE*, pages 1284–1288, 2007.

[5] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[6] E. Chu, A. Baid, X. Chai, A. Doan, and J. F. Naughton. Combining keyword search and forms for ad hoc querying of databases. *SIGMOD Conference*, pages 349–360, 2009.

[7] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. *ICDE*, pages 836–845, 2007.

[8] C. Duda, D. A. Graf, and D. Kossmann. Predicate-based indexing of enterprise web applications. *CIDR*, pages 102–107, 2007.

[9] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. Joshi. Execution strategies for SQL subqueries. In *SIGMOD Conference*, pages 993–1004, 2007.

[10] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *PVLDB*, 1(1):1107–1123, 2008.

[11] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. *VLDB*, pages 670–681, 2002.

[12] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. *SIGMOD Conference*, pages 563–574, 2006.

[13] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. *SIGMOD Conference*, pages 115–126, 2007.

[14] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR*, 2009.

[15] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, sixth edition, 2010.

APPENDIX

A. EXAMPLES OF UNSAFE QUERIES

Unsafe queries can result from any of the following constructs:

1. Queries where at least one parameter is not bound to a database value by an equality condition; for example, a query of the form $P.DeptId >$ '$\$DeptId$' can give the same output for an infinite number of values of \$DeptId.
2. Queries where a parameter value is contained in disjunctive condition which can possibly be satisfied ignoring the parameter value; for example a condition such as $P.DeptId = \$P1\ OR\ P.Id = \$P2$ Then, the solution set can be infinite as any value can be plugged in for the parameter; in the above example if \$P1 = "CS" returns an answer, then every possible value of \$P2 would be a valid answer. A similar problem can arise if parameters are present in queries that are part of the UNION clause.
3. Queries where the parameters are present only in subqueries linked by an antijoin, or a set difference operation (note that SQL *op* ALL operation are also naturally represented by antijoins). A problem arises with empty sets in the right input, because all of the above operators will return the left input as in in these cases. A similar problem also arises with parameters that are present only on the right-hand size of the EXCEPT clause.
4. Queries where the parameters are present in the join clause with the join type being either left, right or full outer join. The reason why these queries are unsafe is because outer joins always preserve the tuples of at least one relation regardless of the constraints specified in the join clause. If the constraint is enforced through a parameter, then the solution set is infinite, as any value can be plugged into the parameter and the tuples of the first and/or second relation(s) will always be returned.

Here are three examples of unsafe queries, pertaining to cases 2-4:

$$Q1 : \Pi_{name}(\sigma_{prof.ID=teaches.ID \vee sem=\$Sem}(prof \times teaches)$$

$$Q2 : \Pi_{name}(prof \overline{\ltimes} \sigma_{sem=\$Sem}(teaches))$$

$$Q3 : \Pi_{name}(prof\ \overline{\ltimes}_{prof.ID=teaches.ID \wedge sem=\$Sem}\ teaches)$$

B. OPTIMIZING UNION INVERSION

In this section we outline two alternative approaches to handling union queries, as well as forms with multiple queries which are treated identically to union queries. The notation in this section assumes that a query Q is a union of queries $Q1, \ldots, Qn$.

B.1 Keyword at a Time (KAT) Implementation

As discussed earlier, the keyword-at-a-time (KAT) approach for handling the union operation is made complicated by the presence of the special "top" value "*", which must be handled specially. We now outline how we can modify the approach to use standard database joins, without requiring special support for the "*" value.

The KAT approach inverts each query in the union with respect to each keyword, and then adds extra columns corresponding to form-parameters that are not used in the query, with the value set to "*". Let the resultant relation corresponding to keyword Ki and query Qj be $R_{i,j}$; each of these relations is materialized as a temporary table. We then define R_i as $\bigcup_j \{R_{i,j}\}$. The next step computes the logical intersection of the R_i's, but using a series of join steps instead of intersections. The result relation R is first set to R_1, and step i computes the logical intersection of R with R_{i+1}.

The logical intersection of two relations r and s is computed as follows. First, both r and s are partitioned into groups, such that each group has an identical pattern of *'s, i.e. all group members have a * in the same attributes. Let the resultant partitions of r and s be denoted as r_i and s_i. Each r_i is then logically intersected with each s_j, by computing $\Pi_L(r_i \bowtie_P s_i)$, where P equates every column of r with the column of s for the same parameter, provided that attribute does not have the value * (all tuples in r_i have the same * pattern, and similarly so do all tuples in s_j). The projection list L is defined as follows: for each parameter p_k, if r_i has a * for p_k but s_j does not, the kth attribute of L is $s_j.p_k$, otherwise the kth attribute of K is $r_i.p_k$. Thus, if both r_i and s_j have the value *, so would the result; if only one of them has the value *, the result would contain the other value, while if both are not *, the result would contain the common value. Thus each tuple in the join result corresponds to a pair of tuples that agree on all non-* attributes, and would have thus been part of the logical intersection; attributes that are non-* in at least one relation are set appropriately.

As described above, the intermediate result R at each step is materialized for use in the next step. However, our current implementation does not materialize the results, but instead creates separate queries for each group in each intermediate R; the queries for the final state of R are combined by a union to get the final result.

B.2 Query at a Time (QAT) Implementation

We now give details of the query at a time (QAT) approach, which first computes those parameter bindings for each query that contain at least one of the given keywords, and then combines these results across queries.

The query-at-a-time (QAT) implementation carries out the following steps:

1. First consider each query Qi separately and do the following: (a) For each keyword K_j, find the parameter values for Qi whose result contains that keyword, i.e. invert the query with respect to K_j. (b) take the union of the parameter values, across all keywords K_j, but with each parameter value additionally annotated with the set of keywords present in the query result with that parameter value; a bitmap is used to represent this set. (This step can be implemented by a minor extension of the union operation, or by a straightforward extended aggregation operation.)

   The result of this step is represented as a relation R_i for each Qi, with one attribute per parameter of Qi, plus an attribute storing the bitmap; the name of the bitmap attribute is set to b_i, so it is unique to R_i. Note that different queries can have different parameters. We filter out tuples from R_i which have a null value for any parameter attribute.
2. The next step is to find parameter value combinations that are common across queries. If all queries had the same parameters, this could be done by a union of the R_i's followed by a grouping step. The basic intuition for handling the general case of different parameters is to do a join of the R_is on their shared attributes.

   By joining the results of the previous step using an inner join, we would ensure that all tuples in the join result agree on the join attributes. However, an inner join would eliminate parameter values from one inverted query that do not occur in another inverted query; such parameter values can still contribute to the final result. A full outerjoin would preserve such parameter values, but may still lose information. For ex-

ample given queries $Q1$ and $Q2$ with parameters (A,B) and (B,C), it is possible that $Q1$ on a particular (A,B) combination, say (a1, b1) returns keywords $K1$ and $K2$, so for a keyword query $K1, K2, C$ should be don't care. A full outerjoin would lose this information if $Q2$ with parameters (b1, c1) contains one or more of the keywords; the (outer)join would then contain only the tuple (a1, b1, c1). Now suppose query $Q3$ with parameters (B,C) set to (b1, c2) contains keyword $K3$, and the keyword query is $K1, K2, K3$. Then even a full outerjoin $(R1 \supset\!\subset R2) \supset\!\subset R3$ would not contain the correct answer (a1, b1, c2). To work around this problem, we use an outer union operation, as outlined below. We use $\Cup$ to denote the SQL outer union operation, which brings all inputs to a common schema by adding required attributes, with their value set to null.

To combine the results across all queries R_i we do the following. We set *result* to R_1, and at each step $i = 2, \ldots, n$, we set

$$result = result \Cup R_i \Cup (result \bowtie_{\theta_i} R_i)$$

with join condition θ_i defined as below.

Before defining θ_i, we note that *result* contains tuples with different "*" patterns, with "*" represented as a null value due to the outer union. One option for computing the join is to partition *result* based on the pattern of null values, similar to the case for KAT. Another option is to use a more complex join condition, as follows. Let $P1, \ldots, Pk$ be the set of all query parameters; each R_i has all or some subset of the parameters Pk. Then, $\theta = C1 \wedge \ldots \wedge Ck$, where Cj is defined as $(result.Pj = R_i.Pj \vee result.Pj$ **is null** $\vee R_i.Pj$ **is null** $)$, if both *result* and R_i contain Pj, and Cj is *true* otherwise. The above disjunction allows matching in case either value is null value, representing "*". The value of Pj projected in the result is null if Pj is null in both input tuples, and is set to the non-null value otherwise.

In addition to the parameter attributes, a new bitmap attribute b is added to the join result, with its value set to the bit-wise OR of the bitmaps b_i. Note that due to the outerjoin, some of the b_is may be null; these are treated as equivalent to the bitmap will all zeros.

3. The join condition described in the previous step can result in poor execution plans, so we make use of the following important optimizations. First, if a parameter Pj is present in all of $R_1 \ldots R_i$, we defined Cj as just $result.Pj = R_i.Pj$, omitting the disjunction, since Pj cannot be null in either input.

   Further, if all parameters Pj present in Qi are also present in all queries already joined into *result*, and vice versa, instead of setting *result* to the outer union of *result*, R_i, and the join result, we set *result* = *result*$\supset\!\subset R_i$, using a natural full outer join on the shared parameter attributes. This condition was in fact satisfied in most cases of forms with multiple queries in our example application.

4. Let the result of the previous step be relation R. Finally, parameter attribute values from tuples from R that have all the bits set in the bitmap attribute b are output as answers. Some of the parameter attribute values may be null, indicating the result contains the given keywords regardless of the value of the corresponding parameter.

C. OTHER CASES

In this section we consider extensions to handle a larger class of forms.

C.1 Set Difference Operation

The case of set difference is a little trickier than intersection, since a query may be unsafe if a parameter occurs only in the right input of set difference. To ensure safety, we require that any such parameter must also be equated to an attribute of some other (finite) relation. Let us denote such a relation corresponding to parameter Pi and $BindRel(Pi)$, and let $BindVals(Pi)$ denote $\Pi_{Pi}(BindRel(Pi))$. Similarly let $Star(Pk)$ denote the constant relation with a single attribute name Pk, and a single tuple with the special top value $*$ for that attribute.

Given the expression

$$Q = Q1 \; - \; Q2$$

where the attribute names of $Q1$ and $Q2$ are identical, suppose that Pi and Pj are two parameters that occur only in $Q2$, and Pk is an attribute that only occurs in $Q1$ (the case of more or less parameters is straightforward and omitted for simplicity). The inverted query is

$$(BindVals(Pi) \bowtie BindVals(Pj) \bowtie INVQ(Q1) \\ - Star(Pk) \bowtie INVQ(Q2)$$

Note that since the special value $*$ matches all concrete values, the set difference has the required semantics: i.e. if for some parameter binding, a particular $Q1$ tuple was also present in $Q2$ that parameter binding would not be present in the above result, and conversely if a $Q1$ tuple was not present in $Q2$, that parameter binding would be in the result.

C.2 Static Text and Forms Without Parameters

Forms often have static text inserted by the application program, which does not depend on database content or on form parameters. We assume that application code that generates the forms has been analyzed, and static text that appear in forms has been indexed; for each keyword, the posting list in such an index contains the identifiers of the forms where the keyword appears as static text. In addition it is often useful to annotate forms with metadata describing the purpose and description of the form, which can be used when searching for forms.

Before executing a keyword query on the queries in a form, all query keywords that appear in static text in that form are removed from the list of keywords, and the remaining keywords are actually used for querying. In a special case, all the keywords in the query may appear in static text, in which case the form parameters don't actually matter. We can use a special value * to denote that all possible values for a corresponding parameter are answers.

Another special case is form queries that do not take any parameter values. Such a form would be an answer to a keyword query if the keywords are part of the form result. Checking this is no different from the usual case, except that the output of inverted query does not have any parameter values; a constant value such as 1 can be used to ensure that the output has at least one attribute. Also, we do not need to execute the inverted query completely, we just need to ensure that its answer is non-empty.

C.3 Multiple Dependent Queries

However, there are forms that can contain multiple queries. For can be handled using the same technique we saw earlier for queries are different doesn't really matter since absent from some of the queries, the special value * identical.

We saw earlier how to handle forms with multiple independent queries. In some forms, however, the result of one query is used as a parameter to a second query. For example, a student roll number

may be used to retrieve a unique student identifier by means of a query Q1, and the identifier may then be used to execute a second query Q2. This situation can be handled by rewriting the second query by adding a join with the first query, and replacing the parameter by a reference to the value from the first query result.

Another common case is where a query Q1 has multiple results, and a loop iterates over these results and invokes query Q2 with parameters set to attributes in the result of Q1. This case can be handled by replacing the loop by a single query which in effect performs a join of Q1 and Q2, as described for example in [4, 10].

C.4 Handling Arbitrary Queries using Parameter Binding Restrictions

Many queries contain restrictions on the parameter bindings that can produce a non-empty result. For example, we can infer that a parameter must be a professor's ID, or a *dept* by analyzing the query. Then, the set of all possible bindings that can lead to non-empty results is exactly the set of values for that attribute in the corresponding relation, such as *prof*.ID or Dept.*dept*Ànother way in which we can infer such a restriction is from the the search form that a user sees; for example, a particular form parameter may come from a drop-down box, which is populated either statically or from a database query.

If each parameter has such a restriction, we can trivially invert a query as follows. Suppose we are given query Q, with parameters B1 through Bm, with each Bi restricted to values in Ri.Bi (if the restriction is from static text in a form, we can create a temporary table Ri containing the relevant values). Then the keyword-independent inverted query is simply

$$(R1 \times \ldots \times Rm) Apply^{\bowtie}(Q')$$

where Q' is simply Q with each parameter $Bi replaced by the corresponding Ri.Bi from the outer level relation. (The LATERAL clause is in standard SQL, although SQL SERVER uses the term CROSS APPLY instead.)

C.5 Extended Ranking Techniques

For form with multiple queries, if in some form result a keyword occurs in the result of an earlier query, that form result could be counted as more important than one where the keyword only appears in the result of a later query. One simple way of giving more importance to keyword occurrences in earlier queries is to treat a multi-query form with queries $q_1, \ldots, q_n$ as a set of n forms, with form F_j containing queries $q_1, \ldots, q_j$. The ranking methods described earlier are applied on each F_i, and the best rank is chosen. We have currently implemented the above scheme manually.

An alternative which we are currently implementing is to modify the queries to track, for each parameter binding in the query result, which queries contained each of the keywords. The occurrence of a keyword in an earlier occurring query can be viewed as providing a higher TF to that keyword in that form. Similarly, statistics about keyword occurrences in the overall database (available from the text index) can be used as a rough estimate of the IDF (with form results treated as documents) of each query keyword. From these statistics, a TFIDF measure can be computed for each parameter binding, and used to rank the bindings for a given form.

We have assumed the AND semantics for keywords, but our techniques are can be modified to support a fuzzy AND, allowing some keywords to be omitted but reducing the score/ranking accordingly. We omit details.

For ranking of parameter values within a single form, we used heuristics; for example, if the parameter value is a year or semester, the current year/semester is given higher preference, if the parameter value is a department, the department that the current user belongs to is given higher preference, and so on.

C.6 Result Presentation

In our implementation, results are displayed as hyperlinks, and pointing at/clicking on a result causes the corresponding form to be executed with the parameter values, and the form result is displayed to the user. Our inversion techniques may return don't care (*) values for certain parameters. If the corresponding parameters are mandatory for the form, we can use domain knowledge of the application to select a meaningful set of values for such parameters, and replace each answer containg one or more *'s by a set of answers with the *'s replaced by the above values.

C.7 Access Control

To provide access control, applications need to have a module that takes a user identifier, a form identifier, and, optionally, parameter values, and can decide if the user is authorized to execute the specified form with the specified parameters. This module can be used to filter query results to return only authorized results. In many authorization systems, some query parameter values, such as the user-identifier in a query, are taken from session parameters such as the user-identifier of the authenticated user. Such parameters can be replaced by the corresponding constant values in the form queries, before the queries are inverted.

C.8 Materializing Form Results

As discussed earlier, materializing and indexing form results is an alternative to our approach. To estimate the cost of using this approach, we materialized the keyword-independent inversion of each form query. Standard techniques for view maintenance such as those described in [15] (Chapter 13) can be used to compute the changes to the form query result when an underlying relation is updated, and the index must be updated correspondingly. We omit details for lack of space.

There are at least two ways to build a full-text index on the result. The first way is to create another view with one tuple per parameter binding, with a text attribute that contains the concatenation of all attribute values from all tuples with the same parameter binding. The view contains a form-id attribute, and all parameter attributes are combined into one single view attribute by concatenating them (with suitable delimiters). The above views across all forms can be combined using a union operation, and materialized. A full-text index can be built on the resultant view. The implementation we used for our performance testing used this approach, but for lack of time omitted the combination across forms, instead building a text index separately on each form.

The second way is to directly use an existing full-text index such as Lucene, and create a (virtual) document corresponding to each tuple in the preceding merged view. Note that in this case the view need not actually be materialized, since Lucene does not insist that the actual documents it indexes be retained after they have been indexed.

D. PERFORMANCE STUDY DETAILS

We now present a few more details of our experimental setup, and additional experimental results, which could not fit in the main body of the paper.

D.1 Experimental Setup

The set of 12 queries we used to study the quality of ranking were as follows. We cannot give all the actual queries since the database we use contains confidential data which cannot be made public.

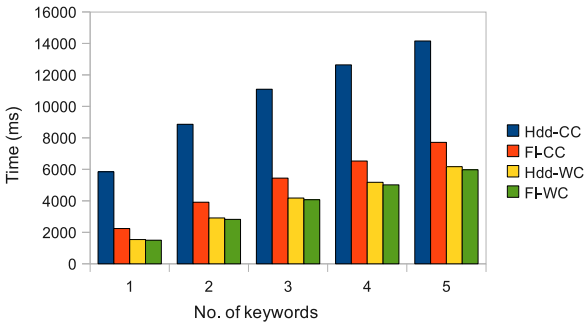

Figure 2: Hard Disk versus Flash, Academic database.

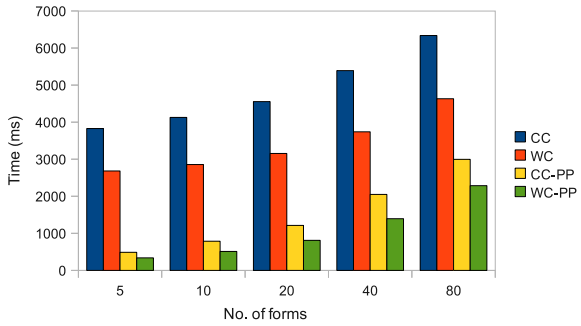

Figure 3: Scalability with no. of forms, Academic database.

The queries modeled common information needs, which were as follows. (a) Given a student identifier (roll number), or a student name, find overall academic information about the student. (b) As above, but find just the grades. (c) Given a course identifier or keywords from the course title, find information about the course. (d) As in (c), but find the students registered for the course, and find if a specified student identifier took the course. (e) Given identifiers of two courses, find students who have taken both courses, using two different sets of descriptive keywords.

Cold cache results were generated by forcing the database to drop all clean buffers (using the DBCC DROPCLEANBUFFERS command of SQL Server, and by restarting PostgreSQL after clearing Linux file system buffers). However, in our context, we run not just one inverted query, but several, for a given keyword query, and it is fine for the later inverted queries to exploit data brought into buffer by earlier queries. Therefore we flush the buffer only once for a single keyword query, instead of once per form query. Warm cache numbers were generated by running the query repeatedly, and ignoring the time for the first execution.

Since it is not possible to create keyword indices on each subset of columns of interest, we create a single index on all columns of each table. To perform keyword lookup on specified columns, we use an index lookup on the all-column index, followed by filtering to ensure the keyword is present in the specified columns. We found that using the PostgreSQL text indexing syntax for filtering was expensive, since it did stemming on the fly. We therefore used a case insensitive substring match of the attributes with the keyword, to implement the filtering step.

D.2 Results

We also ran some experiments on a 4.3 GB TPC-E dataset with 1000 total customers, scale factor of 500 and 100 work days, and had indices on primary/foreign-key attributes. The (fake) application had 16 forms, and we used 7 different queries. These experiments were run on SQL Server 2008. The results here were even better than for the Academic database on PostgreSQL, ranging from 0.2 to 2.3 seconds, with an average time of just 0.6 seconds, on flash disk with cold cache, and using the QAT method. The results using the KAT method were very similar.

We compared the performance on hard disk versus flash, using the QAT method, with an increasing number of keywords, using the same set of 5 keywords as before. The results are shown in Figure 2. As mentioned earlier, cold cache numbers on hard disk are relatively high, since most query execution plans involved indexed nested loops, resulting in random IO, but performance is much better on warm cache.

Next, we studied the advantage of enabling the pruning optimization, using the academic database on PostgreSQL, using a flash disk, and the QAT method, using the same set of keywords used earlier for testing scalability with number of keywords. The improvement due to pruning ranged from 20%-40% (for cold/warm cache resp.) with just 1 keyword, to 95%-98% with 5 keywords. As the number of keywords increased, more queries/subqueries got pruned before execution, accounting for the increase in effectiveness.

Figure 3 shows how the time taken increases with number of forms. For a given number of forms n, we partition the overall set of forms into partitions of size k, and take the average execution time across these partitions. We use a subset of 4 of the 12 queries described earlier for quality of ranking experiments, each with 3 to 4 keywords, and ran the experiments on flash disk. The results for cold cache (CC) and warm cache (WC) appear to indicate that the time taken grows highly sub-linearly with number of forms, with a 20 fold increase in number of forms resulting in less than 2 fold increase in time. However, there is a significant fixed overhead for pruning, which checks, for each keyword, which tables contain the keyword. This overhead does not increase with the number of forms. The bars CC-PP and WC-PP show the time taken after the above pruning step, for cold and warm cache; both these increase by more than 6 fold when going from 5 to 80 forms. Thus, the growth remains sub-linear even in this case.

We found that performance was relatively slow for queries where some keyword was present in the results of some form for a very large number of different parameter values. By default we added a LIMIT L clause to the top level of the inverted queries, to ensure that at most L parameter bindings are fetched for each form. This limit was 10 for ranking by AVG, and 500 for ranking by AVG-MULT. Recall that the top level query is in general an intersection of the results of subqueries, one subquery per keyword. Ideally the database optimizer should use the top-level LIMIT hint, and generate only as many results for the subqueries as are required to get the desired number of final results, using techniques such as those in Bruno et al. [5]. However, we found that the optimizer usually chose plans that generated all results for the subqueries. While we can partially implement the approach of Bruno et al. outside the database, by adding LIMIT clauses to subqueries, there are significant overheads due to the need for recomputation in case not enough answers are generated with the current LIMIT. There are also additional overheads for deciding when recomputation can terminate without any further increase in the LIMIT value. Our current implementation does not include the LIMIT optimization on subqueries.