VLDB manuscript No.
(will be inserted by the editor)

Keyword Search on Form Results

Aditya Ramesh · S. Sudarshan · Purva Joshi · Manisha Naik Gaonkar

Received: date / Accepted: date

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 corresponding 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.

Aditya Ramesh
Stanford University (work done while visiting IIT Bombay)
E-mail: aramesh1@stanford.edu

S. Sudarshan
IIT Bombay
E-mail: sudarsha@cse.iitb.ac.in

Purva Joshi
Sybase, Pune (work done while at IIT Bombay)
E-mail: purvaj28@gmail.com

Manisha Naik Gaonkar
IIT Bombay
E-mail: gaonkar.mani@gmail.com

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], [10] and [1], with a number of systems built to support 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', where the goal is to find courses that 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, consider the 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 courses. The query would also return other form results as well, such as professors who have taught both courses, departments that offer both courses, and so on.

Some enterprise applications support keyword based search for retrieving 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, as long as the metadata of the form 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 relate 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 Duda et al. [7]. 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 by Duda et al. [7], but the keyword-independent query inversion techniques we describe can in fact be used to create 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, but benefits only the occasional keyword query, which is not a reasonable tradeoff. The problem of keyword search on virtual (that is, non-materialized) XML views was addressed by Shao et al. [14]. Their approach is not applicable to SQL, does not consider parameterized queries, and does not have any equivalent of our notion of query inversion. See Section 2 for more details on related work.

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. In contrast, our approach does not require any materialization or view maintenance.

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, but later in Sections 7 and 9.2 we 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" parameterized queries; normally the query is executed, with the given values for its parameters, to get a result. In our context, 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:

- In some cases, given a parameterized query Q and a set of keywords K, it is possible to have an infinite number of parameter values, each of which would generate a result containing the specified keywords. To deal with this problem, we introduce the notion of safety of query inversion, and provide conditions that guarantee safety, in Section 4.
- We then present (in Sections 5 and 6) a two-step algorithm for inverting parameterized queries. Given a parameterized query and a set of keywords, Step 1 of the inversion process creates an inverted query disregarding the keywords, while Step 2 adds keyword conditions to the result of Step 1. Execution of the final inverted query generates the result parameter values. The query inversion approach handles safe queries defined using relational operations such as select, project, join, aggregation, outerjoins and set operations.
  Step 1 of our query inversion algorithm can also be used to create a materialized view which can be used, given an update, to identify the parameter values for which the form result is affected by the relation update. Our technique does not need this feature, but it can be used for incremental maintenance in case form results are materialized using the approach of Duda et al. [7].
- Forms often output results from multiple queries; such forms can be modeled as containing a single query defined as the outer-union of the individual queries. However, when the parameters to the queries are not identical, inversion of the resultant outer-union query is unsafe. In Section 7 we present details of two approaches for inversion of such unsafe union queries, which we call the keyword-at-a-time (KAT) approach, and the query-at-a-time (QAT) approach. We also describe several variants and optimizations of these two approaches.
- With certain keywords, there may be large number of results (a result is a form-id,parameter(s) pair). We discuss (in Section 8) how to rank results in a meaningful fashion.
- Although our algorithm descriptions are in terms of relational algebra, we have implemented our algorithms on SQL queries. We present (in Section 10) results of a performance study using a real academic database application from IIT Bombay. The performance study, using PostgreSQL and SQL Server to evaluate inverted SQL queries, demonstrates the practicality of our proposed techniques, and their benefits over the alternative of materializing form results. The study also demonstrates the scalability of our techniques with respect to number of keywords and number of forms, and compares the alternative algorithms that we have proposed.

The rest of the paper is organized as follows. Related work is described in Section 2, while Section 3 presents the system model, and some assumptions we make. Section 4 addresses the issue of safety of query inversion. Section 5 describes how to invert simple queries with select, project and join, while Section 6 covers safe queries using other relational algebra operations. Section 7 describes two approaches to handling forms with multiple queries (which result in unsafe union operations), including several variants and optimizations of the two approaches. Section 8 describes how to rank the results in case there are multiple results. Section 9 outlines implementation details, optimizations and extensions. In Section 10, we present results of our performance study, while Section 11 concludes the paper and describes directions for future work.

2 Related Work

The problem of keyword search on form interfaces was addressed earlier by Duda et al. [7]; 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 an active area of research, including systems such as BANKS [3], DBXPlorer [1], DISCOVER [10], and algorithms proposed by [6], [12] and [11]. 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 [13] to make keyword query results more relevant by defining (parameterized) queries that gather related information, and which can be subsequently queried; however [13] 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 version of form search is addressed by [5, 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 an 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 [5], parameter values are optional, but if the employee ID is omitted, the form would return names of all employees.)

The problem of keyword search on virtual (that is, non-materialized) XML views was addressed by Shao et al. [14]. Their solution creates a subset of the original dataset (called a pruned document tree, or PDT) making use of path indices, on which the original XML view is executed; their approach guarantees that the result of the view is identical on the original document tree and the PDT. In addition, inverted indices are used to add keyword-containment annotations to the PDT which are used during actual query evaluation to generate only results containing the required keywords. Their approach is not applicable to SQL, does not consider parameterized queries, and does not have any equivalent of our notion of query inversion.

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_i, p_j) such that form f_i 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 7, 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 given values 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 Section 9.2. Some applications construct form queries dynamically, based on which of 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; the translation from relational algebra to SQL is straightforward.

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 parameterized query Q:

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

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

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

Formally, a parameterized 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 set difference or 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 ($\supset\!\bowtie$), and symmetrically, left input of a right outer join ($\bowtie\!\subset$), and either input of a full outer join ($\supset\!\bowtie\!\subset$), and parameters used only in one input of a union ($\cup$) operation.

A *sufficient syntactic condition for safety* of a parameterized query, defined recursively, is as follows. An expression E is *syntactically safe* if one of the following is true:

- E is a relation instance (this is the base case)
- The root of E is a selection (σ) or join ($\bowtie$) operation, and (a) the children of the root operation are safe, and (b) with the selection/join condition θ expressed as a conjunction $\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n$, for every parameter $\$Pi$ that occurs in the condition, there is a conjunct θ_j of the form $\$Pi = Rk.Am$, which equates the parameter to a relation attribute.
- The root of E is a project operator (Π) or an grouping/-aggregation operator (γ), and (a) the children of the root operation are safe, and (b) no parameter is used in any expression that appears in the projection list.
- The root of E is a left outerjoin ($\supset\!\bowtie$) or a right outerjoin ($\bowtie\!\subset$), the children of the root operation are safe, and every parameter $\$Pi$ that occurs in E also occurs in the input that is preserved, i.e., the left input for $\supset\!\bowtie$, and the right input for $\bowtie\!\subset$.

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

The above syntactic conditions for safety are revisited and (in some cases) extended when we cover inversion of the union, set difference, semi-join and antijoin operators in Section 6. In the rest of the paper, unless otherwise specified, we assume that queries satisfy the above syntactic safety condition.

A *sufficient semantic condition for safety* is that the query can be rewritten to a form that satisfies the above syntactic conditions. For example, given a query

$$(\sigma_{\$P1=r.A}(r)) \bowtie (\sigma_{\$P1>s.B}(s))$$

where the subquery $\sigma_{\$P1>s.B}(s)$ fails the syntactic safety condition, we can rewrite it as

$$\sigma_{\$P1=r.A \wedge \$P1>s.B}(r \bowtie s)$$

which does satisfy the syntactic safety condition. As another example, SQL queries containing subqueries can be rewritten before checking for syntactic safety; such queries can be represented in relational algebra using semijoin or antijoin, or in some more complex cases, by using an "apply" operator [8]. A variety of decorrelation techniques are available for such queries, which can be used to remove subqueries (or apply operators, if subqueries are represented using the apply operator), 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 parameterized query, is that every parameter can only take on 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 $\$P1$.

While such a rewriting can be useful in many cases, it could be very inefficient if the domain of P is large. Later in the paper we show how to handle inversion of certain cases of unsafe queries more efficiently, by using a special value ("$*$") which represents the set of all possible values.

Consider a query that takes as parameters a low and a high price, and displays items whose price falls within the specified range. Such a query is unsafe since the parameters are not equated to any relation attribute. Such queries are in fact commonly used in product search applications, to allow users to specify ranges on a variety of attributes. However, for the purpose of keyword search such an unsafe query can be replaced by one which takes an exact value and returns items whose price is equal to the specified price; the replacement query could be safe even though the original is not. Although such a replacement query is not equivalent to the original one, it would permit keyword search on the query result, and the resultant parameter values can in fact be used for (both low and high values of) the range in the original form.

5 Inverting Simple Queries

In this section we consider how to handle simple queries containing σ, Π, $\bowtie$, and $\times$. We consider other relational algebra operations 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 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

Overall, the goal of keyword-independent inversion is as follows: given a parameterized query Q, create a query (the *inverted query*) *INVQ*(Q) which has as attributes all the parameters as well as all the attributes of the original query; further, *INVQ*(Q) must be defined in such a way that for each parameter binding b that leads to a non-empty query result RQ_b, there is a tuple in the result of *INVQ*(Q) corresponding to each tuple in RQ_b, with the value b in the parameter attributes, and vice versa.

Given any query using only the select σ, project (Π), join ($\bowtie$) and Cartesian product ($\times$) operations, we rewrite it into the canonical form

$$\Pi_{A1,\ldots,Am}(\sigma_\theta(r_1 \times r_2 \times \ldots \times r_n))$$

and then apply the keyword inversion technique described below.

Given a safe query Q of the following form, with k parameters:

$$\Pi_{A1,\ldots,Am}(\sigma_\theta(r_1 \times r_2 \times \ldots \times r_n))$$

where parameterized predicate θ is of the form $B1 = \$P1 \wedge B2 = \$P2 \wedge \ldots \wedge Bk = \$Pk) \wedge \theta_0$, where θ_0 is a predicate that does not contain any parameters, Bj is the column to which the parameter $\$Pj$ is bound, and no parameter occurs more than once in θ (we relax this last condition shortly). Then the keyword-independent inverted query *INVQ*(Q) is defined as follows:

$$\Pi_{B1 \textbf{ as } \$P1,\ B2 \textbf{ as } \$P2,\ldots,Bk \textbf{ as } \$Pk,\ A1,\ldots,Am}(\sigma_{\theta_0}(r_1 \times r_2 \times \ldots \times r_n))$$

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 *INVQ*(Q), i.e., whether it is a parameter or an original projection attribute.

For simplicity of presentation, where the attribute name Bi and the parameter name of $\$Pi$ are identical, we replace Bi as $\$Pi$ by just Bi in the projection list; in other cases, we omit the "$" symbol from the parameter name, assuming there is no name conflict between parameters and attributes in the query. In the rest of the paper, for simplicity of presentation, when we describe how inverted queries are created we assume that parameter names are the same as the names of the attribute that they are equated to.

If any of the parameter attributes Bi can take on a null value, we need to add an extra conjunct (Bi **is not null**) to the selection condition θ_0, for each such Bi. This is required, since a value *null* for parameter $\$Pi$ will not equal a null value in Bi.

We can also relax the condition that no parameter occurs more than once in θ as follows. Each parameter must occur in at least one conjunct of the form $Bi = \$Pi$ due to the requirement of safety; we pick one such conjunct for each parameter $\$Pi$, and call Bi the *parameter attribute* of $\$Pi$. Let θ_1 be the result of deleting from θ the conjuncts chosen above for all parameters. Now, θ_1 may contain occurrences of some parameters $\$Pi$; we define θ_0 as the result of replacing in θ_1 all occurrence of $\$Pi$ by its parameter attribute Bi, for every parameter $\$Pi$.

Example 1 Suppose we are given the query

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

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

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

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

It is worth noting that the keyword-independent inverted query can be stored as a materialized view, which can be used to maintain a materialized form index such as the one proposed in Duda et al. [7]. Specifically, the materialized view can be used to find which (form, parameter) values are affected by a database update. 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). Form results for such parameter values must be recomputed and reindexed. Note that [7] does not address how to maintain the index.

5.2 Keyword-Specific Query Inversion

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 inverted query. The selections ensure that the given keywords occur in the result. We first handle the case of keyword queries having only a single keyword, and then address the more general case of queries with multiple keywords.

5.2.1 Inversion for the Single-Keyword Case

Given a query Q, its keyword-independent inverted query *INVQ*(Q), and a single keyword $K1$, the resulting keyword-specific inverted query *INVQ*$(Q, K1)$ is defined as follows:

$$\Pi_{B1,B2,...,Bk}(\sigma_{Contains((B1,...,Bk,A1,...,Am),K1)}(\textit{INVQ}(Q)))$$

where predicate *Contains*$((B1, \ldots, Bk, A1, \ldots, Am), K1)$ checks that keyword $K1$ is contained in at least one of the attributes $B1, \ldots, Bk, A1, \ldots, Am$.

Note that we need to add the parameter attributes ($B1$, $\ldots$, Bk) to the *Contains* predicate 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.

Example 2 Given the inverted query from Example 1, and a keyword 'John', the inverted query taking the keyword into account is

$$\Pi_{\textit{Dept,Sem}}(\sigma_{\textit{Contains}((\textit{Dept,Sem,name,title}),\text{`}John\text{'})}(J))$$

where J denotes the keyword-independent inverted query from Example 1, namely

$$\Pi_{\textit{dept}\ \textbf{as}\ \textit{Dept,sem}\ \textbf{as}\ \textit{Sem,name,title}}(\textit{prof} \bowtie \textit{teaches} \bowtie \textit{course})$$

□

For the case where $B1, \ldots, Bk, A1, \ldots, Am$ are attributes of a single relation r, the *Contains* predicate can be efficiently evaluated, provided a text index has been built on all attributes of relation r (or at least those that appear in the inverted query result). The *Contains* predicate syntax shown is modeled on SQL Server, where the predicate can be written as

$$\textit{contains}((B1, \ldots, Bk, A1, \ldots, Am), K1) > 0$$

but other databases such as PostgreSQL offer equivalent features. See Section 9.4 for implementation details related to the *Contains* predicate.

However, in the above example the *Contains* predicate involves attributes from multiple relations. None of the currently available database systems supports text indices or *Contains* (or equivalent) predicates that span multiple relations. As a result, when $B1, \ldots, Bk, A1, \ldots, Am$ contain attributes from more than one relation, the *Contains* predicate must be split into one *Contains* predicate per relation, which are combined disjunctively. For example, suppose the above set of attributes is $B1, B2, A1, A2$, where $A1, B1$ are from $r1$ and $A2, B2$ are from $r2$; then the predicate can be written as

$$\textit{Contains}((A1, B1), K1) \vee \textit{Contains}((A2, B2), K1)$$

Equivalently, the inverted query can be shown as the union of two queries

$$\Pi_{B1,B2}(\sigma_{\textit{Contains}((A1,B1),K1)}(KIQ)) \cup \Pi_{B1,B2}(\sigma_{\textit{Contains}((A2,B2),K1)}(KIQ))$$

Example 3 For the query from Example 1, which has attributes that come from relations *prof*, *course*, and *teaches*, the inverted query can be expressed as

$$\Pi_{\textit{Dept,Sem}}(\sigma_{P1 \vee P2 \vee P3}(J))$$

where J denotes the keyword-independent inverted query from Example 1, and $P1, P2$, and $P3$ denote, respectively, *Contains((name, Dept),'John')*, *Contains((Sem),'John')* and *Contains((title),'John')*.
Alternatively, the keyword-specific inverted query can be expressed as

$$\Pi_{\textit{Dept,Sem}}(\sigma_{P1}(J)) \cup \Pi_{\textit{Dept,Sem}}(\sigma_{P2}(J)) \cup \Pi_{\textit{Dept,Sem}}(\sigma_{P3}(J))$$

□

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.

5.2.2 Inversion with Multiple Keywords

Keyword queries using multiple keywords can be handled in a straightforward manner for safe queries as follows. Given a query Q, and keywords $K1, \ldots, Kn$, the inverted query *INVQ*$(Q, K1, K2, \ldots, Kn)$ is defined as

$$\textit{INVQ}(Q, K1) \cap \textit{INVQ}(Q, K2) \cap \ldots \cap \textit{INVQ}(Q, Kn)$$

Handling multiple-keyword queries is more complicated in cases where some operations such as union ($\cup$) are used which may result in unsafe queries; we will see how to handle some such unsafe queries later in Section 7.

Example 4 Continuing with our earlier form query example, if the keyword query were $\{Avi, database\}$, the inverted query would be as follows:

$$\Pi_L(\sigma_{C1 \vee C2 \vee C3}(J)) \cap \Pi_L(\sigma_{C4 \vee C5 \vee C6}(J))$$

where J denotes the keyword-independent inverted query from Example 1,
L denotes *Dept*, *Sem*,
$C1$ denotes *Contains ((name, Dept),'Avi')*,
$C2$ denotes *Contains ((Sem), 'Avi')*,
$C3$ denotes *Contains ((title),'Avi')*,
$C4$ denotes *Contains ((Sem),'database')*,
$C5$ denotes *Contains((name, Dept),'database')*, and
$C6$ denotes *Contains ((title),'database')*.
Alternatively, the query can be expressed as:

$$(\Pi_L(\sigma_{C1}(J) \cup \Pi_L(\sigma_{C2}(J) \cup \Pi_L(\sigma_{C3}(J)) \cap (\Pi_L(\sigma_{C4}(J) \cup \Pi_L(\sigma_{C5}(J) \cup \Pi_L(\sigma_{C6}(J))$$

□

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

$$\Pi_L(\sigma_{(C1 \vee C2 \vee C3) \wedge (C4 \vee C5 \vee C6)}(J))$$

where L, J and the Ci are as defined earlier. We call the above optimization the *primary key* optimization.

6 Inversion of Other Relational Operations

We now consider the problem of computing the keyword-independent inversion of queries containing relational operations other than select, project and join. These operators include aggregation, intersection, union, set difference, semijoin, and antijoin (semijoin and antijoin are used to translate nested subqueries into relational algebra).

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.

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

In this section we only deal with safe queries, where the result of inversion is finite. However, there are many unsafe queries where the result can be represented in a finite manner by using a special value "*" representing the set of all possible values. In Section 7 we discuss extensions to handle some special cases of unsafe queries, which are important in practice.

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.

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

We now consider how to perform keyword-independent inversion for the intersection operation ($\cap$). Let $Q = Q1 \cap Q2$. In case $Q1$ and $Q2$ have identical parameters, $\mathit{INVQ}(Q)$ can be defined in a straightforward manner as

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

However, in general different parameters may be used in each of the inputs, leading to different parameter attributes being present in the inverted forms of $Q1$ and $Q2$, and a direct intersection is not possible. When the intersection is a set intersection (the default in SQL) an alternative is to use a natural join in place of intersection. Specifically, given a query $Q = Q1 \cap Q2$ where the attribute names of corresponding attributes of $Q1$ and $Q2$ are identical, the inverted query $\mathit{INVQ}(Q)$ 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. If the attribute names of $Q1$ and $Q2$ are not identical, they should be appropriately renamed.

Example 5 For example, given the query

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

the inverted query is

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

The above inversion may change the number of duplicates with a given parameter value, as compared to the original query running on a given parameter value. The duplicate count matters if the operation is part of an expression with an aggregation operation above it.

We can solve the above problem by defining a version of the multiset join operation whose duplicate semantics matches that of the intersection operation. This can be done by adding to each tuple an extra attribute recording the duplication count of that tuple, and replacing all duplicates by a single tuple with an appropriate count; this can be done easily by a aggregation operation. The count associated with a

tuple t in the result of a join operation $r \bowtie s$ would then be the minimum of the counts of the tuples t_r and t_s, if tuples t_r and t_s joined to give tuple t. All operations further up in the query tree would have to be modified to correspondingly take the count attribute into account. We omit details, since the approach of replacing duplicate tuples by using an extra count attribute, and defining multiset relational algebra operations based on the count attribute, is well known.

There is an alternative solution which has the following two steps

1. Add extra parameter attributes to each of *INVQ*$(Q1)$ and *INVQ*$(Q2)$, so they have the same schema. The value of the parameter attributes added thus is set to the value "*", which matches with every concrete value.
2. The query is then rewritten as
   INVQ$(Q1) \Cap$ *INVQ*$(Q2)$
   where the inverted queries have the extra attributes added as above, and where $\Cap$ denotes intersection taking the special semantics of the "*" value.

Such an operation is, however, not supported by any database we are aware of. It is worth noting that the result of the $\Cap$ operation will not have a "*" value in the above case, since the value of each parameter attribute will be non-"*" in at least one of the two inputs.

6.3 Union and Outer Union Operations

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 $Q = Q1 \cup Q2$. 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. In this case, where both subqueries have the same set of parameters, as long as the subqueries $Q1$ and $Q2$ are safe, so is Q.

However, if the subqueries have different sets of parameters, the problem is more complicated. Suppose that $Q = Q1 \cup Q2$ 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 attributes $B1$ and $B2$, while that of $Q2$ would contain $B2$ and $B3$.

The query in the above example is in fact unsafe even 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. (It is possible, however, that for specific pairs of keywords $K1, K2$, the query is not unsafe.)

If we know that the parameters that are present in only one of the inputs to the union come from a finite domain, we can rewrite each input by adding a cross product with the domain of those parameters that are missing from that input, as described in Section 4. For example, if parameter $\$P1$ is not present in $Q1$, we rewrite $Q1$ as $Q1 \times \sigma_{P=\$P1} D_P$, where D_P is a relation with a single attribute $P1$, containing all values that parameter $P1$ can take. However, such a rewriting would be rather inefficient if the domain is large.

A more efficient (and more general) approach is to use a special value "*" to represent the set of all possible parameter values. For example, if $Q1$ has parameter $\$P1$ and $Q2$ has parameter $\$P2$, if keyword $k1$ is present in the result of $Q1$ with $\$P1 = 4$, then the inverted result of $Q = Q1 \cup Q2$ with respect to $k1$ would contain a tuple $(4, *)$, indicating that $k1$ is present in the result of Q invoked with $\$P1$ set to 4, with $\$P2$ set to any possible value. We explore this approach in more detail in Section 7.

The outer-union operation is an extension of the union operation, that allows the operands to have different schema. The outer-union operation first pads the input tuples with extra attributes to bring them to a common schema (containing the union of all the attributes of the input relations); for each tuple, the values of the attributes added above are set to null. After bringing the inputs to a common schema, the outer-union operation performs a regular union operation. The outer-union operation is important in practice, since it permits us to model a form that contains multiple queries $Q1, \ldots, Qn$ as a form containing a single query defined as the outer-union of all the Qis.

Since the outer-union operation can be expressed in terms of projection and union, it can be inverted using the inversion techniques we have seen earlier (as well as those which we will see later in Section 7) for the union and projection operations.

6.4 Set Difference Operation

Given the expression $Q = Q1 - Q2$

1. If the parameters of $Q1$ and $Q2$ are identical, the inverted query is
   INVQ$(Q1) -$ *INVQ*$(Q2)$
2. More generally, as long as all parameters used in $Q2$ are also used in $Q1$, the inverted query is
   INVQ$(Q1) \overline{\ltimes}$ *INVQ*$(Q2)$
   where $\overline{\ltimes}$ denotes the natural anti-semijoin.
3. Finally, we consider the case where some of the parameters used in $Q2$ are not used in $Q1$. Let these parameters be $\$P1, \$P2, \ldots, \$Pk$. In this case, in general, there is a possibility of the query being unsafe, since there

may be an infinite number of values for these parameters for which the query result may contain a particular keyword; at the same time there may be some values for these parameters, for which the query result does not contain the keyword. Therefore, to ensure safety, we require that any such parameter must also be equated to an attribute of some other (finite) relation. We can then rewrite $Q1$ as follows, along the lines described in Section 4. Let us denote such a relation corresponding to parameter Pi as $BindRel(Pi)$, and let $BV(Pi)$ denote $\sigma_{Pi=\$Pi}(\Pi_{Pi}(BindRel(Pi)))$, and let BV denote $(BV(P1) \times BV(P2) \times \ldots \times BV(Pk))$ Then the inverted query is

$$(BV \times \mathit{INVQ}(Q1)) \overline{\ltimes} \mathit{INVQ}(Q2)$$

Inversion using the $\overline{\ltimes}$ does not, however, preserve the count of duplicates. If the count of duplicates needs to be preserved, we can use modified forms of the techniques described in Section 6.2 for intersection.

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 [8]; but as shown in Elhemali et al. [8], 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 proposed by [8] 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 for set difference in Section 6.4 to handle this case.

6.6 Outer Joins

We first consider the case of left outerjoin. Consider a query $Q = Q1 \rightleftarrows Q2$. If all the parameters used in Q are also present in $Q1$, the inverted form of Q is $\mathit{INVQ}(Q1) \rightleftarrows \mathit{INVQ}(Q2)$. (In case $Q = Q1 \rightleftarrows_\theta Q2$, the inverted query is $\mathit{INVQ}(Q1) \rightleftarrows_{\theta \wedge \gamma} \mathit{INVQ}(Q2)$, where predicate γ equates all parameters that appear in both $\mathit{INVQ}(Q1)$ and $\mathit{INVQ}(Q2)$.)

However, if some parameters used in Q are not present in $Q1$, Q can be unsafe. For example, suppose parameter $P1$ is used in $Q1$, and $P2$ in $Q2$. If the given keywords are present in $Q1$ for a particular value of $Q1$, then that $P1$ value combined with any arbitrary value for $P2$ (represented by a "*" value) would be an answer. We address the inversion of some cases of unsafe outer joins later, in Section 7.4.

As an alternative, if the domain of $P2$ is finite we can introduce parameter $P2$ into $Q1$ by rewriting $Q1$ to $Q1 \times BV(P2)$, where BV is as defined in Section 6.4, and then perform inversion. However, in case the domain of $P2$ is large, the above query can be inefficient.

The case of $\sqsubset\!\!\rightarrow$ is symmetric to the case of $\rightleftarrows$. The case of full outerjoin $\rightleftarrows\!\!\sqsubset$ can be handled similar to the case of $\rightleftarrows$ by the expression

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

provided that $Q1$ and $Q2$ have the same set of parameters.

7 Inversion of Unsafe Union Queries

In this section we focus on inversion of union operations, for the case where the inputs to the union are safe queries, but may have different parameters. Such union queries are used to model forms that output results from multiple queries, which are quite commonly used in practice. As discussed in Section 6.3, such union queries are unsafe, even though the subqueries are safe. Thus, in this section we address inversion of a special case of unsafe queries.

In the rest of this section we assume that the given query Q is a union of queries $Q1 \cup \ldots \cup Qn$, where each Qi is a safe query (i.e., safe with respect to the parameters used in Qi). We assume that the union operation is at the top level of the query (in other words, we assume that the result of keyword-independent inversion of Q is not used as the input for inverting a higher-level operation).

The general case of handling unsafe queries is beyond the scope of this paper, although in Section 7.4 we briefly discuss how to handle certain unsafe outerjoin operations, and to remove the restriction of unsafe operations occurring only at the top of the query.

7.1 Motivation and Intuition Behind Our Approaches

Recall from Section 6.3 that if the inputs Qi to a union operator all have exactly the same parameters, and are individually safe, the union query Q is safe, but if the Qi have different parameters, the union query Q is unsafe. Forms often

have multiple queries, with different queries having different parameters; for example, a form that takes a student ID and a year/semester may have two queries, with the first taking the student ID as parameter and fetching and displaying the students name, and the second taking student ID, year and semester as parameters, and fetching and displaying the students academic performance in that year/semester. (The different queries may have different output columns, but as discussed in Section 6.3, the union operation in this case is really an outer-union operation, which adds extra columns to each of the inputs to bring them to the same schema, and sets the values for these added columns to null.)

Thus, this case of unsafe queries does occur commonly in practice, and must be handled. To handle such unsafe queries, we use a special value "*" to represent the set of all possible parameter values, allowing an infinite number of parameter bindings to be represented efficiently in a finite manner.

For example, if $Q1$ has parameter $\$P1$ and $Q2$ has parameter $\$P2$, if keyword $k1$ is present in the result of $Q1$ with $\$P1 = 4$, then the inverted result of $Q = Q1 \cup Q2$ with respect to $k1$ would contain a tuple $(4, *)$, indicating that $k1$ is present in the result of Q invoked with $\$P1$ set to 4, with $\$P2$ set to any possible value.

More formally, saying that $(4, *)$ is an inverted query result is equivalent to saying $\forall X \in$ domain $\$P2$, $(4, X)$ is an inverted query result. Note that we use the symbol "*" for notational convenience; an implementation could use the SQL null value instead of "*". We also assume for simplicity that the literal value "*" does not appear in inverted query results; if it could, our query rewriting mechanisms can be modified to introduce suitable escape characters.

Keyword-independent inversion is performed for Q as follows:

- Each Qi is inverted individually; *INVQ*(Qi) has columns corresponding to each of the parameters used in Qi, but not for parameters used in other subqueries Qj.
- If parameter $\$B_k$ is not present in subquery Qi, we add a column corresponding to B_k to *INVQ*(Qi), with the special value "*". Thus, the schema of all inverted subqueries of a union becomes the same.
- *INVQ*(Q) is then simply the union of the *INVQ*(Qi) modified as above.

Keyword-specific inversion of the resulting *INVQ*(Q) is straightforward for the case of single-keyword queries, except that the result may contain the special value "*" for one or more parameters.

However, a problem arises when we need to process a multi-keyword query, since the intersection of the results for each keyword, whether implemented using intersection or join, 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'). Similarly intersecting (*, 'B') and (*,'B') would result in (*,'B').

Unfortunately, database systems do not support the special "don't care" value "*" when performing intersection or join (for union, the value "*" can be treated as a normal value). Intersection or join taking "*" values into account can be done in application code, but efficiency would still be an issue, since standard techniques for intersection, such as sorting, cannot be applied in a straightforward manner in the presence of the "*" value.

Another closely related problem is that if the query has some other operations such as a join, above the union operation, in the inverted query those operations too must deal with the "*" value. Although it is possible to define extended versions of all relational operators taking the semantics of the "*" value into account, such an extension is beyond the scope of this paper. To avoid this problem, as stated earlier, we assume that such a union operation is the top-level operation of the query.

We present two alternative solutions to handle the keyword-specific inversion of such queries, for the case of multiple keywords; both techniques allow the inverted query to be processed efficiently, entirely in the database.

1. The first solution, which we call the *Keyword-at-a-time (KAT)* implementation, is described in Section 7.2. This approach first computes, for each keyword Ki, the set of parameters that result in the keyword being present in the result of at least one of the Qj. The approach then combines the results across the keywords to get the final answer, by using intersection of the parameter values. The approach is complicated by the presence of the "*" value in the result of single keyword inversion.
   The basic idea behind handling intersection in the presence of the "*" values 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. Details are given in Section 7.2.
2. The second solution, which we call the *Query-at-a-time (QAT)* approach, is described in Section 7.3. The QAT approach solves each query with all the keywords, but allows 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 bindings that contain all the keywords. 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.

The QAT approach also has to deal with the presence of "*" values, and details are presented in Section 7.3.

Later, in Section 10 we present an experimental comparison of the two alternatives.

7.2 Keyword at a Time (KAT) Implementation

As mentioned in Section 6.3, the *keyword-at-a-time* (KAT) approach is one of the two approaches we propose for handling queries with a union, where different inputs Qj to the union may use different subsets of the query parameters. Intuitively, the KAT approach first computes the inversion result for each keyword Ki, that is, the set of parameters that result in the keyword being present in the result of at least one of the Qj. The approach then combines the inversion results across the different keywords by intersecting them to get the final answer.

The approach is complicated by the fact that different queries may have different parameters; to represent the fact that a particular query Qj does not use a particular parameter Pi, we invert query Qj as usual, but add an extra attribute named Pi to *INVQ*(Qj, Ki), with the special value "*" to denote that all values for that parameter are valid results. We call the above query as $R_{i,j}$. The normal intersection operator cannot be used in this situation.

We now outline how we can modify the approach to use standard database joins, without requiring special support for the "*" value. This can be done in two ways, one which we call the KAT with patterns (KATP) approach, and one which we call the KAT with is-null (KATIN) approach.

7.2.1 KAT using Patterns (KATP)

We now describe the *KAT using patterns* (KATP) approach. There are three variants of the KATP approach: KATP with No Materialization (KATP-NM), KATP with Initial Materialization (KATP-IM) and KATP with Full Materialization (KATP-FM). Except where otherwise specified, the techniques described below are the same for all three variants.

The following are the key steps in the KATP approach.

1. As mentioned earlier, the KATP approach inverts each query in the union with respect to each keyword, and then adds extra attributes corresponding to form-parameters that are not used in the query, with the value set to "*". Let the inverted query corresponding to keyword Ki and query Qj, with extra attributes added as above be $R_{i,j}$. We then define R_i as $\bigcup_j \{R_{i,j}\}$.
   In the two materialized variants, KATP-IM and KATP-FM, each of these relations $R_{i,j}$ is materialized as a temporary table, while in KATP-NM, each $R_{i,j}$ is defined as a non-materialized view; each R_i is defined as a non-materialized view, in both variants.
2. The next step computes the logical intersection of the R_i's, using a series of join steps instead of intersections. The result relation *result* is first set to R_1, and step i computes the logical intersection of *result* with R_{i+1}.
   The logical intersection of two relations r and s that contain tuples with the "*" value can be computed using the *pattern approach*, as follows:
   (a) 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_j.
   (b) Each r_i is then logically intersected with each s_j, by computing $\Pi_L(r_i \bowtie_P s_j)$, where P equates every column of r with the column of s for the same parameter, provided that attribute does not have the value * in both r_i and s_j (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 (forced to be equal by the join condition).
   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.
   (c) We create separate queries for each group in each intermediate *result*; the queries for the final state of *result* are combined by a union to get the final result.

In the KATP with Full Materialization (KATP-FM) variant, the intermediate result *result* at each step (other than the very first step, where we use R_1 directly, and the last step, whose result is output to the user) is materialized for use in the next step. The result is partitioned based on the "*" pattern, and each partition is joined separately, using the pattern approach described above.

In the other two variants (KATP-NM and KATP-IM), instead of materializing *result* at each iteration, we create a new result query, using the result query of the previous iteration, as follows. The result query at the beginning of an iteration is, in general, a union of multiple queries, with possibly different patterns. To perform the logical intersection, we group all queries having the same result pattern (i.e., "*"'s in the same positions), and define their union as *result*$_j$. In iteration i we compute the join of R_i with *result*$_j$ for each j.

The query generated in the final iteration is executed to get the required result.

7.2.2 KAT using Is-Null (KATIN)

The joins required to compute the logical intersection can be computed in another way, using the *is-null approach* which we describe below. We call this method the *KAT using Is-Null* (*KATIN*) approach. Here, we use null values to represent the "*" value, and do not partition the *result* relation. Unlike with KATP, we do not add extra attributes to the inverted queries $R_{i,j}$, and thus $R_{i,j}$ contains as attributes only the subset of parameters that Q_j uses. The relation R_i is now defined as the outer union of $R_{i,j}$ across all the j values. The logical intersection of *result* with R_i is done by a join followed by a projection, with the join condition and projection list defined as follows:

1. Let $P1, \ldots, Pk$ be the set of all query parameters. Then, $\theta = C1 \wedge \ldots \wedge Ck$, where Cj is defined as

   $\textit{result}.Pj = R_i.Pj$

   if Pj is present in all the queries (and therefore will not take the "*" value, represented here by null); otherwise Cj is defined as

   $(\textit{result}.Pj = R_i.Pj \vee \textit{result}.Pj$ **is null** $\vee R_i.Pj$ **is null** $)$

   Intuitively, the predicate ensures that either the corresponding parameters have the same value, or at least one of them represents the "*" value.
2. The ith attribute in the projection list is defined as *coalesce(result.*Pj*,* $R_i.Pj$*)*: the *coalesce* operation picks the non-null value if either one of the two inputs is null, and picks the first value otherwise (in this case both values would be the same).

As we did for KATP, we define three variants of the KATIN approach: without materialization (KATIN-NM), with initial materialization (KATIN-IM), and with full materialization (KATIN-FM). These are identical to the corresponding KATP variants in terms of what intermediate results they materialize.

7.3 Query at a Time (QAT) Implementation

We now give details of the *query-at-a-time* (QAT) approach. This approach first computes those parameter bindings for each query that contain at least one of the given keywords, along with a bitmap indicating which keywords are present for a given parameter binding. Then, the bitmaps from different queries for each relevant parameter binding are combined, to find the final answers.

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.
   In the QAT variants with initial materialization (IM) and with full materialization (FM), the result R_i is materialized, and the materialized result is used in subsequent steps. In contrast, in the no materialization (NM) variant, the query defining R_i is used in further processing.
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_i's 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.
   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 values set to null.
   Thus, to combine the results across all queries R_i we run the following pseudocode.

   result = R_1
   for $i = 2, \ldots, n$ {
   result = *result* $\Cup R_i \Cup$(*result* $\bowtie_{\theta_i} R_i$)
   /* See below for details on handling bitmaps */
   }

   The condition θ_i should equate parameters that occur in both *result* and R_i. However, computing (*result* $\bowtie_{\theta_i} R_i$) is not trivial, since both *result* and R_i may contain null values representing the "*" value. We discuss how to perform the join later in this section.

Note that each R_i has a bitmap b_i. We have a single bitmap attribute b in *result*. In the above union, in the term R_i, we rename b_i to b. In the term (*result* $\bowtie_{\theta_i} R_i$), we generate the value of attribute b as the bit-wise OR of the attribute b of *result* and attribute b_i of R_i. In some cases described later, where we use an outerjoin instead of a join, one of the b_i's may be null; such a bitmap is treated as equivalent to the bitmap with all zeros.

3. The final step is to select only tuples from *result* for which the bitmap b has all bits set to 1; these are the parameter values for which the result contains all the given keywords.

We now return to the issue of how to join *result* with R_i. We note that *result* contains tuples with different "*" patterns, with "*" represented as a null value due to the outer union operator $\Cup$. There are two ways of performing the join.

1. One option for computing the join is to partition *result* based on the pattern of null values, and use different predicates for the different partitions. This option, which we call QAT using Patterns (QATP), is discussed in Section 7.3.1.
2. Another option to enforce the semantics that "*" matches all possible values, is to use a join condition that explicitly checks for null values; this option, which we call QAT with Is-Null (QATIN), is discussed in Section 7.3.2.

The QAT algorithm can be optimized using the following two (related) optimizations:

1. 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* $\supset\!\bowtie\!\subset R_i$, 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.
2. For a keyword query having only one keyword, there is no need to include the joins of the inverted subqueries; in other words, we can drop the term (*result* $\bowtie_{\theta_i} R_i$). It suffices to use *result* $\Cup R_i$, since we do not have to worry about situations where one of the queries contains one keyword, and the other contains the other keyword.
3. We need to include a tuple from (*result* $\bowtie_{\theta_i} R_i$) in the result only if it contains a strict superset of the keywords that the constituent tuples from *result* and R_i contained. For example, suppose for a parameter binding $(v1, *)$ *result* contained the keywords $K1$ and $K2$, while with binding $(v1, v2)$ R_i contained only $K2$; then there is no need to add a result tuple with binding $(v1, v2)$ with keywords $K1$ and $K2$ since there is a more general result tuple with binding $(v1, *)$ containing the same keywords. In other words, the binding that is generated would be subsumed by an existing equal or more general binding generating the same keywords.
   Subsumed results of the above form can be eliminated by adding a join condition which checks that the bit-wise OR of the two bitmaps is a strict superset of the individual bitmaps. This optimization can reduce the number of results generated significantly.

It is worth noting that using a full outerjoin of *result* with R_i appears to be an alternative to using *result* $\Cup R_i \Cup$ (*result* $\bowtie_{\theta_i} R_i$). While this approach works in some special cases described later, in general it may lead to loss of information. For example 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 $(R1 \supset\!\bowtie\!\subset R2) \supset\!\bowtie\!\subset R3$ would not contain the correct answer (a1, b1, c2). Our solution of using the outer union avoids this problem.

7.3.1 QAT using Patterns (QATP)

In the *QAT using Patterns* (QATP) approach, we compute the joins using the patterns approach, instead of the is-null approach. The basic idea of joins using patterns is the same as in KATP, although some implementation details vary. For example, in the QAT version, the "*" value is represented by the null value.

The implementation details of QATP vary depending on the version we use.

1. In the QATP version with no materialization (QATP-NM), none of the queries (including the R_i) are materialized, and as in KATP with no materialization (KATP-NM), at each iteration *result* is a query defined as the union of queries, each with a different pattern.
2. The QATP version with initial materialization (QATP-IM) differs only in that the inverted queries R_i are materialized initially, and the stored results are used in subsequent queries. Since R_i occurs multiple times in the subsequent queries, materialization avoids the overhead of recomputation.
3. In the QATP version with full materialization (QATP-FM), in addition to materializing the R_i, in each iteration *result* is materialized (but with the tuples partitioned based on the "*" pattern). As in KATP, we do not materialize the initial value of *result*, since it is the same as the already materialized R_1, and the final value of *result* since it is consumed immediately to find the final results, and is not reused subsequently.

7.3.2 QAT using Is-Null (QATIN)

We now describe the *QAT using Is-Null* (QATIN) option for defining the join condition; the basic procedure of combining results across queries is as described earlier in Section 7.3. 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 the 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.

The join condition containing the **is null** disjunctions can result in poor execution plans, so we make use of the following optimizations:

1. 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.
2. For the join of *result* and R_2, we can drop the **is null** conditions, since none of the attributes can be null at this step.

In the QATIN variant with full materialization (QATIN-FM), the *result* relation at each iteration is materialized, and used in subsequent iterations. In the variants with no materialization (QATIN-NM), and initial materialization (QATIN-IM), each iteration defines a query using the query from the previous iteration. The query generated in the final iteration is executed to get the required result.

7.4 Extensions to Handle Other Unsafe Operations

Consider a query $Q = Q1 \rightleftarrows Q2$. If some parameters used in Q are not present in $Q1$, Q can be unsafe. For example, suppose parameter $P1$ is used in $Q1$, and $P2$ in $Q2$. If the given keywords are present in $Q1$ for a particular value of $P1$, then that $P1$ value combined with any arbitrary value for $P2$ (represented by a "*" value) would be an answer.

To handle the above problem, the keyword-independent inversion can be defined as

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

where $\Cup$ denotes the outer union operation of SQL, which brings all inputs to a common schema by adding required attributes, with their value set to null. Here, null values for parameters represent the "*" value, which denotes the set of all possible values. In effect, outerjoin has been transformed into union, and inversion performed on the union query.

The case of right outerjoin is symmetric with left outerjoin, while keyword-independent inversion of full outerjoin can be defined as

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

If the unsafe outerjoin is the top-level operation of the query, only keyword-specific inversion for the multiple-keyword case needs to handle the semantics of the "*" value. The different variants of the KAT and the QAT techniques which we saw earlier can be used to perform keyword-specific inversion.

However, if there are other operations above the unsafe union or outerjoin, keyword-independent inversion of those operations needs take the semantics of the "*" value into account. Although such an extension is possible for many operations such as select, project, join, union and intersection, details are beyond the scope of this paper.

8 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 all result parameter bindings together, to avoid mixing up results corresponding to different forms. Thus, the ranking problem is broken up into two problems: ranking forms, and ranking parameter values within each form.

8.1 Ranking Techniques with Single Queries

We first consider the case of forms with a single query, and later consider forms with multiple queries. We experimented with two variants of form ranking.

1. The first variant, which we call AVG, 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.
2. The second variant, which we call AVGMULT, multiples the average form result length with the number of different parameter values returned as answers to the given query. This helps lower the ranking of forms for which the keyword query result contains a large number of different parameter values.

In the case of AVG, in cases where the higher ranked forms provide sufficient answers, inverted queries may not even need to be executed for lower ranked forms. For AVGMULT, we cannot use this optimization.

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.

8.2 Ranking Techniques with Multiple Queries

For forms 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. 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 would like to give higher importance to the occurrence of a keyword in the first query than in the second, when ranking the form.

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 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.

8.3 Other Ranking Issues

We have assumed the AND semantics for keywords, but our techniques can be modified to support a fuzzy AND, allowing some keywords to be omitted, assigning a lower score to partially matching results. The QAT technique can be easily modified to implement such a scoring scheme.

For ranking of parameter values within a single form, we found that application specific heuristics seem to be quite effective. For example, if the parameter is a year or semester, the current year/semester is given higher preference, if the parameter is a user identifier, the identifier of the current user 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.

8.4 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 containing one or more *'s by a set of answers with the *'s replaced by the above values.

9 Implementation Details, Optimizations and Extensions

In this section we describe implementation details, such as how we handle SQL queries and certain form constructs. We also outline some optimizations which we have implemented.

9.1 Handling SQL Queries

Although our description of query inversion is in terms of relational algebra operations, any practical implementation has to support inversion of SQL queries. SQL queries which can be translated directly into relational algebra using the select, project, join/outerjoin, aggregation and set operations can be handled using the techniques for relational algebra inversion. Inverted relational algebra queries can be translated back to SQL for execution. SQL queries, including those with subqueries, can be translated into relational algebra queries by using techniques such as those described by Elhemali et al. [8]. In essence, the techniques of Elhemali perform query decorrelation. We note that the translation of some complex SQL queries that cannot be decorrelated requires the use of an extended relational algebra operator called the "Apply" operator, which is described in [8]; we do not currently handle the Apply operator.

Our actual implementation works directly on SQL queries, without going through a relational algebra translation; however, the techniques underlying our implementation are exactly the same as those we have described for relational algebra.

9.2 Handling Other Form Constructs

In this section we consider extensions to handle a larger class of forms with inter-related queries and static text.

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 Section 7 we considered how to handle forms with multiple queries, which can be modeled by using an outer-union of the queries; we had (implicitly) assumed that the queries are independent, that is they can be evaluated independently using the form parameter values.

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, 9].

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.

Static text is handled as follows: 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 the special value * to denote that all possible values for a corresponding parameter are answers.

Example 6 For example, suppose we have form F_1 with static text {*"Professor", "Course", "Teach"*} and form F_2 with static text {*"Student", "Course", "Take"*}. If the query keywords are {*"Professor", "Silberschatz", "database"*}, then the inverted query for F_1 should only include {*"Silberschatz", "database"*}, while that for F_2 should include the keywords {*"Professor", "Silberschatz", "database"*}. □

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 the 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.

9.3 Pruning

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 the condition *Contains(*$(Ri.A1, \ldots, Ri.An), Kj$*)*, 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. In our experiments, the pruning optimization was turned on by default.

9.4 Text Indexing Details

Both SQL Server and PostgreSQL allow a text index to be created on multiple attributes. In our implementation, we created a single index per relation, on all attributes of the relation. If a *Contains* predicate specifies only some of the attributes involved in the index, the result of the text index lookup has to be filtered to remove cases where the keyword occurs in an attribute other than the ones specified in the *Contains* predicate. Such filtering is done implicitly in SQL Server, whereas in PostgreSQL the query needs to specify the index to be used, as well as the extra predicates for filtering as above. 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.

Attributes with non-text types, such as integer or date, can be included in a full-text index by casting to text type in PostgreSQL. SQL Server does not support text indexing

of non-text types (whether directly or using casting), so instead for each such non-text column we add a (persisted) computed column of text/varchar type containing the textual representation of the non-text value. The text index is then built on text columns, including the computed columns generated above. As an alternative to adding such attributes to the full-text index, they can be handled by adding separate predicates in the inverted query, but doing so would increase the query overheads.

Another issue with SQL Server text indices was the requirement that a table on which a text index is defined must contain a single column unique key. To satisfy this requirement, for tables that had multi-column primary keys we had to add a new column defined as an identity type; SQL Server automatically generates unique values for such columns.

9.5 Materializing Form Results

The approach of [7], which materializes and indexes form results for each possible parameter value, is an alternative to our approach. While this approach may be faster than ours for answering queries, it would have a significant time overhead for maintaining the indices. The issue of incremental index maintenance for the materialized form results is not addressed by [7]. However, as mentioned in Section 5.1, we can use the keyword-independent inverted query $INVQ(Q)$ to find parameter values for which the materialized form result is affected, given an update to the database.

We did not measure the cost of processing a keyword query using the approach of [7]; it is reasonable to assume that the approach of [7] would be faster than our approach. Instead we focused on the cost of maintenance of the index on form results, when the database is updated.

To estimate the view maintenance overheads of using this approach, we materialized the keyword-independent inversion of each form query. Standard techniques for view maintenance such as those described in Silberschatz et al. [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.

There are at least two ways to build a full-text index on the result. The first way is to create and materialize a view *FormIndex*(*formid, parameters, allTupleAttrs*), with one tuple per parameter binding for each form. 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 attribute *allTupleAttrs* contains the concatenation of all attribute values from all tuples in the result of that formid with that parameter binding. A full-text index can be built on the resultant materialized view.

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. However, the underlying inverted queries still need to be materialized for incremental view maintenance.

The implementation we used for our performance testing used a variant of the first approach, but omitted the combination across forms as well as across tuples in a given form result. Thus, the textindex was built directly on the inverted query results. This approach underestimates the cost of maintenance since the combined index would require more effort to maintain, and thus our overhead measurements are actually conservative.

9.6 Extensions to Handle 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. Our implementation supports such replacement of parameters by session parameters before query inversion.

10 Experimental Results

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

10.1 Experimental Setup

The code for our system is written in Java, and currently works on PostgreSQL and SQL Server databases. Our performance study is based on a real database application, used to handle all academic information at IIT Bombay, with about 1 GB of data, and 90 form interfaces. The application runs on the PostgreSQL database. Each form had a short description, which was treated as static text for the form.

For the bulk of our performance tests, we used PostgreSQL 9.1 as the database, on a machine with an Intel Core i5-2500K, 3.30 GHz processor, with 16 GB of RAM, running Ubuntu with a Linux 3.0.0-14-generic kernel. The application and the database ran on the same machine. We report numbers using a 1 TB 7200 rpm hard disk (Seagate ST31000524AS), as well as with a Intel SSDSA2M080 80

GB solid state disk (SSD) with an eSATA interface (which we refer to as flash disk here on).

We also ported our data and form queries to SQL Server, and present some results using SQL Server 2008 running on Windows 7, which itself ran as a virtual machine (with Ubuntu as the host OS) on the same hardware as above, with 2 GB of memory allocated to the virtual machine.

We used a set of 12 keyword queries to study the quality of ranking as well as performance. We cannot give all the actual keyword queries since the database we use contains confidential data which cannot be made public, but 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. The number of keywords in these queries ranged from 1 to 4.

To study scalability of execution time with number of keywords, we used another set of keyword queries based on an overall set of keywords K; for each number of keywords i, the keyword queries consisted of all size-i subsets of K, and we took the average time across all these keyword queries. We chose the keywords such that all the keyword queries had at least one answer. Thereby we avoided bias that could result from different choices of keywords for different i. We call the above set of keyword queries the *scalability queries*.

We present numbers for cold cache (CC) and warm cache (WC). Cold cache results were generated by forcing the database to drop all clean buffers, which we enforced by restarting PostgreSQL after clearing Linux file system buffers by using the command "echo 3 $>$ /proc/sys/vm/drop_caches". (The command "DBCC DROPCLEANBUFFERS" achieves the same effect in the context of SQL Server.) 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.

For both warm and cold cache, the numbers reported are the averages computed from 6 runs, with the lowest and highest numbers dropped before computing the average of the remaining numbers.

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. A single full-text index is built for each relation, covering all attributes of the relation.

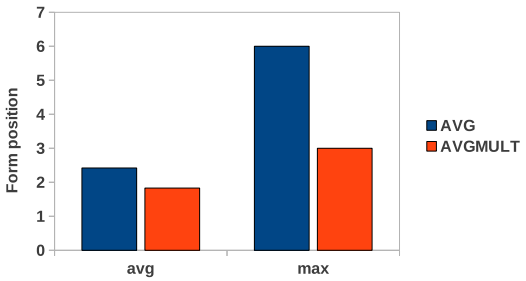

Fig. 1 Quality study on Academic database.

10.2 Effectiveness of Keyword Search on Forms

The first set of experiments studied the effectiveness of keyword querying in retrieving desired forms. We compared the following form ranking methods: (a) ordered (in ascending order) by average form result size (AVG), and (b) ordered (in ascending order) 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 keyword queries for that task, and found the position at which the desired form was present. Figure 1 shows that across all the keyword 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.

There are of course other ways of ranking form results, for example based on term frequency and inverse document frequency of keywords, and on (inverse of) document length. The AVG technique provides, in effect, an estimate of document length for (form, parameter) combinations, while AVGMULT provides a similar estimate for a form, summed up across all result parameter values. We have also found that when a form has multiple queries, it makes sense to give higher weightage to terms in the results of queries that occur earlier in the form. For example, in a form showing the academic records of a student, the first query is likely to retrieve the name of the student, while later queries may retrieve titles of courses taken by the student. Such a ranking is similar to the standard technique of giving higher weightage to terms that appear in the document title or early in the document text.

Exploration of alternative ranking techniques is certainly an important area of future work, although our results above show that the AVGMULT technique gives good results for the set of tasks we considered.

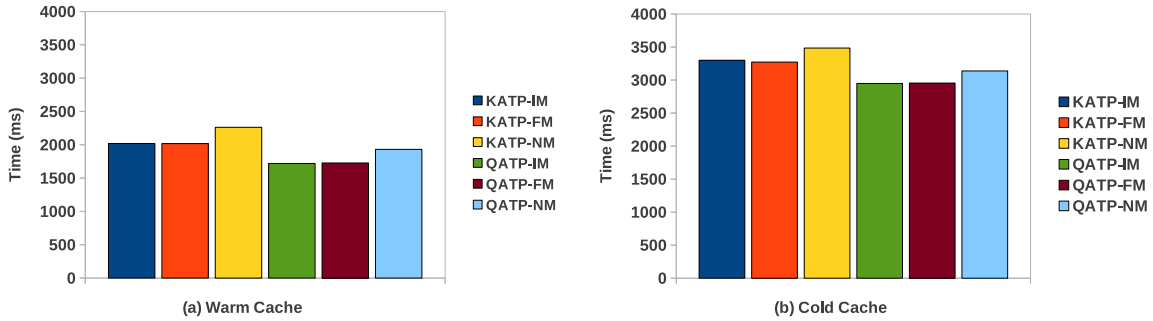

Fig. 2 Performance of KATP and QATP on PostgreSQL with Flash (a) Warm Cache and (b) Cold Cache

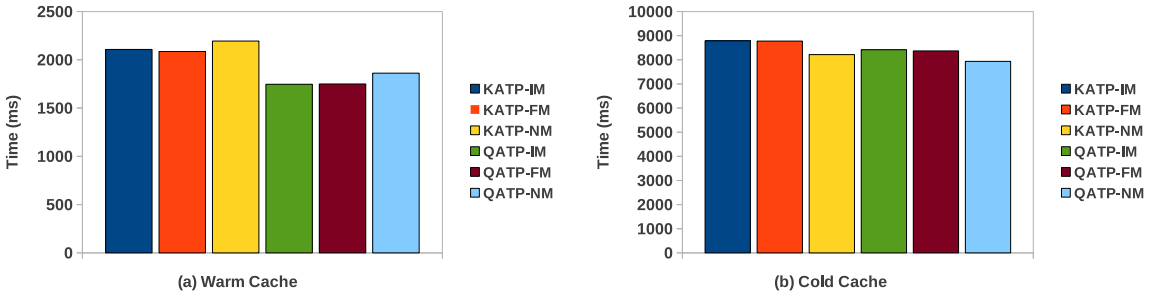

Fig. 3 Performance of KATP and QATP on PostgreSQL with Hard Disk (a) Warm cache and (b) Cold Cache.

10.3 Query Execution Time: Basic Results

Using the same set of 12 keyword queries, we measured the average execution time across all the keyword queries, using variants of the QAT using patterns (QATP) and KAT using patterns (KATP) methods on cold and warm cache. The variants we study are QATP with Initial Materialization (QATP-IM), QATP with Full materialization (QATP-FM), QATP with No Materialization (QATP-NM), KATP with Initial Materialization (KATP-IM), KATP with Full Materialization (KATP-FM), and KATP with No Materialization (KATP-NM).

The results with all 90 forms from the academic database, running on PostgreSQL, using flash disk, are shown in Figure 2 (a) and (b), for warm cache and cold cache respectively. The first point to note is that overall performance on flash disk, with average execution time under 2 seconds, is clearly good enough for interactive use; cold cache performance is at least 50% slower than warm cache performance, but the average execution time of under 3 seconds with QATP-IM and QATP-FM is still quite acceptable.

The results in Figure 2 show that initial materialization (-IM), i.e. the materialization of the initial inverted queries reduces the execution time compared to no-materialization (-NM) for all KATP and QATP variants, for both warm and cold cache. In both cases the initial inverted queries are used multiple times, making their materialization worthwhile. Full materialization (-FM) performed similar to initial materialization, indicating that materialization of intermediate results did not have a significant impact in these experiments. This was because the -IM and -FM variants are identical for forms with only two queries (since there are no intermediate results for -FM to materialize), and we had only a few forms with more than two queries in this set of forms.

Figure 2 also shows that the variants of QATP perform slightly better than all the variants of KATP. The difference is quite small since most of the forms had only one single query using only selection, projection, join and aggregation, and for such forms all the variants of KATP and QATP perform exactly the same actions. To highlight the performance differences between the variants better, in Section 10.6 we present results using a smaller set of forms with multiple queries, and discuss the relative performance of the different KAT and QAT versions using those results.

The results using hard disk, running on PostgreSQL, are shown in Figure 3 (a) and (b) for warm cache and cold cache respectively. For warm cache, there is hardly any difference in the flash and hard disk timings, since data is memory resident. However, with cold cache all the variants took about 7 to 8 seconds on average, which is significantly more than the time taken with warm cache, and with cold cache on flash. We believe the reason is that query execution plans for the inverted queries usually involve keyword index lookup as well as indexed nested loops joins, both of which require a good deal of random IO if the cache is cold; and random IO on flash is much faster than random IO on hard disk.

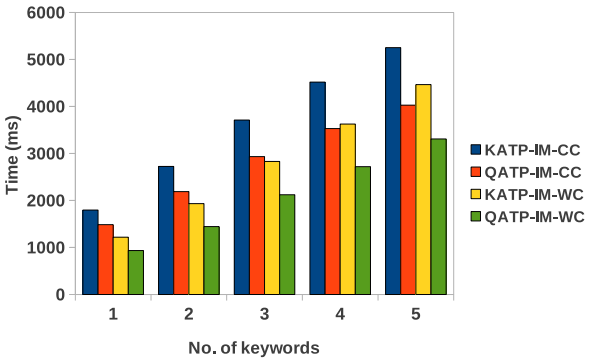

Fig. 4 Scalability of KATP and QATP with number of keywords (with Flash).

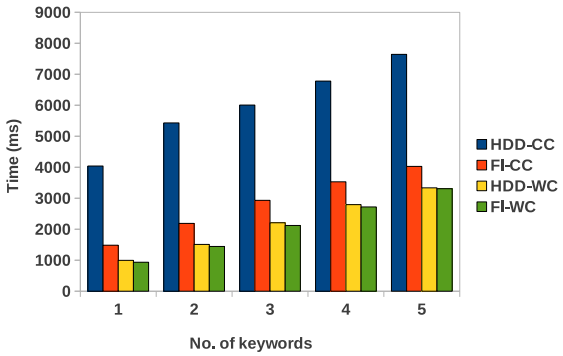

Fig. 5 Scalability with number of keywords on Hard Disk and Flash (with QATP-IM).

Again, the QAT variants are slightly better than the KAT variants. Interestingly, Figure 3 shows that for hard disk, cold cache timing for non-materialization variants (QATP-NM and KATP-NM) are better than the materialized variants, which is the opposite of the case with hard disk warm cache, and with flash warm and cold cache. This result was surprising, and on investigation we found that the reason behind this was that just creating a materialized table on hard disk for cold cache in PostgreSQL was around 4 times more expensive than for warm cache and around 3-4 times more expensive than flash (for cold and warm cache respectively). We believe that for a production system, where the cache will be warm at least for metadata, the materialized variants are likely to outperform the non-materialized variants.

We also performed the above experiments using SQL Server to execute the inverted queries. The results are very similar to those for PostgreSQL on average, although results did vary for individual queries with SQL Server taking less time than PostgreSQL for some queries, and more time for others.

Overall, the results show that keyword search runs with performance good enough for interactive use with flash disk, even with cold cache, although performance is not quite as good on hard disk with cold cache. Given current hardware trends it is quite reasonable to assume that enterprise application data will fit on flash disk for all but the very largest enterprises; as a result, we believe search performance will be quite acceptable in such settings.

10.4 Scalability

The next sets of experiments studies the scalability (in terms of query execution time) of our approach with respect to the number of keywords, and to the number of forms.

The first set of experiments addresses the issue of scalability with number of keywords. For these experiments we used the scalability queries described in Section 10.1, with the number of keywords varying from 1 to 5. This set of queries is based on a set of 5 keywords, and for each i, its size i-subsets together form the set of queries using i keywords. We report the average time for each value of i. We used the QATP-IM and KATP-IM methods for these experiments. The results for the case of forms on the academic database, running on PostgreSQL, using flash disk, are shown in Figure 4, with separate numbers for cold cache (CC) and warm cache (WC).

As can also be seen from the Figure 4, both KATP-IM and QATP-IM scale slightly sub-linearly with the number of keywords, for both cold and warm cache, and QATP-IM performs slightly better than KATP-IM. One reason for sub-linear performance could be that the pruning optimization described in Section 9.3 eliminates more forms as the number of keywords increases, as we will see shortly. Another reason could be that our system has some initialization costs related to reading and parsing form queries, which are independent of the number of keywords.

We also compared the performance on hard disk versus flash, using the QATP-IM method, with an increasing number of keywords, using the same scalability query set. The results are shown in Figure 5. Similar to the results we saw for the original set of 12 keyword queries, cold cache numbers on hard disk are relatively high, but warm cache and flash (both warm and cold cache) numbers are quite good. Again the time taken scales sublinearly with number of keywords, for the same reasons we saw earlier.

Next we studied scalability with an increasing number of forms. For this experiment we used only the first 80 out of the 90 forms. For a given number of forms n, we partitioned the overall set of 80 forms into partitions of size k, and took the average execution time across these partitions. We used the same set of 12 queries described earlier for quality of ranking experiments, and ran the experiments on flash disk.

Figure 6 shows how the time taken increases with number of forms. The results for cold cache (bar labeled CC) and warm cache (bar labeled 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 result-

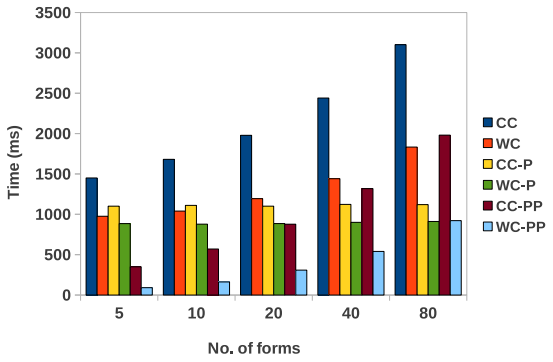

Fig. 6 Scalability with number of forms on Academic database (set of 12 queries, using QATP-IM).

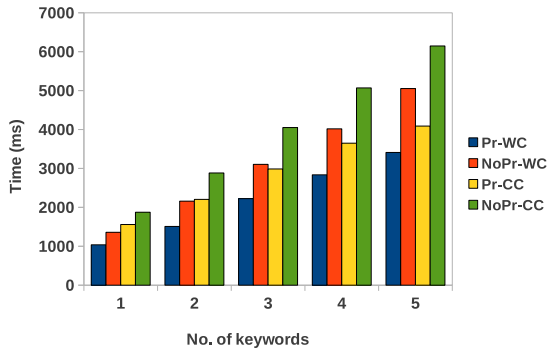

Fig. 7 Effect of pruning of forms on Academic database with varying number of keywords (scalability queries, using QATP-IM).

ing 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. To quantify this effect we measured the time taken for the pruning step, shown in bars CC-P and WC-P for the cold cache and warm cache cases; as can be seen from the figure these numbers do not increase with the number of forms. We also measured the post-pruning time, that is, the time taken after the pruning step, and show these times in bars CC-PP and WC-PP (for the cold cache and warm cache cases). It can now be seen that the growth is no longer as highly sublinear as earlier, with the post-pruning time increasing by 6-10 fold (for cold/warm cache resp.) when the number of forms went up 20 fold from 5 to 80 forms. Thus, the growth remains sub-linear even in this case, but less remarkably so.

Given that our database has a total size of 1 GB, while available main memory is significantly larger, warm cache numbers basically reflect completely in-memory query evaluation, while the cold cache numbers do not reflect the potential for repeated fetches of the same data. Thus, another scalability related issue is: "how will the techniques run if the database size is larger than memory?"

We could not create a larger dataset, nor could we actually decrease the memory capacity of the system we used. Instead, we kept the database size fixed and reduced the PostgreSQL buffer size. By default, PostgreSQL uses a very small buffer, leaving the job of buffering primarily to the OS file system cache, and most PostgreSQL buffer misses do not result in actual IO. Thus, the buffer misses do not get reflected in execution time changes. Therefore, instead of studying the execution time variation, we studied the variation in the number of buffer misses reported by PostgreSQL, as the buffer size is varied. In a system where the amount of real memory corresponds to the PostgreSQL buffer size (or where file system buffering is turned off) these buffer misses would corresponding to actual IO operations.

Specifically, we ran the inverted queries on PostgreSQL with shared buffer size set to 24 MB, 128 MB and 1228 MB respectively. We measured the number of buffer hits and misses (we verified that the total of hits and misses were the same across all buffer sizes). With cold cache, the number of buffer misses was the same at 128 MB and 1228 MB for all the queries. The number of misses was also the same for buffer sizes of 128 MB and 24 MB for 8 out of 12 of the keyword queries; the ones where there was a difference were those that used frequently occurring keywords. The worst case increase in buffer misses when going from 128 MB to 24 MB buffer size was by a factor of 3, while the average increase across all keyword queries was by a factor of 2. We further stressed our system by considering a new set of 9 keyword queries, each of which contained one or more frequent keywords; even then, the average increase was less than a factor of 5 when buffer size changed from 128 MB to 24 MB. Thus, we believe our techniques will work well even with database sizes that are significantly larger than memory.

10.5 Effect of Optimizations

Next, we studied the advantage of enabling the pruning optimization described in Section 9.3, using the academic database on PostgreSQL, using a flash disk, and the QATP-IM method, using the same set of keywords used earlier for testing scalability with number of keywords.

Figure 7 shows the time for the same keyword queries, with and without pruning, with different numbers of keywords. It is clear that pruning has a significant effect both on cold and on warm cache, reducing the execution time by 20 to 30 % compared to the no-pruning version, with the benefit roughly the same with the number of keywords ranging from 1 to 5. Figure 8 shows the average number of forms for which inverted queries had to be executed (i.e. the average number of forms that were not pruned), with different numbers of keywords. Although each form takes longer to process with increasing number of keywords, there are fewer forms to process.

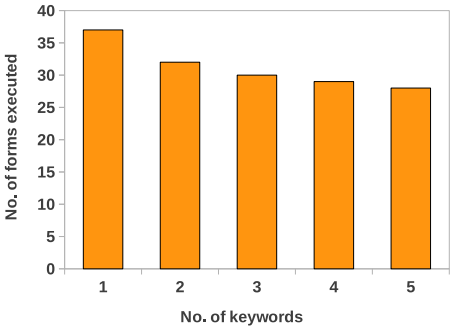

Fig. 8 Pruning of forms against number of keywords on Academic database (scalability queries).

We now consider the effect of subsumption checking, which we implemented in the QAT approach as described in Section 7.3. Subsumption can only happen in case of forms with more than one query, with different sets of parameters, when more than one keyword is specified. As an anecdotal example of the importance of subsumption, we considered one of the forms with two queries, with $Q1$ having parameters academic year, semester and department code, and $Q2$ having parameters academic year, semester, and course code. With the keywords "CS631" and "database", the inverted query of $Q1$ returned 250 results, the inverted query of $Q2$ returned 9 results; each of these results have "*" for one parameter. The logical intersection of the two inverted queries returned 679 results, but all of these were subsumed by the original set of 259 results. Thus, far fewer results had to be returned to the user. Checking subsumption is more complicated in the KAT approach since it cannot be done as part of the join condition, and has not been implemented currently.

10.6 Comparison of KAT and QAT Variants

The earlier experiments did not highlight the difference between the different KAT and QAT variants since most of the forms in the IITB academic application had just a single query. And out of the 22 forms with more than one query, 20 had the same parameters in all queries, and as a result QATP and QATIN are identical, and KATP and KATIN are similarly identical, on these forms. The remaining 2 forms had queries with different parameters, but they both had only two queries; with only two queries, again QATP and QATIN are identical.

In particular, there is a potential for poor performance of the KATIN and QATIN approaches in cases with more than 2 queries, where there is no parameter common across all the forms. In such cases there would be no join condition free of the **is null** disjunct, and as a result the plans could require nested loops joins.

To study the effect of the variants, we added 4 new forms each containing 3 queries with different parameters. We used the original set of 12 keyword queries, augmented with 7 more keyword queries for which the new forms would generate answers.

Figure 9 (a) and (b) show the performance of all the materialization variants (-NM, -IM and -FM) of the QATP, QATIN, KATP and KATIN approaches on cold cache for flash and hard disk respectively, on the 4 new forms. Figure 10 (a) and (b) show the same results as Figure 9 (a) and (b), but using SQL Server instead of PostgreSQL. We omit warm cache numbers since they are similar to the cold cache numbers with flash. Although the raw numbers are better with SQL Server than with PostgreSQL, our goal here is not to compare results across the two databases, but rather to compare the alternative techniques on both systems.

Figure 9 shows that with PostgreSQL the QATIN variants are about 2 times more expensive than the QATP variants with QATP-IM performing the best overall, and QATIN-NM performing the worst amongst the QAT alternatives. We also studied the best case and worst case ratios of QATP versus QATIN performance across individual keyword queries. We found that QATP variants were never worse than QATIN variants, while QATIN variants performed 2.5 to 3 times worse than QATP-IM on several queries. Figure 10 shows that the pattern is similar with SQL Server, although the differences are more marked with SQL Server on flash.

With both PostgreSQL and SQL Server, the KATIN variants were also somewhat more expensive than the corresponding KATP variants, with one exception: KATP-NM performed significantly worse than all other KAT variants. We believe this is because in KATP the initial inverted queries get repeated multiple times, and the no-materialization version evaluates the same query multiple times leading to poor performance. In contrast with QATP, the number of repetitions is less, so while QATP-NM performs worse than the other QATP variants, the difference is not as marked. Overall, KATP-IM is the best amongst the KAT variants. Comparing the performance ratios for individual queries, we found that KATP-IM was never worse than the best KATIN variants by more than a very small value, whereas in some cases all the KATIN variants performed 50% worse than KATP-IM.

Although not explicitly shown in our results, across all the forms (the original forms as well as the newly added forms), we found that performance was relatively slow with PostgreSQL for queries where some keyword was present in the results of some form for a very large number of different parameter values, resulting in very large results for some inverted queries. How to optimize such queries to avoid computing large intermediate results is an area of future work.

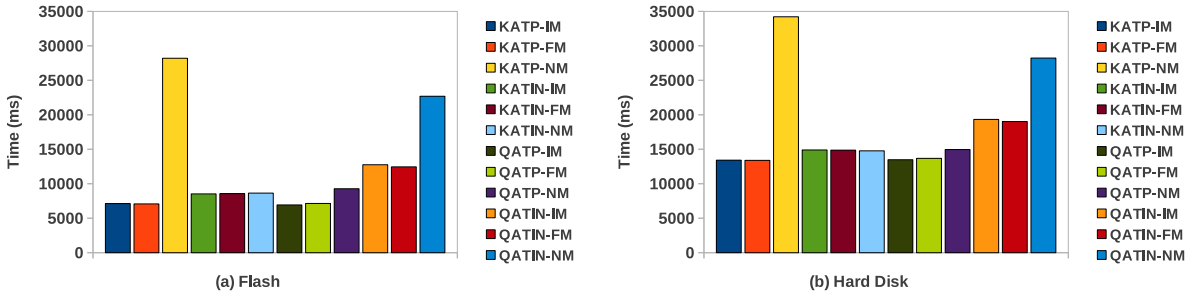

Fig. 9 Performance with newly added forms on PostgreSQL (a) on Flash and (b) with Hard Disk (with cold cache)

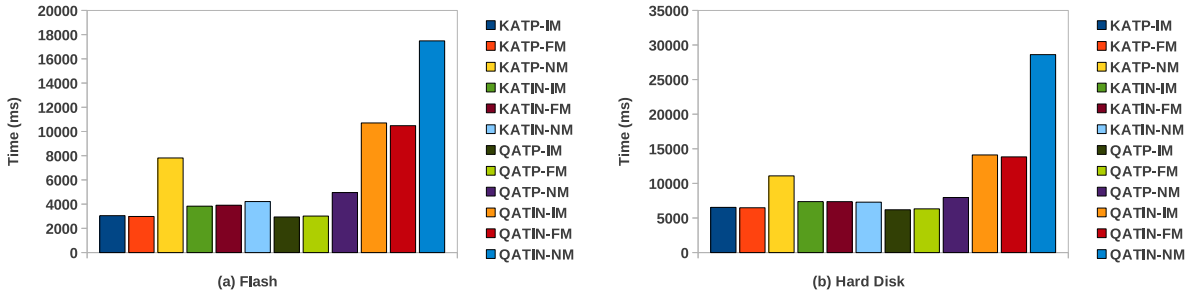

Fig. 10 Performance with newly added forms on SQL Server (a) on Flash and (b) with Hard Disk (with cold cache)

10.7 Comparison with Materializing and Indexing Form Results

The approach of Duda et al. [7], 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 Section 9.5, 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; we also created 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 2.7 seconds with a hard disk, and 500 milliseconds with a flash disk, for an update that normally takes a few tens of milliseconds; this is an unacceptable overhead for the academic application.

We note that the time is actually an underestimate of the actual cost, since (a) we did not create a merged index across forms, which would require extra effort to maintain, and (b) 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 occasionally. 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.

Thus, we believe our approach is better suited for production systems where keyword queries are likely to be less common than updates, insertions and deletions.

11 Conclusions and Future Work

The problem of keyword search on the results of form interfaces is of 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. Another important area of future work lies in dealing with keywords

that are present in a very large number of form results with different parameter bindings.

We also plan to extend our implementation to work with a larger class of SQL queries. In particular we need to generate rewritings that extend operators to work with multiple input partitions, each with a different set of parameters with the "*" value; such an extension would allow us to handle, for example, outerjoins which are not at the top level of the query.

We also need to handle complex application code with conditional execution of queries. A possible approach to handling such forms is to create a separate logical form for each possible execution path, with associated conditions under which each of the logical forms will be executed. We also plan to address the form ranking problem in more detail.

Acknowledgements We thank Surajit Chaudhuri for discussions leading to the idea of inverting form queries. This work was partially supported by the Indo-German Max Planck Center for Computer Science (IMPECS) project, which is supported by DST India, BMBF Germany and MPG, Max Planck Society.

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. 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.
6. 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.
7. C. Duda, D. A. Graf, and D. Kossmann. Predicate-based indexing of enterprise Web applications. *CIDR*, pages 102–107, 2007.
8. M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. Joshi. Execution strategies for SQL subqueries. In *SIGMOD Conference*, pages 993–1004, 2007.
9. R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *PVLDB*, 1(1):1107–1123, 2008.
10. V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. *VLDB*, pages 670–681, 2002.
11. F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. *SIGMOD Conference*, pages 563–574, 2006.
12. Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. *SIGMOD Conference*, pages 115–126, 2007.
13. A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR*, 2009.
14. F. Shao, L. Guo, C. Botev, A. Bhaskar, M. Chettiar, F. Yang, and J. Shanmugasundaram. Efficient keyword search over virtual XML views. *VLDB Journal*, 18(2):543–570, 2009.
15. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, sixth edition, 2010.