Toward Scalable Keyword Search over Relational Data

Akanksha Baid, Ian Rae, Jiexing Li, AnHai Doan, and Jeffrey Naughton
University of Wisconsin, Madison
{baid, ian, jxli, anhai, naughton}@cs.wisc.edu

ABSTRACT

Keyword search (KWS) over relational databases has recently received significant attention. Many solutions and many prototypes have been developed. This task requires addressing many issues, including robustness, accuracy, reliability, and privacy. An emerging issue, however, appears to be performance related: current KWS systems have unpredictable running times. In particular, for certain queries it takes too long to produce answers, and for others the system may even fail to return (e.g., after exhausting memory). In this paper we argue that as today's users have been "spoiled" by the performance of Internet search engines, KWS systems should return whatever answers they can produce quickly and then provide users with options for exploring any portion of the answer space not covered by these answers. Our basic idea is to produce answers that can be generated quickly as in today's KWS systems, then to show users query forms that characterize the unexplored portion of the answer space. Combining KWS systems with forms allows us to bypass the performance problems inherent to KWS without compromising query coverage. We provide a proof of concept for this proposed approach, and discuss the challenges encountered in building this hybrid system. Finally, we present experiments over real-world datasets to demonstrate the feasibility of the proposed solution.

1. INTRODUCTION

The success of search engines demonstrates that untrained users are comfortable using keyword search to find documents of interest to them. Over the past decade, this success has spawned tremendous interest in keyword search (KWS) over relational databases, in order to accommodate users who cannot issue a formal structured query or are unaware of the database schema. DBXplorer [2], DISCOVER [12], and BANKS [1] were among the first systems that supported keyword search over relational databases, and many other systems (e.g. [4, 10, 11, 21, 24, 26, 28, 29, 30]) have since been developed. As more structured data becomes available at organizations and on the Web, and as more untrained users want to use such data, we expect that more efforts will be devoted to building such systems in the near future.

Naturally, building such systems requires addressing many issues, including robustness, accuracy, reliability, and privacy. However, in our own effort, we have been struck by how quickly we were hit with the first roadblock: current KWS solutions have unpredictable performance issues. Specifically, while the systems produce answers quickly for many queries, for many others they take an unacceptably long time, or even fail to produce any answer after exhausting memory. Clearly, such a performance profile is unacceptable for a real-world system.

In this paper we investigate how to address this problem. We begin by examining the problem and argue that it is a fundamental problem unlikely to be solved in the near future by software or hardware advances. Intuitively, this is because current KWS solutions often require the solution of sub-problems that are NP-complete. Consequently, the time spent on these problems often grows very quickly.

Next, based on our experience with KWS systems, we argue that KWS systems should produce answers under an absolute time limit (say a few seconds), even if such answers are only partial in some sense. This is largely because (a) Web search engines have conditioned most users to expect near-instantaneous responses from KWS systems, an expectation likely to be carried over to the relational setting, and (b) KWS is by nature often an interactive enterprise, which requires responses at interactive speed. We then argue that when showing these (possibly partial) answers, the system should also somehow characterize the portion of the answer space that is as yet unexplored, so the user knows what it is that he or she is potentially missing. The system should then allow for a way for the user to explore this portion of the answer space should he or she choose to do so. We propose that one way to do so is to provide *form interfaces* to characterize the yet-unexplored answer space. We believe that the above two requirements of "time limit" and "overview of the yet-unseen" can help increase the chances that the KWS system will be perceived as useful and will be widely adopted by real-world customers.

We hope that our work will be viewed as an attempt to open the debate on what it takes to bypass the performance bottleneck of current KWS systems without compromising on query coverage. Toward this goal we make the following contributions:
• A discussion of performance issues in current KWS solutions
• A minimal-overhead fix, based on the ideas of "time limit" and "overview of the as-yet-unseen using query forms"
• A concrete realization of the fix for a candidate-network based KWS system and,
• Experiments with a real-world dataset to demonstrate the effectiveness and feasibility of the proposed approach when coupled with some representative KWS systems, namely, DISCOVER, BANKS-II, BLINKS and EASE.

The rest of this paper is structured as follows: we discuss the limitations of current keyword search solutions in Section 2. The basic idea behind our solution is presented in Section 3. Section 4 describes the the detailed implementation of our approach for a

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

candidate-network based KWS system. Section 5 contains an experimental evaluation of our approach over the DBLife and DBLP datasets. Related work is presented in Section 6, and we conclude in Section 7.

2. CURRENT KWS SOLUTIONS

Current KWS systems falls into two broad categories: candidate network based systems and graph based systems. In this section, we briefly describe both these solutions and then discuss their performance problems.

Candidate-network based solutions: DISCOVER [12] and DBXplorer [2] are examples of the first approach. In these systems text-indices over the data and the user's keywords are used to generate "candidate networks" (CNs) by using an approach similar to BFS, with the aid of the underlying schema. Intuitively, each CN encodes a way to join the relations in the database to produce answers to the user's keyword query. The CNs are then translated into SQL queries and the corresponding queries are executed to obtain result tuples. In some KWS systems, the answers of the executed queries may be pipelined and presented to the user, as more SQL queries are being generated and executed. The answers may be ordered using a predefined ranking function before being presented to the user. Note that some of the generated candidate networks may not lead to any result tuples.

Data/Graph based solutions: BANKS [1, 15], BLINKS [10] and DBPF [8] are examples of the second approach. These solutions are not schema-aware. Consequently, these approaches can be applied to any data that can be modeled as a graph. In the relational context, the nodes correspond to actual keywords and the edges correspond to key-foreign key relationships between the underlying data. Results of keyword queries are usually modeled as trees that connect nodes matching the keywords.

2.1 Performance problems

As we pointed out earlier, KWS systems suffer from a fundamental problem: they cannot guarantee a performance "cap" in the sense that certain user queries may take a very long to complete. This often happens as the number of joins or the fanout of the nodes increases.

Figure 1 shows four queries (of varying complexity) issued over a CN-based KWS system, for which the run time increases as the number of joins is increased on the X-axis. In fact, for our experimental setting, the queries "agrawal chaudhuri das" and "dewitt widom," exhaust system memory at 5 and 6 joins respectively. The other two queries,"keyword search sigmod" and "publications icde," exhaust memory at 7 and 8 joins respectively. All queries run for times between 200 and 400 seconds for long join sequences.

The fundamental reason for this drastic degradation in performance is that even though different approaches to keyword search use different techniques, at the core they all deal with the problem of searching a graph to find all sub-graphs that satisfy certain properties. It is well-known that variations of this problem (i.e., the minimum Steiner tree problem) are NP-hard [15]. For instance, the number of CNs is known to be exponential in the query size [11], unbounded in the presence of cycles in the schema graph, and bounded by the size of the data in the presence of many-to-many relationships in CN-based KWS systems. This makes query execution prohibitively expensive.

Consequently, as shown in Table 1, as we increase the number of allowed joins the run time of the CN generation step increases exponentially. For example, using the query "dewitt widom," as we increase the number of allowed joins from 2 to 4 to 6, the number of candidate sub-graphs for CNs that the KWS system must generate and consider (called *joining networks*) increases from 178 to 6,592 to 218,842. As this number grows, it soon overwhelms the amount of available memory, which causes the system to return with no

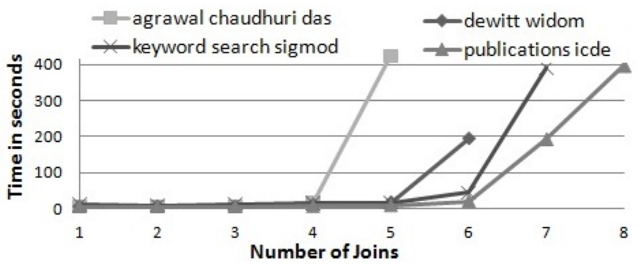


Figure 1: The performance of KWS degenerates as the number of joins grows for four-keyword queries, issued over a candidate network based KWS system.

answer. To reduce the run time, KWS systems often find only CNs of up to a certain maximum number of joins, thus compromising coverage.

Number of Joins	0	1	2	3	4	5	6
Number of JNs	2	14	178	1214	6592	37788	218842
Number of CNs	0	0	6	6	46	46	296

Table 1: The number of join networks (JNs) and candidate networks (CNs) for the query "dewitt widom." As the number of allowed joins increases, the number of joining networks to be explored grows exponentially, adding to the total response time.

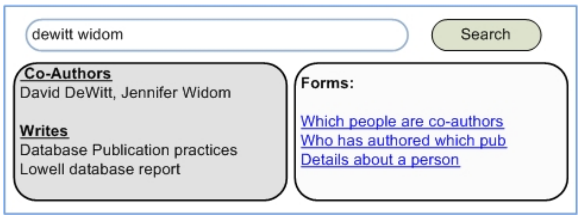
It may be possible to further optimize our implementation of KWS or implement a version of KWS for bounded main memory (by spooling to disk) but it is unlikely that we can escape the exponential growth of CN generation. Several efforts have focused on quickly producing just the top-k answers (of the ranked list), based on the heuristic that a user is likely to look only at the top few results produced by the system. Work on top-k answers falls into two groups: those that produce the exact top-k (e.g., [11]) and those that produce approximate top-k answers (e.g., [9, 17, 18]). While these approaches work very well in certain settings, they may fail in cases where a good ranking function is not available. Furthermore, these approaches are not effective when the top-k answers themselves require long join sequences.

Other recent solutions (e.g., [9, 18]) guarantee a polynomial bound on producing the first few answers. However, this theoretical bound is with respect to the size of the database, so it can translate into an impractical run time. More importantly, these bounds only hold for relatively simple ranking functions, such as ranking answers based on the height of their trees [11, 26]. Many ranking functions used in practice are more complex or contain "black-boxes" defined by the user. In such cases, again we have no way to estimate how long the search for even the first few top answers will take.

In summary, the run time performance of current KWS systems for relational databases is unpredictable. Given a user query, we have no way to know in advance how long it will take before the user sees any result. For certain queries this time can be very long indeed—on the order of tens of minutes—and more unsettlingly, for other queries, the system may even fail to return. This unpredictability is fundamental, and it poses a serious problem for adopting such systems in practice. We note that KWS systems suffer from this unpredictability especially for queries that involve long join sequences. In the next sections, we consider combining KWS with a forms based solution that does not suffer from this problem.

3. KEYWORD SEARCH USING FORMS

The basic idea of using forms in querying structured data is not new, of course. It has been used extensively for many years for naïve users to query Web databases (e.g., querying books on Amazon.com). Most recently [13] used forms to make databases easier to query. Additionally, [5] used forms to allow a naïve user to pose SQL queries with complex constructs over relational data. The focus of [5] was on using forms as a powerful and intuitive interface to issue queries with complex SQL constructs. However, they did


dewitt widom

Search

Co-Authors
David DeWitt, Jennifer Widom

Writes
Database Publication practices
Lowell database report

Forms:

Which people are co-authors
Who has authored which pub
Details about a person

Figure 2: An illustration of our proposed approach that combines KWS (in the left pane) and KWS-F (in the right pane).

not address performance issues or queries with long join sequences, and they did not consider operating in the same space as a generic KWS system.

We claim that the idea in [5] can be extended to apply to our setting. Hence, in what follows, we will first describe that solution, which we refer to as KWS-F. Next, we argue why KWS + KWS-F would be a good hybrid solution. Finally, we discuss that hybrid solution in depth.

3.1 Overview of KWS-F

The work in [5] was motivated by a simple question: how can we enable naïve users to pose complex SQL queries over a relational database, given that they do not know SQL? The basic idea is as follows. First, given the database, generate a set of SQL queries that are most likely to be asked by naïve users. Next, generate a set of query forms that encode those SQL queries (multiple SQL queries may map to the same form). The number of forms generated could be very large, in hundreds or even thousands. A key challenge of this work is how to search for forms effectively.

Two inverted indexes, one on the data set (*DataIndex*) and the other on the set of forms (*FormIndex*), are used to address this challenge. Given a keyword query $Q = q_1, \ldots, q_m$ the system first maps each term q_i to a set of possible schema-term interpretations/mappings $\{I\{q_i\}\}$ using the *DataIndex*. After this mapping is complete, a set of form-queries $\{Q_{form}\}$ is generated, where each query in $\{Q_{form}\}$ is conjunctive and contains m schema terms (one for each q_i), such that each term is derived from the corresponding $I\{q_i\}$.

All the queries in Q_{form} are issued over the *FormIndex*, which returns a set of form-ids, identifiers of the forms that contain all the terms in Q. The final answer is the union of all the forms returned. At this point, the user can examine this list, find the desired form, fill it out, and submit it. Once the form is instantiated and submitted, the SQL query corresponding to that particular form is evaluated and the results are displayed to the user.

Limitations of KWS-F While KWS-F works well for queries with complex SQL constructs, we believe that using KWS-F alone for simple keyword search is not a good idea. The main reason for this is that user queries in a KWS system fall roughly into two groups: easy queries and hard queries. Easy queries run very quickly, while hard queries take a long time or may not even return, as we discussed earlier.

KWS-F displays a set of forms, and the user must examine the forms, select promising ones, fill them out, submit them, examine the results, and potentially repeat, a tedious process. Thanks to its predictable performance, KWS-F is suitable for hard queries despite this tedium; however, KWS is more appropriate for easy queries where performance is not as important.

3.2 A case for combining KWS and KWS-F

We now make the case for combining KWS and KWS-F to build a keyword search system for relational data by describing some cases where KWS and KWS-F work as complementary approaches.

To achieve predictable performance and good coverage: As we stated earlier, we believe that a KWS system, if it is to be adopted widely, must be responsive and predictable. To implement

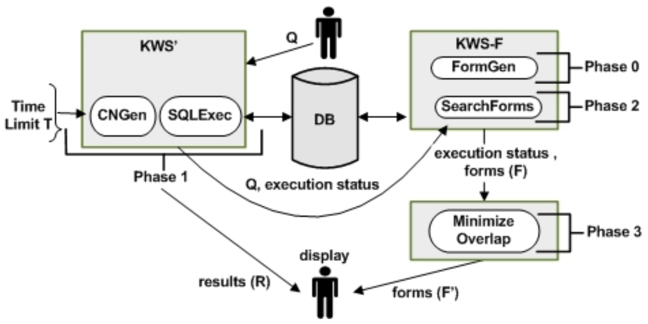


Figure 3: System architecture for the proposed hybrid approach.

the above idea, a natural solution is to impose a time limit on the keyword search system: when the time limit has been reached, a (possibly partial) result must be returned to the user no matter what.

In case the system has been able to explore only a portion of the answer space within this time limit, it runs the risk of having shown the users only some suboptimal answers, or worse, missed desired answers all together. In such cases, we believe it is highly desirable that the system give the user an idea about what the unexplored portion of the answer space "looks like" and what the user can do to explore that portion.

One way to do this is to combine KWS and KWS-F. Figure 2 illustrates this proposal. Given the keyword query "dewitt widom," the system returns a set of answers, just as in a traditional KWS system. But unlike these traditional KWS systems, our system also returns a set of forms to characterize the unexplored portion of the answer space. If the user does not find what he or she wants in the answers on the left, he or she can examine the forms on the right, and then click on a desired form to fill and submit.

Data and schema dependence is an artifact of KWS. For instance KWS performs badly for star schema or graphs with many outgoing edges. However, it is very efficient for simple queries over a simple schema. In contrast, by using forms and imposing a time limit, independent of schema, the hybrid system mitigates the impact of this dependence. Thus in a sense combining KWS and KWS-F allows us to use each approach where it works best: KWS to handle easy queries, and KWS-F to handle hard queries.

When ranking cannot help: Ranking can be of great benefit in a KWS system. A wealth of top-k systems successfully avoid exposing the performance bottleneck inherent to KWS to the end user by using a good ranking function. A system that combines KWS and KWS-F can also greatly benefit from the presence of a good ranking function. However, there are cases where a good ranking function does not exist or when many result tuples having the same score are returned in response to a user query. Additionally, top-k systems may also fail when the number of answers to a keyword query is less than *k* and all of the answers require many joins. Combining KWS and KWS-F can allow the user to explore the search space in these cases where KWS alone could be inefficient.

Forms offer a "guidance effect": We believe that by combining KWS and KWS-F, forms can potentially offer a good transition to go from an unstructured keyword query to the results of a structured query. Bernstein et al. [3] make a similar observation when they state that when querying structured data, a partially structured query interface is preferable to a completely unstructured interface because of its guidance effect.

In the next section we describe the proposed hybrid approach in detail in the context of a DISCOVER-like candidate network based system, and address the various challenges we face in building such a system.

4. THE HYBRID APPROACH

We start with an overview of our proposed hybrid approach and then drill down into its various aspects in the sections that follow.

Algorithm 1 : Hybrid approach (Phase 0)

```
Input: Set of tables $T = \{t_1, t_2, \ldots, t_N\}$
Parameters: Maximum number of joins $M_{max}$
Output: A set of form-ids $F$
Algorithm:
$F_0 = \{$forms consist of a single table $t\}$
$F_1 = \{$forms consist of two table pairs $(t_i, t_j)$, where $i \le j\}$
for ($k$ = 2; $k \le M_{max}$; $k$ ++) do
    for each pair of forms $f_i$ and $f_j \in F_{k-1}$, where $i \le j$ do
        if ($f_i.t_2 = f_j.t_1$, $f_i.t_3 = f_j.t_2 \ldots f_i.t_k = f_j.t_{k-1}$)
            Create a new form $f$ by combining $f_i.t_1$, $f_i.t_2, \ldots, f_i.t_k$, and $f_j.t_k$
            Insert $f$ in $F_k$
return $F = \bigcup_{k=0}^{M_{max}} F_k$
```

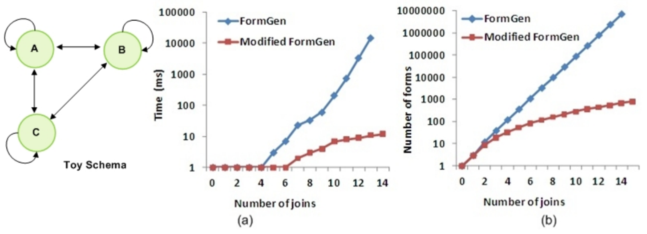


Figure 4: For the toy schema (a) the modified form generation algorithm is faster than the previous algorithm that does not eliminate duplicate join sequences, and (b) the modified form generation algorithm generates fewer forms even for long join sequences. Y-axis is in log scale.

To keep the discussion concrete, we describe the details of the proposed hybrid system using DISCOVER [12], which is a CN based system. However, as we discuss later, our approach can be applied to other KWS systems as well.

4.1 Overview

The architecture of the proposed hybrid system is shown in Figure 3. Like a KWS-F system, we start by generating a large set of forms offline. We refer to this stage as **Phase 0**. In Section 4.2 we propose a modification to the form generation algorithm in [5].

In **Phase 1**, the user's keyword query Q is sent to KWS$'$, which is a KWS system modified to operate within a time limit T. Specifically, the modified system attempts to only generate those CNs and executes those SQL queries that can be completed within time T. Eventually KWS$'$ will terminate, either because (a) it has finished executing all generated SQL queries, or (b) the time limit T has been reached, whichever is earlier. We describe this phase in more detail in Section 4.3

At this point, KWS$'$ sends query Q together with a status report on its execution to the KWS-F subsystem. This system executes query Q to obtain a ranked list of forms, as in a traditional KWS-F system. We refer to this portion as **Phase 2**.

Next, in **Phase 3**, the KWS-F subsystem examines the status report sent by KWS$'$ to see which forms have been "covered" by the KWS$'$ subsystem. These forms are eliminated from the ranked list of forms. After this removal, the KWS-F subsystem returns the revised list of forms. The hybrid system then combines this list of forms with the list of answers produced by KWS$'$ and presents this combination to the user.

We will now drill deeper into each phase of the hybrid approach starting with Phase 0.

4.2 KWS-F: form generation (Phase 0)

As in KWS-F systems, form generation in the hybrid system is also performed offline. [5] presents a form generation algorithm that generates forms combinatorially based only on the key-foreign key relationships in the underlying schema graph. However, in trying to use this form generation algorithm for long join sequences, we observed that the number of forms generated grows steadily and that the set of generated forms contains many duplicates. As an example, applying this algorithm with M_{max} = 8 to the toy schema

Algorithm 2 : Hybrid approach (Phase 1)

```
Input: A keyword query $Q = [q_1, q_2, \ldots, q_n]$
Parameters: Total query execution time $T$
Output: A set of unexecuted SQL queries $S$, result data tuples $R$, and list of CN templates $G$

Main:
*Pick keyword ordering*
$S = \{\}, R = \{\}, G = \{\}$
*Start thread* $CN_{gen}$
*Start thread* $SQL_{exec}$
*wait for* $CN_{gen}$ *and* $SQL_{exec}$ *to complete*
return $S$, $R$ and $G$

$CN_{gen}$ thread:
$prev = null$
while execution time ¡ $T$ and CN generation has not terminated
    $cn_j \leftarrow$ GenCN($Q$) //modified $GenCN$ from $KWS$
    Map $cn_j$ to corresponding SQL query $s_j$
    Add $s_j$ to $S$ //sorted on query cost C
    $curr$ = GenCNTemplate($s_j$)
    if $prev$ is not null and $prev \neq curr$
        $G \leftarrow G \cup prev$
    $prev = curr$
return $G$

$SQL_{exec}$ thread:
while (execution time ¡ $T$ and $S$ has elements) or $CN_{gen}$ is running
    if $S$ has elements:
        $s \leftarrow Dequeue(S)$
        $R \leftarrow R \cup Execute(s)$
return $R$ and $S$
```

in Figure 4 generates a total of 29,523 forms, of which 29,304 are duplicates; this leaves only 219 unique join sequences.

Duplicate form elimination is important for increasing the efficiency of the form generation step, minimizing storage, and presenting the forms to the user. To address this issue, we propose a modified form generation algorithm (Algorithm 1). The algorithm takes the maximum number of allowed joins and the schema graph as the input and generates all possible key-foreign key join sequences until the maximum number of joins is reached.

Determining equivalent join sequences is simple in the keyword search context, because our join sequences do not contain any projections or selections, and also because like KWS systems, we consider only key-foreign key joins. This allows us to exploit the property that joins are both *associative* and *commutative*. If there are N tables in the schema graph and the maximum number of joins allowed is M_{max}, then the new approach generates $\sum_{m=0}^{m=M_{max}} \binom{N+m}{m+1}$ join sequences, instead of $\sum_{m=0}^{m=M_{max}} N^{m+1}$. Figure 4 shows this gain for the toy schema.

Furthermore, in terms of query coverage, since a form is like a template for multiple CNs, the forms in a KWS-F system can cover all the queries generated by any KWS system, i.e., the forms in KWS-F system have at least the same expressive power as any KWS system that explores only key-foreign key joins.

4.3 KWS$'$ (Phase 1)

Phase 1 is the modified KWS system. In this subsection we will address the changes we make to a traditional DISCOVER-style CN based system, with a focus on CN generation and SQL query execution. These changes are specific to DISCOVER-like systems.

In Phase 1, we start by exploring how the time budget T needs to be divided between the two major components of KWS$'$: CN generation and SQL execution. There are multiple options and various constraints that we need to take into account in order to utilize T well in our hybrid system.

One option is to start SQL query execution only after CN generation has terminated. However, as we pointed out in Section 2.1, CN generation can take up a large portion of the total query execution time in a KWS system as the number of allowed joins increases. Therefore, waiting for CN generation to terminate before starting the SQL query execution step is not a good option.

We could divide T into two parts and then spend the first part generating CNs and the second part executing the SQL queries for the generated CNs. The main issue with this approach is that it is hard to determine how T should be divided. Failure to estimate this division accurately can lead to the system being idle when it could be generating more CNs or executing more SQL queries.

One natural alternative that comes to mind is to interleave CN generation and SQL query execution, as is done by most current KWS systems. While this approach has the advantage of not wasting any CNs that were generated, it does not take query cost into account at all. With this approach, the system could end up executing a time-consuming query that was generated prior to many inexpensive queries. Based on this observation, we propose a heuristic that is intended to maximize the number of SQL queries that the hybrid system executes.

To account for query execution cost, we propose Algorithm 2 where CN generation and SQL query execution are performed in a producer-consumer fashion using two separate threads. Once a CN cn_j is generated, the corresponding SQL query s_j is placed in a priority queue S, which is ordered by query execution cost C. In our implementation, this cost C is in disk page fetches and is obtained by using the EXPLAIN function of the underlying database. The value of C depends on various factors, like the distribution of the underlying data, machine specifications, join selectivity estimates, etc. Estimating C accurately is a complex problem and is outside the scope of this work. However, we note that most query optimizers estimate the relative costs for multiple queries fairly accurately and that this is sufficient for our approach because we only use C to determine the query execution order.

In addition to S, we also maintain a list of CN templates G. A CN template contains the tables involved in a CN and is obtained by applying the function **GenCNTemplate** over a SQL query s_j. We also modify the **CNGen** algorithm in DISCOVER to produce CNs in a particular order. More specifically, we ensure that all CNs that map to a particular CN template are generated together. We will talk about S, G, and the reason for this modification to **CNGen** in Section 4.5.

While CN generation continues, SQL queries are executed based on where they occur in the priority queue S. When the system times out, it could be the case that some SQL queries were generated and placed in the queue, but not executed. The results R of the executed queries, the priority queue of unexecuted queries S, and the list of CN templates G are then passed to the next phase of the hybrid system. In Phase 3, these values are used to minimize the overlap between the portion of the search space that was explored by KWS$'$ and KWS-F.

4.4 KWS-F: search (Phase 2)

Once KWS$'$ has terminated, the results R of the executed queries are displayed to the user. In addition to this the query Q and the execution status of KWS$'$ are passed to the KWS-F subsystem. If KWS$'$ completes in the given time budget, no forms need to be displayed. Otherwise, the hybrid system is left with the task of allowing the user to explore the search space that KWS$'$ could not explore given the time budget T.

This is accomplished by using forms and this step is exactly like the form-search step in [5]. The forms that the hybrid system contains were generated in Phase 0. In addition to this, a full-text index $FormIndex$ over the forms was also created in Phase 0. As described in Algorithm 3 of the hybrid system, we take each term q_i in the user query Q and map it to its corresponding schema term, retaining any schema terms already present in Q. The **Bucket Algorithm** in [5] is used to generate multiple conjunctive, schema term queries which cover all possible interpretations of the user query Q. This set of modified queries is issued over $FormIndex$, and the union of the resulting forms F is returned.

Algorithm 3 : Hybrid approach (Phase 2)

```
Input: A keyword query $Q = [q_1, q_2, \ldots, q_n]$
Output: A set of forms $F$

Algorithm:
$FormTerms[] = \{\}, F = \{\}, B = \{\}$
//Get all interpretations of each $q_i$ in $FormTerms_i$
for each $q_i \in Q$
    if DataIndex($q_i$) returns $< table >$ // $q_i$ is a data term
        Add each $table$ to $FormTerms[i]$
    Add $q_i$ to $FormTerms[i]$ // $q_i$ could be a form term
$B = \{$BucketAlgorithm$(FormTerms[])\}$ // set of conjunctive, schema queries
// Get form-ids based on the queries in $B$
for each $b_j \in B$
$F \leftarrow F \cup \{FormIndex(b_j)\}$
return $F$
```

We now have a list of forms F and the set of results R from the queries that KWS$'$ executed within the time budget T.

4.5 Minimizing overlap (Phase 3)

As discussed earlier, Phase 1 of the hybrid algorithm passes S (the list of unexecuted SQL queries), R (the results of executed queries), and G (a list of CN templates) to Phase 3. In Phase 3, we focus on the interaction between the KWS$'$ and KWS-F subsystems of the hybrid system. In particular, we explore the various options of dealing with the overlap in the SQL queries that both subsystems cover.

There are two reasons why overlap minimization is important: (i) we do not want to display redundant forms for simple queries already answered by KWS, and more importantly, (ii) given that the screen real estate is a limited resource we try to eliminate as many forms as possible.

To this end, we focus on the relationship between the CNs generated, the SQL queries that were executed by KWS$'$, and the forms returned by KWS-F. Many CNs (and their corresponding SQL queries) can map to a single form. This mapping implies that we cannot eliminate a form returned by the KWS-F subsystem unless all the corresponding SQL queries were generated and also executed to completion in Phase 1.

In other words, if a query corresponding to a form f is present in the priority queue S, we cannot remove f from the list of forms returned to the user. However, this does not mean that we can discard a form f if no query corresponding to it is present in S. This is because it may very well be that KWS$'$ timed out before generating all the CNs corresponding to f. Thus, we can discard a form f only if the following two conditions have been met:

Condition 1. All CNs that map to the form f have been generated in Phase 1. The list of CN templates G populated in Phase 1 is used to verify this condition.

Condition 2. All SQL queries corresponding to the form f have been executed in Phase 1. The list of unexecuted queries S is used to verify this condition.

We now explain how Condition 1 is verified by using G, the list of CN templates. As mentioned earlier, a CN template consists of the tables involved in a CN and is obtained by applying the function **GenCNTemplate** over a SQL query s. As shown in Algorithm 2, we populate G during the CN generation in Phase 1. We modified the CN generation function **GenCN** used in KWS$'$ to ensure the following property: all the CNs corresponding to SQL queries that map to same set of tables are generated "together," i.e., the **GenCN** function moves on to generate the next set of queries only after all the queries corresponding to a particular CN template are generated. Further, as shown in Algorithm 2, we populate G with a new CN template g only after all the SQL queries corresponding to g have been generated.

The above property leads us to the following claim: Let t_g be the time when CN template g is placed in G. Let t_s be the time when a SQL query s is placed in S. Then, $\forall s \in S$, if $t_s > t_g$

Algorithm 4 : Hybrid approach (Phase 3)

```
Input: A set of unexecuted SQL queries $S$, result data tuples $R$, list of CN templates $G$ and the set of forms $F$
Output: Result data tuples $R$ and list of forms $F'$

Algorithm:
$F' = F, tempForm = \{\}, canDiscard = true$
Sort $S$ on number of joins
for each CN template $g_i \in G$
    for each SQL query $s_j \in S$
        $canDiscard = true$
        if number of joins in $s_j >$ number of joins in $g_i$
            break
        if GenCNTemplate $(s_j) = g_i$
            $canDiscard = false$
    if $canDiscard = true$
        $tempForm \leftarrow$ form corresponding to $(\{g_i\})$
        $F' \leftarrow F' - tempForm$
return $R$, $F'$
```

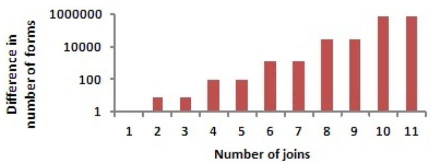


Figure 5: Comparing FormGen and the modified FormGen algorithm that eliminates duplicates, for the DBLife schema. The number of maximum joins is shown on the X-axis. The Y-axis shows the difference in the number of forms on the log scale. The new approach generates 30-40% fewer forms.

then **GenCNTemplate**$(s) \neq g$. In other words, $\nexists$ $s \in S$ such that $t_s > t_g$ and **GenCNTemplate**$(s) = g$.

The presence of a CN template g in G therefore ensures that all the SQL queries corresponding to it were generated. Further, the absence of a query s in S corresponding to g implies that the query must have been executed. Phase 3 of the hybrid approach uses these properties for minimizing the overlap as presented in Algorithm 4. We start by looking at all the unexecuted queries in S and sorting them based on the number of joins in descending order. We do this for performance reasons because G is already sorted by the number of joins given that CNs are generated iteratively. We go through each CN template g in G, and if there is no query in S that can map to the current template g, then the corresponding form can be discarded from F. This leaves us with a set of forms F'.

At this point the set of forms F' and the results of the executed SQL queries R are presented to the user. Given the many-to-one mapping between SQL queries and forms, we note that the overlap minimization algorithm does not guarantee that all overlap between KWS$'$ and KWS-F can be eliminated.

5. EXPERIMENTAL EVALUATION

In this section we evaluate our proposed hybrid approach. We ran our experiments using PostgreSQL 8.3.6 on an Intel(R) Core(TM) 2 Duo 2.33 GHz system with 3 GB of RAM. All query processing algorithms were implemented in Java, and JDBC was used to connect to the database. The inverted indexes for KWS-F were implemented using Lucene [25]. We evaluated the proposed approach over the DBLP [7] and the DBLife [6] datasets. We used a 40 MB snapshot of the DBLife dataset which has 801,189 tuples in 14 tables. We downloaded the current version of dblp.xml which is 680 MB in size. The corresponding relational data measures 1340 MB.

5.1 Sub-systems of the hybrid approach

We now present some results for our DISCOVER-based implementation of the hybrid system described in Section 4. Due to space constraints we present these results only over the DBLife dataset.

Evaluating the modified form generation algorithm: Figure 5 shows the number of duplicate forms that the modified form generation algorithm eliminates when compared to the algorithm in [5]. The number of forms is plotted on a log scale. In our setting, where a very large number of forms exist, eliminating duplicate forms is important for storage reasons as well as ranking and presenting forms. We find that on average, for the DBLife schema, the new approach generated 30-40% fewer forms than the form generation algorithm in [5].

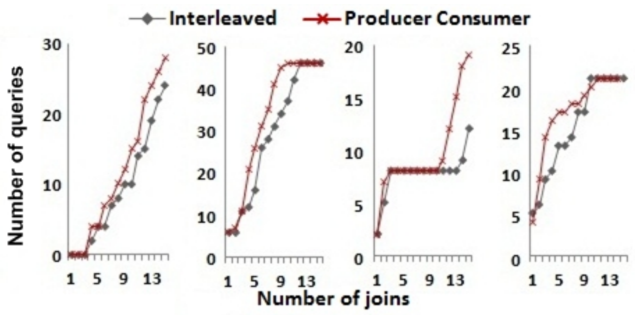


Figure 6: Number of queries executed at T = 15 seconds for the interleaved and producer-consumer style query execution approaches. The producer-consumer approach performs better in most cases.

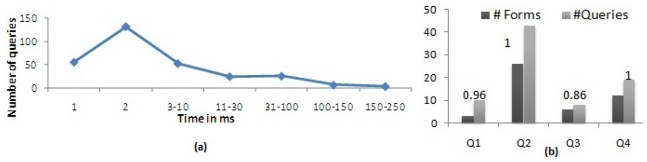


Figure 7: (a) Shows the distribution of the estimated execution times for the SQL queries generated in response to the keyword query "dewitt widom." This distribution shows that taking SQL query execution cost into account can help increase the total number of SQL queries executed by the KWS sub-system within the time limit T. (b) Shows the number of queries and forms eliminated by the hybrid system after overlap minimization.

Evaluating KWS$'$: Figure 6 compares the two approaches outlined for the interaction between CN generation and SQL query execution in the presence of a timeout based strategy: (i) the interleaved approach, where the SQL query corresponding to a CN is executed before the next CN is generated, and (ii) the producer-consumer style interaction where SQL queries are ordered and run based on their estimated execution cost. We set the timeout $T = 15$ seconds, and plot the number of SQL queries executed by each approach, with time on the X-axis. The producer-consumer based approach that takes SQL query execution cost into account performs better than the interleaved approach in most cases.

Figure 7(a) shows the distribution of the SQL queries generated in response to the keyword query "dewitt widom." Of the 296 queries, 201 queries take under 2 ms each to execute while 3 queries take around 200 ms each. This tells us that taking SQL query execution cost into account can help increase the total number of SQL queries executed by the KWS$'$ sub-system. We find that ordering the queries by estimated execution cost does yield some benefit, i.e., the overall number of queries executed increases.

Evaluating the overlap minimization algorithm: We start by defining a metric for overlap minimization. Even though KWS$'$ and KWS-F operate at different granularities (SQL queries and forms, respectively), the results of both approaches can be mapped to SQL queries. Since the number of ways in which a form can be instantiated is very large (and is data dependent), the overlap for a particular keyword query can be calculated as follows. In addition to keeping track of the CN templates whose CN generation has run to completion (i.e., the CNs in G), we also keep track of the number of SQL queries that were generated for each CN template in Phase 1 (KWS$'$) of the hybrid approach. Let this number be $\mathcal{N}(g_i)$ for each $g_i \in G$. Let G' be the set of CN templates, s.t. the form corresponding to each $g_i \in G'$ has been eliminated. The metric to quantify overlap $\mathcal{M}$ is computed as follows: $\mathcal{M} = \sum_{g_i \in G'} \mathcal{N}(g_i) / \sum_{g_i \in G} \mathcal{N}(g_i)$. The goal of the overlap

Q1	distributed david	Q6	keyword search relational
Q2	relational john	Q7	improving performance relational david
Q3	database michael	Q8	jeffrey optimal kazutsugu index
Q4	performance hector	Q9	micheli algorithm optimization kevin
Q5	database michael david	Q10	improving performance relational databases

Figure 8: Keyword queries issued over the DBLP dataset.

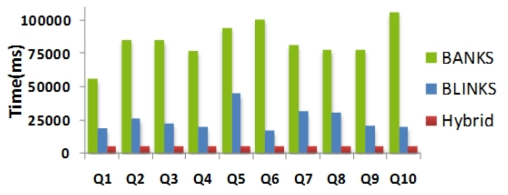


Figure 9: The online response times for BANKS-II, BLINKS, and the hybrid approach over the DBLP dataset. T = 5 seconds for the hybrid approach and M_{max} = 10 for all three approaches. Other than the obvious claim, this graph demonstrates that (1) there are keyword queries for which current KWS systems are inefficient and (2) response time can be bound to an acceptable number with the hybrid system, without compromising query coverage.

minimization algorithm in the hybrid approach is to maximize $\mathcal{M}$ without compromising query coverage. $\mathcal{M}$=1 implies that there is no overlap between the queries executed by KWS$'$ and the forms displayed.

We set T to 15 seconds and run the hybrid system for 4 keyword queries. Figure 7(b) shows the number of forms and the number of SQL queries eliminated for each query and also shows the corresponding $\mathcal{M}$ values. For the query "agrawal chaudhuri das," $\mathcal{M}$ = 0.96. This implies that after the overlap minimization step there is only 4% overlap between the forms displayed to the user and the queries already executed by KWS$'$. The forms for the remaining 4% of the queries cannot be eliminated because not all the SQL queries that correspond to those forms were executed by KWS$'$ within time T.

5.2 Hybrid approach for other KWS systems

Up until now, we have presented the results for our DISCOVER-based implementation of the hybrid system. In this section, we show through experiments that the benefits of the hybrid approach also apply to more recent systems like BANKS-II, BLINKS, and EASE. We present the results for all these systems over the DBLP dataset using the 10 queries listed in Figure 8.

Our goal for this evaluation is three-fold. First, even though the low run time for the hybrid system is to be expected, given that we place a time limit on its execution, we want to know how it fares compared to the state-of-the-art in KWS systems, for the same query coverage. Second, given various values for time limits, we want to determine the increase in coverage provided by the hybrid approach for various KWS systems. Third, given that the hybrid approach leverages offline computation (i.e., offline form generation and indexing) to achieve predictable performance, we are interested in examining and comparing the pre-computation strategies employed by other KWS systems to the hybrid approach proposed in this paper.

We start by looking at BANKS-II and BLINKS, starting with short descriptions of both systems.

BANKS-II: Like BANKS-I, BANKS-II [1] operates on a data graph where each tuple is a node and each foreign-key relationship between tuples is represented as a bidirectional edge. In BANKS-I single source shortest paths iterators are run in a BFS fashion from each node containing a keyword. As soon as the iterators meet, a result is produced. This technique is improved in BANKS-II which uses both forward and backward expansion. While BANKS-II performs well for a variety of keyword queries, its performance significantly degrades in the presence of high-degree nodes during the expansion process.

BLINKS: Another graph based approach —BLINKS—avoids the NP-hard Steiner tree problem by giving up on completeness of

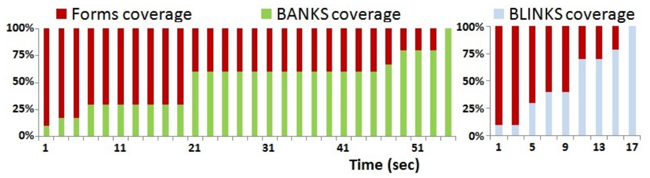


Figure 10: Breakdown of query coverage of the KWS and KWS-F subsystems for BANKS-hybrid and BLINKS-hybrid for different values of timeout T.

answers. Specifically, it uses "distinct root" semantics and only explores a portion of the search space. This allows for efficient answer generation at the cost of some coverage and greatly improves the run-time performance. BLINKS [10] also uses data partitioning in addition to bidirectional search proposed in BANKS [1]. This system relies heavily on the ranking function used and performance guarantees cannot be made if the ranking-function is a black-box. The system returns only the roots of the answers and their distances from each keyword query. Reconstructing the answer trees from this information requires extra work. Additionally, the graph and bi-level index used in BLINKS must fit in memory for BLINKS to be efficient.

Figure 9 shows the online run-times for BANKS-II and BLINKS alongside the time taken by the hybrid approach when T is set to 5 s. These numbers are for the 10 queries listed in Figure 8. We remove ranking from both systems and set the maximum number of joins allowed in the result tuples to 10. Even though these numbers demonstrate the obvious, they show that adding forms to a KWS system might be necessary because there are queries for which each of these systems have poor performance, and also that adding forms to each of these systems can reduce the absolute time with very little overhead.

Our second goal is to experimentally evaluate the gain in coverage obtained by using the hybrid approach. The number of results that would be generated if the KWS system was allowed to run to completion is considered to be 100% coverage. Consider the keyword query Q1 for which BANKS-II and BLINKS take 56 seconds and 18 seconds, respectively, to achieve 100% coverage. We plot the coverage achieved by the KWS and KWS-F portions of the hybrid system as we vary the time limit T on the X-axis in Figure 10. As an example if T was set to 5 seconds, BANKS would get 18% coverage while the remaining 82% would be covered by forms in the hybrid system. This demonstrates the usefulness of forms in increasing the query coverage of a KWS system, specially for interactive response times (i.e., low values of T).

EASE: Offline pre-computation is a common technique used by many systems to better online performance. Graph based approaches like EASE [21], [22], [27] and TASTIER [20] heavily utilize offline computation by indexing sub-graphs. These approaches are similar to generating and indexing forms offline, but operate at the data level by exploiting the notion of r-radius Steiner graphs [21]. Since these systems bypass the expensive graph search process, they have good response times (on the order of a few seconds).

Table 2 shows the offline time spent by all 5 systems. DISCOVER requires no pre-computation, while BANKS and BLINKS need to pre-compute information about the underlying graph. 22 MB is the size of the graph file used by BLINKS and BANKS. For the DBLP dataset, our implementation of EASE required 30 GB of disk space and over 90 minutes of pre-computation time. This offline computation helps EASE achieve good online performance. The interested reader can refer to the appendix for more experiments with EASE.

As shown in Table 2, KWS-F causes very little offline overhead compared to EASE. Furthermore, by virtue of their nature, EASE-like approaches are vulnerable to changes in the underlying data. In contrast, forms are generated solely on schema information and are indifferent to changes in the data itself. More importantly, since

	DISCOVER	BANKS-II	BLINKS	EASE	Hybrid
Offline time	0	137 s	68 s	92 min	5 s
Offline storage space	0	22 MB	22 MB	30 GB	1 MB

Table 2: Offline time and storage space consumed by each of the 5 systems.

submitting a form corresponds to executing a SQL query over the actual data, not only can the hybrid approach help EASE for interactive values of T, it can also help with correctness during the time when changes in the data have not yet been propagated to the index.

6. RELATED WORK

The breadth and depth of work related to the ideas presented in this paper is extensive.

KWS systems: In the interest of space and since we have already covered some KWS solutions like DISCOVER, BANKS, BLINKS, KWS-F and EASE in previous sections, we will talk about other related work here.

CN generation remains the most costly component in most systems and is proven to be exponential. Sagiv et al. [9, 18] and [17] present an algorithm for enhancing the performance of locating minimal and total sub-graphs within a graph. Ding et al. [8] propose a dynamic programming based solution for finding candidate networks. While their approaches have the ability to speed up the selection of candidate networks in KWS, all the systems recognize that at some point the space of join networks gets so large that searching through it becomes infeasible. Our focus is to provide an alternative in this case.

Top-k: A wealth of work exists in top-k systems. [11, 26] and many others use ranking strategies that favor short join sequences, and focus only on generating the top few answers efficiently. Luo et al. [26] treat tuples as "virtual documents" and use search engine like retrieval and ranking. Top-k systems perform well in practice when a good ranking function exists and when the user is only interested in the top few answers, all of which require very few joins. However, in the event that a good ranking function does not exist, or when retrieving the top-k answers itself is very expensive, the hybrid approach offers a viable alternative.

Forms: A system that exploits KWS over forms for querying a relational database was proposed by Chu et al. [5]. Unlike [5], who focus on providing complex query constructs like aggregates, group by, and so forth, we are focused only on answering those queries that KWS can answer. Jayapandian et al. [13, 14] described an approach that automatically generates forms for a database based on a sample query workload. Since their goal is to create a small set of forms, they do not consider the problem of choosing from a set of forms. Instead, when the forms do not support a user query, they allow users to modify an existing form.

DataCloud [19] and DataLens [23] are some recent works that offer an overview of the search space to the user, like forms do.

Avatar [16] combines KWS with form-based support for structured querying. However, their goal is not to solve the performance problem we identify in this paper; rather, it is to suggest structured queries that may be of interest to users based upon their keyword queries. All of this form-based work emphasizes the fact that non-programmers often find forms a useful way to query structured data.

7. CONCLUSION

Our goal in this paper was to explore techniques that allow keyword search over relational data to be implemented in such a way that the system can guarantee a reasonable response time. Our main idea is to let the traditional keyword search generate all the answers it can within some time bound, and to augment the search with a form-based approach that "covers" potential answers that the keyword search could not find in the specified time limit. Results from experiments with this approach indicate that it is successful in always returning a covering combination of answers and forms in a bounded and predictable amount of time.

We regard this work as a first step toward building this kind of system, and hope that it is a springboard for follow-on work that improves the performance and quality of such systems. In general, exploring the trade-offs between the form-based component and the keyword-based component is fertile ground for future work. For example, a form that returns no answers when executed can convey information to the user about facts that are not present in the database, which is something that seems difficult to capture with a pure keyword-based approach. As another example, if a keyword search returns too many answers that have similar rankings, exhibiting a form may prompt a user to fill in attributes and narrow her search more easily than requiring her to come up with additional keywords to disambiguate the results. Most likely the answers to these and other related questions will require studies of user behavior when interacting with such systems.

8. REFERENCES

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S. Sudarshan. BANKS: browsing and keyword searching in relational databases. In *VLDB '02*.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: enabling keyword search over relational databases. In *SIGMOD '02*.

[3] A. Bernstein and E. Kaufmann. Making the semantic web accessible to the casual user: Empirical evidence on the usefulness of semiformal query languages. In *TKDE*.

[4] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD '09*.

[5] E. Chu, A. Baid, X. Chai, A. Doan, and J. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *SIGMOD '09*.

[6] DBLife. `http://dblife.cs.wisc.edu`,.

[7] DBLP. `http://www.informatik.uni-trier.de/~ley/db/`,.

[8] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE '07*.

[9] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD '08*.

[10] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD '07*.

[11] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB '2003*.

[12] V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *VLDB '02*.

[13] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *Proc. VLDB Endow.*, 1(1).

[14] M. Jayapandian and H. V. Jagadish. Automating the design and construction of query forms. In *ICDE '06*.

[15] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB '05*.

[16] E. Kandogan, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar semantic search: a database approach to information retrieval. In *SIGMOD '06*.

[17] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: Steiner-tree approximation in relationship graphs. *ICDE'09*.

[18] B. Kimelfeld and Y. Sagiv. Efficiently enumerating results of keyword search over data graphs. *Inf. Syst.*, 33(4-5).

[19] G. Koutrika, Z. M. Zadeh, and H. Garcia-Molina. Data clouds: summarizing keyword search results over structured data. In *EDBT '09*.

[20] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD '09*.

[21] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD '08*.

[22] G. Li, X. Zhou, J. Feng, and J. Wang. Progressive keyword search in relational databases. In *ICDE '09*.

[23] B. Liu and H. V. Jagadish. Datalens: making a good first impression. In *SIGMOD '09*.

[24] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD '06*.

[25] Lucene. `http://apache.lucene.org`,.

[26] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD '07*.

[27] A. Markowetz, Y. Yang, and D. Papadias. Reachability indexes for relational keyword search. In *ICDE '09*.

[28] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD '09*.

[29] M. Sayyadian, H. Lekhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE '07*.

[30] S. Tata and G. M. Lohman. Sqak: doing more with keywords. In *SIGMOD '08*.

APPENDIX

A. EXPLAINING THE PREDICTABLE PERFORMANCE OF KWS-F OVER KWS

In Section 2.1, we identified the performance bottlenecks that make current KWS solutions unpredictable and therefore difficult to adopt. We claim that KWS-F approaches do not suffer from these same performance problems and are suitable for queries with long join sequences. In this section, we highlight the fundamental differences between KWS-F and KWS systems that allow us to make this claim.

Preliminaries: Let $T_{KWS} = T_{TSGen} + T_{CNGen} + T_{SQLExec}$ $(+T_{User})$ be the total time taken by a KWS system, where T_{TSGen} is the time for generating the tuple sets, T_{CNGen} is the candidate network generation time, and $T_{SQLExec}$ is the time to execute all the generated SQL queries. T_{User} corresponds to the time taken by the user to browse through the results of all the executed SQL queries and find the relationships of interest to them.

Let $T_{KWS-F} = (T_{FormGen}) + T_{FormSearch} + (T'_{User}) + T'_{SQLExec}$ be the the total time taken by a KWS-F system, where $T_{FormGen}$ is the time spent on offline form generation and is not a part of the response time of the system. $T_{FormSearch}$ is the time taken to go from the user's keyword query to the point where the system displays the forms to the user. T'_{User} is the time taken by the user to choose and fill the form(s) that interest him, and $T'_{SQLExec}$ is the time taken to execute only the corresponding SQL queries.

Query	FormGen	FormSearch	CNGen	FormGen	FormSearch	CNGen
	3 joins			6 joins		
agrawal chaudhuri das	1.09	0.065	0.38	2.15	0.08	NA
dewitt widom	1.09	0.05	0.234	2.15	0.06	81.2
sigmod keyword search	1.09	0.09	0.5	2.15	0.09	18
publications icde	1.09	0.1	0.09	2.15	0.16	249

Table 3: Comparing T_{CNGen} in a KWS system to $T_{FormGen}$ and $T_{FormSearch}$ in a KWS-F system. All times are in seconds.

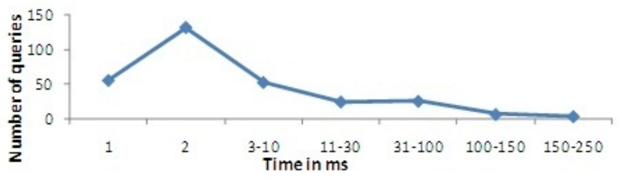
Analysis: In our experiments we find that T_{TSGen} is small (i.e., a few milliseconds) even for long join sequences. We ignore T_{TSGen} for the remainder of this discussion and structure this discussion around the performance bottlenecks in a KWS system: T_{CNGen} and $T_{SQLExec}$.

The CNGen step in a KWS system is analogous to the combination of the form generation and form search steps in a KWS-F system. Table 3 shows $T_{FormGen}$ (offline) and form $T_{FormSearch}$ along with T_{CNGen} for 4 queries. While the CN generation times are comparable to the sum of form generation and form searching times for 3 joins, T_{CNgen} gets much larger in the 6 join case. In what follows, we investigate the causes for this time difference.

We start by comparing the properties of CNs and forms. In a KWS system, there is a one-to-one correspondence between CNs and SQL queries. On the other hand, there is a one-to-many correspondence between forms and SQL queries in a KWS-F system. Said otherwise, each form can be instantiated in many ways and each instantiation corresponds to a SQL query. Also, form instantiation is performed by the user, i.e., forms do not contain any data terms. Each CN on the other hand maps to an executable SQL query. In a way, this implies that KWS-F systems operate at a different granularity than KWS systems.

Table 1 shows that as the number of joins increases, the number of joining networks that needs to be explored to generate candidate networks increases exponentially. As discussed in Section 2.1, it is well-known that variations of the CN generation problem are NP-complete. This complexity is at the core of the poor performance of CN generation in the presence of long joining sequences.

As stated earlier, the CNs in a KWS system are both *total* (contain all the keywords specified by the user) and *minimal* (we can-



(a)

Figure 11: SQL query times for the query "dewitt widom" with 6 joins. A small number of queries take a disproportionately large execution time.

not remove any tuple from a CN and still have a total CN). While being total is required, minimality is imposed to bind the CN generation step in a KWS system. Forms, on the other hand, contain schema terms and no data terms. Therefore, unless the user's keyword query contains only schema terms, the forms returned to the user cannot be total by design. Also, since the form instantiations are user-dependent, the minimality condition does not hold. The forms in KWS-F system are therefore neither total nor minimal. This relaxed requirement contributes to the scalability of a KWS-F system.

In summary, obviously KWS-F systems do not scale by efficiently solving the NP-complete problem that CN generation faces. Instead, the predictable performance comes from the fundamental differences between CNs and forms and from the pre-computation of forms, which allows replacing the CN generation problem in a KWS system with the much simpler document search problem in a KWS-F system. Additionally, even though the forms in a KWS-F system are in some sense an approximation of the CNs in a KWS system, a user can find any answer that KWS system would return using a KWS-F system.

In addition to the sub-problems that each system solves, it is important to note that the results obtained by both systems are also very different. The final result of a KWS system is a set of data tuples R. The elements in R are obtained after the KWS system executes a number of SQL queries. In contrast, a KWS-F system returns a set of forms F. Each form maps to a set of SQL queries and needs to be instantiated and executed to get to the resulting data tuples.

Query	Approach		Number of Joins							
			1	2	3	4	5	6	7	8
agrawal chaudhuri das	KWS	# JNs	14	196	3252	46714	581340	NA	NA	NA
		# CNs	0	0	0	88	88	NA	NA	NA
	Hybrid	# JNs	14	196	3252	46714	42311	42401	42415	42458
		# CNs	0	0	0	88	88	87	86	88
dewitt widom	KWS	# JNs	14	178	1214	6592	37788	218842	NA	NA
		# CNs	0	6	6	46	46	296	NA	NA
	Hybrid	# JNs	14	178	1214	6592	37788	52063	52231	53095
		# CNs	0	6	6	46	46	46	46	46
keyword search sigmod	KWS	# JNs	7	21	93	896	7357	60294	555328	NA
		# CNs	1	1	1	8	45	45	NA	NA
	Hybrid	# JNs	7	21	93	896	7357	60294	27210	27179
		# CNs	1	1	1	8	8	45	45	45
publications icde	KWS	# JNs	4	13	60	595	5121	43378	208359	406567
		# CNs	0	0	0	7	7	55	55	55
	Hybrid	# JNs	4	13	60	595	5121	43378	208359	406567
		# CNs	0	0	0	7	7	55	55	55

Table 4: Comparing the join networks and candidate networks generated for KWS and for the hybrid approach. As the number of joins allowed grows, many JNs need to be considered before finding a CN, thus leading to a performance bottleneck.

A KWS system executes all the SQL queries that it generates. We find that the number of queries to be executed can grow very quickly and that executing all of them can take up a large portion of the total execution time. We also find that as the number of queries grows, a small number of queries (typically those involving long joins) take up a disproportionately large portion of the execution time. The keyword query "dewitt widom," for example, generates 6 SQL queries to be executed for 2 joins, and this grows to 296 for 6 joins. Figure 11 shows that of these 296 queries, 201 queries take under 2 ms each to execute while 3 queries take around 200 ms each. Evaluating these expensive queries might be unnecessary if we are not even sure if the user is interested in them. A KWS-

F system leverages user input and only executes those queries in which the user shows interest.

B. DBLIFE SCHEMA

The schema for DBLife (dblife.cs.wisc.edu) is presented in Figure 12. This dataset describes entities and relationships in the database research domain.

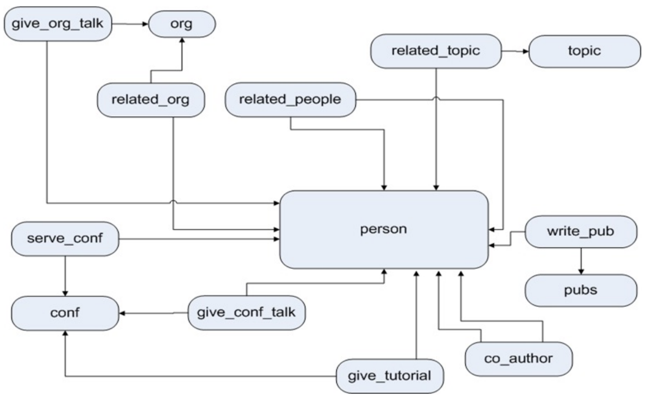


Figure 12: Relational schema for the DBLife dataset.

C. OPTIMIZING KWS′

The following optimizations can also help with the performance of the KWS′ sub-system of the proposed hybrid system.

C.1 Number of joins as a parameter

Like DISCOVER, the hybrid approach could also use a value M (maximum number of allowed joins) to restrict the number of CNs that the system evaluates. This value can be determined based on the schema, for schemas without many-to-many relationships or cycles. For other schemas this value can be set to a reasonable number. A conservative estimate of M_{max} in a traditional KWS system would hurt coverage. In the hybrid approach a conservative estimate of M only means that a form will be displayed for CNs of size greater than M. A reasonable choice for M could be the point at which a steep increase in candidate network generation time is observed for a set of representative queries. Setting this value in addition to T can help the system tremendously by reducing the overhead required in retaining all CNs that would need to be expanded if this threshold was not set.

C.2 Starting keyword

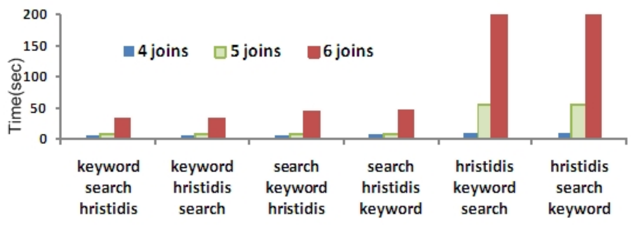


Figure 13: All 6 permutations of the keyword query "hristidis keyword search" are plotted, for a varying number of joins. The figure shows that choice of starting keyword for CN generation greatly impacts performance.

As in [1, 10] we find that choosing the appropriate starting keyword can make a tremendous difference in the number of CNs to be explored given that most KWS systems use BFS for CN generation. As an example, for the query "dewitt gamma" starting with the keyword "dewitt" that maps to the *Person* relation greatly increases the amount of work involved in CN generation. This is because the *Person* relation has key-foreign key relationships with 9 other tables in the DBLife schema. Instead, starting with the keyword "gamma" that maps to the *Publications* relation and has only one edge is more advisable. Determining which keyword to start from depends heavily on the underlying schema. For instance, starting with a keyword corresponding to a fact table in a snowflake schema is not advisable.

We use the following heuristic to choose the starting keyword: we start with the set of keywords entered by the user and map each of them to the relations to which they belong. Using the undirected version of the schema graph of the underlying database, for every keyword k_i we pick the relation with the maximum degree d_i. Here d_i is the maximum number of edges that the keyword k_i could generate. We then pick the keyword with the minimum d_i value as our starting keyword. Clearly it is possible to have a schema for which the above algorithm is not very useful. Any other algorithm for choosing the starting keyword may be chosen and plugged-in as appropriate.

Figure 13 demonstrates the effect of choosing the right starting keyword for the 6 permutations of the query "hristidis keyword search" in a KWS system. We fix the number of M_{max} for the KWS approach at 6. Since the *Person* table has the maximum fanout in the DBLife schema, starting with a person name as the first keyword leads to worse performance than in the other 4 cases.

D. EXPERIMENTS WITH THE EASE SYSTEM

Figure 14 shows the run times for EASE for the 10 keyword queries in Figure 8.

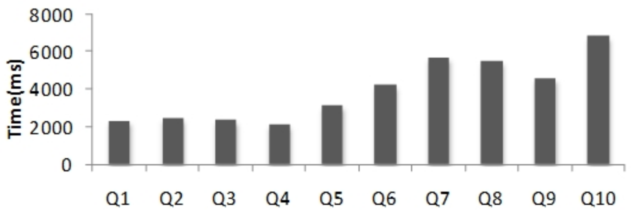


Figure 14: EASE leverages offline computation to achieve good on-line query performance.