On Debugging Non-Answers in Keyword Search Systems

Akanksha Baid Wentao Wu Chong Sun AnHai Doan Jeffrey F. Naughton
Department of Computer Sciences, University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI, USA
{baid, wentaowu, sunchong, anhai, naughton}@cs.wisc.edu

ABSTRACT

Handling non-answers is desirable in information retrieval systems. Current e-commerce websites usually try to suppress the somewhat dreaded message that no results have been found. Possible solutions include, for example, augmenting the data with synonyms and common misspellings based on query logs. Nonetheless, this is only achievable if we can know the cause of the non-answers. Under the hood, most e-commerce data sits in some *structured* format. Debugging non-answers in the underlying KWS-S systems is therefore not trivial — non-answers in a KWS-S system could be a problem of the data (e.g., absence of some keywords), the schema (e.g., missing key-foreign-key joins), or due to empty join results from one of possibly several joins in the generated SQL queries. So far, we are unaware of any previous work that explores how to enable developers to debug non-answers in a KWS-S system. In this paper, we take a first step towards this direction by proposing a KWS-S system that can expose non-answers to the developers. Our system presents the developers with the maximal nonempty sub-queries that represent the frontier cause of the non-answers. We outline the challenges in building such a system and propose a lattice structure for efficient exploration of the non-answer query space. We also evaluate our proposed mechanisms over a real world dataset to demonstrate their feasibility.

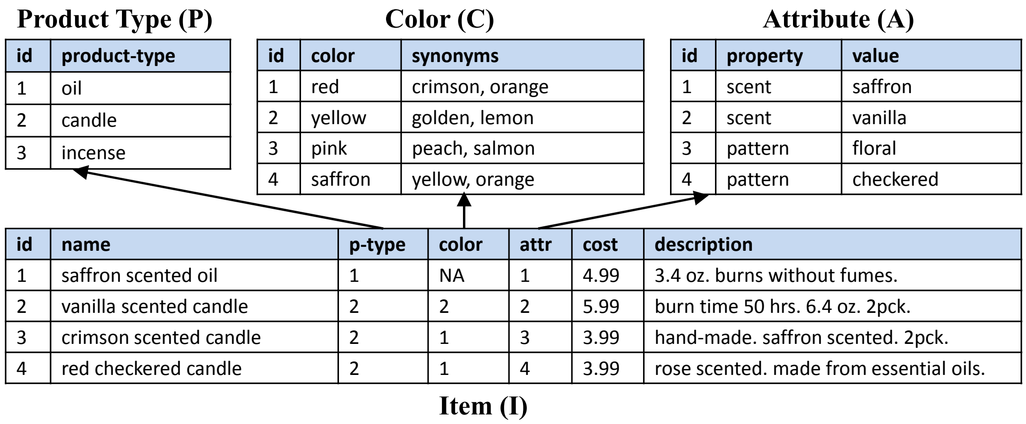
1. INTRODUCTION

Handling non-answers (i.e., queries that return no results) is now a common practice in information retrieval systems. Current SEO companies and e-commerce websites like Orcale Endeca [21], HP Autonomy [9], and IBM Coremetrics [13] often try to avoid showing the somewhat dreaded "*No results found*!" message when they fail to return any results that can match user's keyword queries. Possible strategies include, for instance, substituting user's original keywords with different keywords from a controlled vocabulary (e.g., synonyms, hyponyms, and hypernyms), or displaying a "*Did you mean*?" style response with spelling corrections. Doing so is critical to helping customers find what they are looking for and improving user experience and ultimately retention.

Nonetheless, implementation of such seemingly simple strategies is not trivial. While users interact with a search box, under the hood most e-commerce data sits in some *structured* format, largely due to maintainablity reasons. To employ strategies such as augmenting the data with synonyms and common misspellings based on query logs, we first need to understand the cause of the non-answers. For example, we need to convince ourselves that a non-answer query is really caused by missing keywords in the data before we decide to add the missing words into the vocabulary; otherwise this action is not helpful. Unfortunately, debugging non-answers in the underlying KWS-S (acronym for KeyWord Search over Structured data) systems is challenging — non-answers in a KWS-S system could be a problem of the data (e.g., absence of some keywords), the schema (e.g., missing key-foreign-key joins), or due to empty join results from one of possibly several joins in the generated SQL queries. So far, we are unaware of any previous work that explores how to enable developers to debug non-answers in a KWS-S system.

In this paper, we take a first step towards systematically exploring non-answers in KWS-S systems, and we focus on seeking the *maximal* partial matches or sub-queries of the non-answers. Similar ideas have been explored in the *unstructured* world. For example, Figure 1 presents the screenshot from buy.com in response to the keyword query "`saffron scented candle`". Although no saffron-scented candles are found, rather than displaying a blank page that shows no results, other saffron-scented products and other scented candles are presented to the user, corresponding to the three sub-queries "`saffron scented`", "`saffron candle`", and "`scented candle`". We believe that this kind of information could also be very helpful for debugging non-answers in KWS-S systems, and our primary goal in this paper, akin to what has been done over unstructured data, is to find results from sub-queries of non-answers, but over *structured* data.

Moving from unstructured to structured data, however, is more complicated than we might have thought. In typical KWS-S systems such as Banks [1, 14], DBXplorer [2], and DISCOVER [11], users enter a set of keywords and the system responds with a multitude of relationships connecting those keywords. In [2, 11, 17, 19] and many other KWS-S systems, this is done by mapping the keyword query to several structured queries (i.e., SQL queries). All of these structured queries are then evaluated, and the tuples corresponding to the queries that produce answers are returned to the user. Sub-queries of a non-answer query in a KWS-S system thus refer to sub-queries of a structured SQL query rather than the original keyword query. Since a KWS-S system can usually generate many SQL queries in response to a single keyword query, naively evaluating all possible sub-queries at runtime could be quite expensive. To efficiently explore the space of sub-queries, our basic idea is to exploit the common sub-queries that are shared by



Product Type (P)

id	product-type
1	oil
2	candle
3	incense

Color (C)

id	color	synonyms
1	red	crimson, orange
2	yellow	golden, lemon
3	pink	peach, salmon
4	saffron	yellow, orange

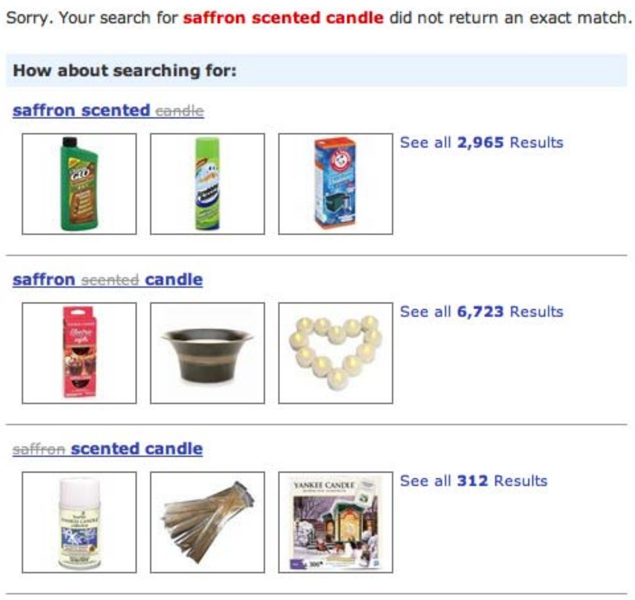
Attribute (A)

id	property	value
1	scent	saffron
2	scent	vanilla
3	pattern	floral
4	pattern	checkered

id	name	p-type	color	attr	cost	description
1	saffron scented oil	1	NA	1	4.99	3.4 oz. burns without fumes.
2	vanilla scented candle	2	2	2	5.99	burn time 50 hrs. 6.4 oz. 2pck.
3	crimson scented candle	2	1	3	3.99	hand-made. saffron scented. 2pck.
4	red checkered candle	2	1	4	3.99	rose scented. made from essential oils.

Item (I)

Figure 2: Product database containing an `Items` Table (I), `Product Type` table (P), `Color` table (C) and `Attributes` table (A).



Sorry. Your search for **saffron scented candle** did not return an exact match.

How about searching for:

saffron scented ~~candle~~

See all **2,965** Results

saffron ~~scented~~ candle

See all **6,723** Results

~~saffron~~ scented candle

See all **312** Results

Figure 1: Screenshot from buy.com where sub-queries and their results are suggested to the user when "saffron scented candles" returns zero products.

multiple structured SQL queries. To put things in context, let us consider the following example:

EXAMPLE 1 (NON-ANSWERS IN KWS-S). *Figure 2 shows a toy database that will be used throughout this paper. It contains an* `Items` *table I, a* `Product Type` *table P, a* `Colors` *table C, and an* `Attributes` *table A. The arrows here present the key-foreign-key associations between the tables.*

Consider the keyword query "`saffron scented candle`*". The KWS-S system maps it to two structured SQL queries (R^k here means the keyword k is mapped to the table R):*

(q_1) $P^{candle} \bowtie I^{scented} \bowtie C^{saffron}$, *which tries to "find scented candles whose color is saffron."*

(q_2) $P^{candle} \bowtie I^{scented} \bowtie A^{saffron}$, *which tries to "find scented candles whose scent is saffron."*

Both q_1 and q_2 return no result tuples with the given database, namely, they are non-answers. In the case of q_1, while every keyword does occur in the database, the join of all the involved tables produces no results. Exposing this information and q_1 can allow the developer or SEO person to add `saffron` *as a synonym of* `yellow`*, thus returning several relevant results to the user. In existing KWS-S systems q_1 would never be exposed.*

As for q_2, its sub-queries, which are $P^{candle} \bowtie I^{scented}$ and $I^{scented} \bowtie A^{saffron}$, do return answers, even though q_2 does not. More specifically, while the merchant does not carry any saffron-scented candles, it does carry scented candles and saffron-scented products that are not candles. Knowing this information may not help return answers to the original query, like in the q_1 case. However, it could be useful for merchandizing purposes. Additionally (like in Figure 1), the partial queries serve as a good alternative to returning nothing.[1]

Inspired by Chapman and Jagadish [5], to explain causes of the non-answers, we report their *maximal* sub-queries that return at least one tuple. To illustrate, in Example 1, our system will display $P^{candle} \bowtie I^{scented}$ and $C^{saffron}$ for q_1, and $P^{candle} \bowtie I^{scented}$ and $I^{scented} \bowtie A^{saffron}$ for q_2. Intuitively, these sub-queries sit on the *boundary* of answers/non-answers and provide the developer with information about the *frontier* causes of the non-answers. Similar notions have been proposed in previous work as solutions to "why not" style questions. For instance, in [5] the authors proposed using *frontier picky manipulations* which are the highest operators in a query tree or the latest manipulations in a workflow that rule out data items interested by the user from the results.

To find the maximal nonempty sub-queries for the non-answers, a naive strategy could be to enumerate all sub-queries and evaluate them (i.e., run the SQL query over the database) to check if they are empty. This is clearly inefficient. Our key observation here is that sub-queries of the non-answers overlap. For example, the q_1 and q_2 in Example 1 share the common join query $P^{candle} \bowtie I^{scented}$. In our experiments, we found that this overlap is significant on real data. Motivated by this observation, we propose a lattice structure that represents all the structured queries that a KWS-S system explores (details in Section 2). This structure is constructed offline and is used to capture the overlap between the sub-queries of each query (Section 2.2). Additionally, it also lends itself to systematic exploration of the sub-queries of non-answer queries. Note that, once we know the status of a query (i.e., if it is empty),

[1] The situation here is a bit more symmetric than that described in this example: given another instance of the tables, it might be q_2 that can be fixed via a synonym, whereas q_1 might be the query where non-answers are explained via maximal sub-queries.

this information can be utilized to determine the status of other queries based on the hierarchical relationships between queries presented in the lattice. For instance, in Example 1, we do not need to run the two SQL queries corresponding to P^{candle} and $I^{scented}$ once we know $P^{candle} \bowtie I^{scented}$ is nonempty — they must both be nonempty as well. This raises the interesting question of in which order we should visit the nodes in the lattice with the purpose of minimizing the number of SQL queries that need to be executed, which is the key to runtime system performance. We studied both *top-down* and *bottom-up* strategies to traverse the lattice structure (Section 2.5), and found that their performance depends on the distribution of the non-answer sub-queries within the hierarchy. Specifically, top-down/bottom-up strategies are more efficient when the maximal nonempty sub-queries are at higher/lower levels of the lattice. With this in mind, we further propose a greedy algorithm based on a scoring function that measures the potential reduction in the search space from examining a certain sub-query (Section 2.5.3). Our experimental results show that, while top-down and bottom-up strategies suffer from certain distributions of the non-answers, the greedy algorithm can perform relatively well in all the cases we tested.

While our proposed framework can efficiently find all the maximal non-empty sub-queries for non-answers, in our experiments we observed that the number of sub-queries is sometimes large. This is actually an inherent problem of KWS-S systems. Existing KWS-S systems usually use ranking functions to present users with only the most relevant results. For instance, Hristidis et al. [10] studied the problem of efficiently presenting the end users with a list of top-k matches. However, such strategies cannot work for the goal of debugging non-answers in KWS-S systems. This is simply because of the nature of debugging, which needs to find the cause of the non-answers no matter how trivial the cause might be. It is akin to debugging a normal computer program, where *all* possible bugs should be reported. Of course, there is a number of possible solutions to alleviate the problem of overwhelming number of sub-queries. For example, one option could be to allow the developer to define various filters or a priority hierarchy on the returned sub-queries. We do not try to explore all of these possible postprocessing techniques in this paper, most of which are application-specific and therefore may not have a uniformly optimal solution. Rather, we focus on an essential foundational task that must be solved before any higher-level postprocessing can be performed: the task of efficiently finding non-answers in response to a keyword query over structured data. It is our hope that our solution for this task provides a building block that can be used in conjunction with future research to build more customized systems.

In the rest of the paper, we start by introducing the system architecture of the proposed system and detailing its components in Section 2. We then evaluate our proposed approaches in Section 3. We discuss related work in Section 4 and conclude in Section 5.

2. EXPLORING NON-ANSWERS

In this section we describe our proposed solution for efficiently determining and explaining non-answer queries. Figure 3 presents the proposed system in its entirety. **Phase 0** is performed offline. In this phase, based on the schema graph of the underlying database, we generate a lattice in which each node is labeled with an uninstantiated SQL query. This structure is designed to exploit the overlap between the queries that are explored by our system. Following this, in **Phase 1** user's keyword query is accepted and used to prune the lattice generated in **Phase 0**. At the end of **Phase 1** each node in the pruned lattice is labeled with an instantiated SQL query with respect to the keyword query. In **Phase 2** we prune the lattice even further by retaining only those nodes that contain answer queries or non-answer queries, with respect to the current keyword query, and their respective descendants. Finally, we traverse the pruned lattice to determine and explain non-answer queries in **Phase 3**. Based on the information obtained, the user can subsequently choose to modify the keyword query as needed. We start by providing a formal problem definition describing the input and output of the proposed system.

2.1 Problem Definition

The input to our system is the keyword query submitted by the user. We convert the keyword query to join networks of tuple sets and candidate networks (see DISCOVER [11] for definitions).

The output of our system contains three parts: (i) answer queries, i.e., candidate networks that return at least one tuple; (ii) non-answer queries, i.e., candidate networks that return no tuples; (iii) additionally, for non-answer queries, we return the *maximal* sub-networks (i.e., subgraphs) that return at least one tuple. This is analogous to the queries in Figure 1, and is meant to provide some insight into the reasons behind the non-answer queries.

More formally, let J be a join network of tuple sets (JNTS) of a keyword query K. Let $q(J)$ be the SQL query corresponding to J and $R(J)$ be the result set of tuples obtained by executing $q(J)$.

Let $\mathcal{C}(K)$ be the set of candidate networks (CNs) generated for K. For each $C \in \mathcal{C}(K)$, we say that C is an *answer query* of K if $R(C) \neq \emptyset$. Otherwise C is a *non-answer query*. We denote $\mathcal{A}(K)$ and $\mathcal{N}(K)$ as the sets of answer and non-answer queries of K.

For each $C \in \mathcal{N}(K)$, let $\mathcal{S}(C)$ be the set of JNTSs that are sub-networks of C. A $J \in \mathcal{S}(C)$ is said to be *maximal* if: (i) $R(J) \neq \emptyset$ and (ii) there is no $J' \in \mathcal{S}(C)$ s.t. J is a sub-network of J' and $R(J') \neq \emptyset$. We use $\mathcal{M}(C)$ to denote the *maximal* JNTSs in $\mathcal{S}(C)$ and $\mathcal{M}(K)$ to denote the set of maximal JNTSs for all the non-answer queries, i.e., $\mathcal{M}(K) = \bigcup_{C \in \mathcal{N}(K)} \mathcal{M}(C)$.

With the above definitions and notation, we can now formally define the input and output of our system as follows:

- **Input**: An unstructured keyword query K.
- **Output**: $\mathcal{O}(K) = \mathcal{A}(K) \bigcup \mathcal{N}(K) \bigcup \mathcal{M}(K)$.

2.2 Offline Lattice Generation (Phase 0)

Phase 0 of the system is performed offline. In this phase we generate a lattice-structure that serves as the starting point for every keyword query. The goal of this structure is to capture all the queries that a KWS-S system explores (i.e., join-queries that contain no projections). Each node in the lattice is labeled with the SQL query corresponding to the node. The base level nodes of the lattice contain the simplest queries — single table queries, one for each table. The next level is generated by joining in tables to each single-table query (avoiding cross products and using the joins that are implicit in the schema graph), and so forth.

Our goal is to cover all queries with up to m joins. Since each relation can appear many times in a single query, we maintain copies $R_1 \ldots R_{m+1}$ of each relation R. In this way we know we can generate all possible m-join queries (including the extreme case where a m-join query contains $m + 1$ instances of the same relation). In addition to this we also maintain a copy R_0 of every relation R in the database (explained in the next section).

EXAMPLE 2 (LATTICE). *Consider a database with only two relations $R(a, b)$ and $S(c, d)$. Assume that $m = 1$, namely, we allow only one key-foreign-key join $R.b \bowtie S.c$. As a result, the lattice contains two (i.e., $m + 1 = 1 + 1 = 2$) copies for each relation (except for the special R_0 and S_0).*

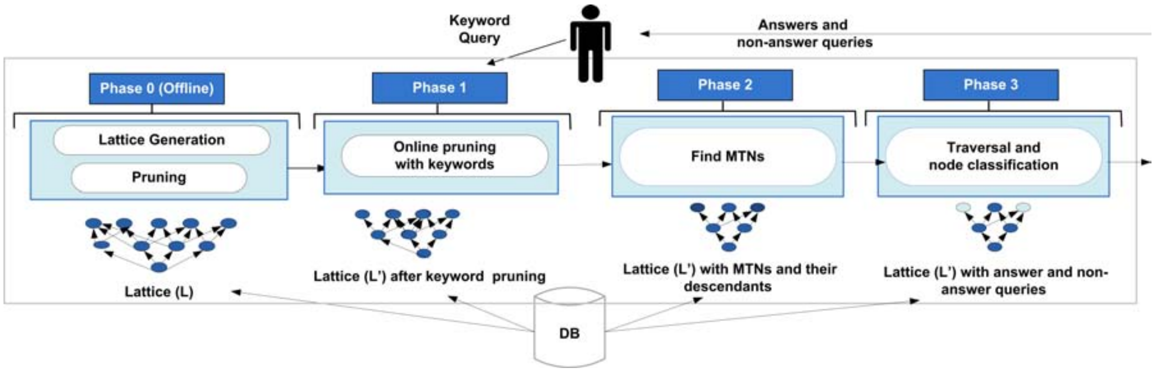


Figure 3: System Architecture for the proposed system. (MTN means Minimal-Total Node: see Section 2.4.)

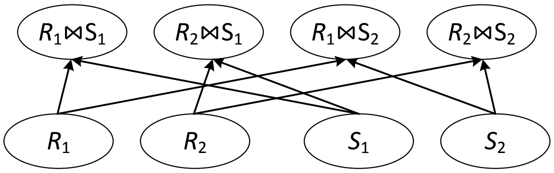


Figure 4: Example lattice with two relations $R(a, b)$ and $S(c, d)$, and a schema graph containing only one key-foreign-key join $R.b \bowtie S.c$.

Let “k_1 k_2” be the user’s keyword query. As shown in Figure 4, the generated lattice has two levels. Additionally, each node in the lattice is bound to a SQL query template. For instance, the node $R_1 \bowtie S_2$ corresponds to the template

```
SELECT * FROM R1, S2 WHERE R1.b = S2.c
AND R1.a LIKE '%k1%' AND S2.d LIKE '%k2%'.
```

Note that, the copies here are just conceptual symbols rather than physical replicas. The purpose of introducing the copies is to maintain a 1-1 mapping between lattice nodes and SQL query templates, which reduces the run-time query processing overhead. If, on the other hand, no additional copies were maintained, then the lattice in Figure 4 can be reduced to containing only three nodes R, S, and $R \bowtie S$. While this could reduce the storage overhead, each node in the lattice would correspond to multiple SQL query templates, and the parent-child relationships between the nodes would need to be reconstructed at run-time. This would adversely impact the time required to process keyword queries.

Furthermore, if a node N in the lattice is a descendant of a node N', then the query in N is a sub-query of the query in N'. The lattice structure hence organizes the queries and sub-queries that in a hierarchical fashion. As we will see, this structure has three primary advantages — (i) it allows reuse of evaluated queries; (ii) sometimes, it allows us to infer the outcome of a SQL query without executing it; and (iii) its hierarchical structure allows us to systematically explore sub-queries, which can be used to better understand non-answer queries. Also, since this structure is computed offline, it bypasses the costly candidate network generation phase, which is a part of traditional KWS-S systems.

While offline processing allows us to generate all the combinations of join-queries without taking a performance hit at run-time, this process leaves us with many duplicates. The duplicates are due to the fact that a node in the lattice can be obtained by different extensions of its children. For example, in Figure 4, the node $R_1 \bowtie S_2$ can be obtained by either extending the node R_1 (joining it with S_2) or extending the node S_2 (joining it with R_1). We therefore need to eliminate as many duplicates as possible offline, to avoid expensive graph isomorphism tests at run-time. Eliminating duplicate nodes also helps with reuse. We talk more about reuse in Section 2.5.2.

Algorithm 1: Lattice Generation

```
1  Input: $\mathcal{R} = \{R_1, R_2, ..., R_n\}$, set of instance relations; $SG$,
   schema graph; $maxJoins$, max number of joins
2  Output: $\mathcal{L}$, lattice
3  // Generate the base level $\mathcal{L}_1$
4  $\mathcal{L}_1 \leftarrow \emptyset$
5  foreach $R_i \in \mathcal{R}$ do
6      for $1 \le j \le maxJoins + 1$ do
7          $\mathcal{L}_1 \leftarrow \mathcal{L}_1 \cup \{CreateSingleNodeGraph(R_i)\}$
8
9  // Generate higher levels $\mathcal{L}_k$, for $2 \le k \le maxJoins + 1$
10 foreach $2 \le k \le maxJoins + 1$ do
11     $\mathcal{L}_k \leftarrow \emptyset$
12     foreach Graph $G \in \mathcal{L}_{k-1}$ do
13         foreach $R \in Nodes(G)$ do
14             $\mathcal{G}' \leftarrow ExtendGraph(R, G, SG)$
15             foreach $G' \in \mathcal{G}'$ do
16                 // Offline Pruning 1: detect duplicates
17                 if $G' \notin \mathcal{L}_k$ then
18                     $\mathcal{L}_k \leftarrow \mathcal{L}_k \cup \{G'\}$
19 return $\mathcal{L}$
```

The details of the lattice generation algorithm are presented in Algorithm 1. It works as follows. We first create $maxJoins + 1$ copies for each input instance relation (lines 5 to 7). These constitute the bottom level of the lattice. We then construct the upper levels (lines 9 to 18). When generating the graphs at the level k (i.e., $\mathcal{L}_k$ for $2 \le k \le maxJoins + 1$), we check each graph G at the level $k - 1$ (i.e., $\mathcal{L}_{k-1}$). For each relation R in G, we look up the schema graph SG to find possible edges that are connected with R. Whenever we find such an edge $e = (R, R')$, for each copy R'_c of R', we create a new graph G' by first copying G and then inserting the edge (R, R'_c) into G'. This is done by calling the function $ExtendGraph$ (line 14). For each such extension G', we then check whether $G' \in \mathcal{L}_k$. If not, G' is added into $\mathcal{L}_k$ (lines 15 to 18). Note that here, to detect the duplicates, we need to test

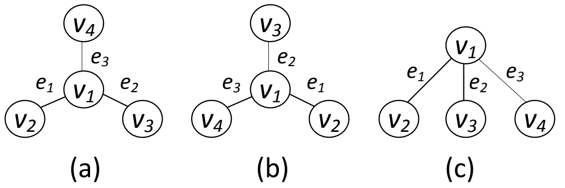


Figure 5: Two isomorphic trees and their canonical form.

the isomorphism between graphs, which is a problem not known as either P or NP-complete. However, since G' is a candidate join-query network, which by definition must be a tree [11], there are efficient algorithms (in linear time) for this special case. We use a variant of the algorithm in [3], by computing a *canonical labeling* for each graph (tree). Two graphs (trees) are isomorphic if and only if they have the same canonical labeling.

Specifically, given a candidate join-query network (tree) T, let $V(T)$ and $E(T)$ be its nodes and edges. For any $v \in V(T)$, the label of v is defined as the relation name R_i associated with v. For any $e \in E(T)$, the label of e is defined as $(R_i.a, S_j.b)$, where $R_i.a \bowtie S_j.b$ is the join associated with e. We further map the labels to integer ID's. Let $id(v)$ and $id(e)$ be the ID's assigned to a node v and an edge e. We compute the canonical labeling of T as shown in Algorithm 2. Example 3 illustrates this.

Algorithm 2: Canonical Labeling

```
1  Input: $T$, a candidate join-query network
2  Output: $l_T$, canonical labeling of $T$
3  GetCode($u$):
4  $l \leftarrow$“[$id(u)$”
5  if $HasChildren(u)$ then
6      $l.Append$(“|”)
7      foreach $v \in Children(u)$ do
8          $l(v) \leftarrow$“$id(e)GetCode(v)$” // $e = (u, v)$
9      Sort $v \in Children(u)$ with respect to $l(v)$
10     foreach $v \in Children(u)$ do
11         $l.Append(l(v))$
12 $l.Append$(“]”)
13 return $l$
14
15 Main:
16 $\mathcal{R} \leftarrow \{r | r = \arg\min_v \{id(v) | v \in V(T)\}\}$
17 $L_{\mathcal{R}} \leftarrow \{l_r | l_r \leftarrow getCode(r), r \in \mathcal{R}\}$
18 $l_T \leftarrow \min\{l_r | l_r \in L_{\mathcal{R}}\}$
19 return $l_T$
```

EXAMPLE 3 (CANONICAL LABELING). *Figure 5(a) and (b) show two isomorphic trees. Their canonical form is shown in (c). The corresponding canonical labeling computed by Algorithm 2 is:* $[v_1|e_1[v_2]e_2[v_3]e_3[v_4]]$.

In Algorithm 2, we first define $\mathcal{R}$ to be the set of nodes with the minimum node ID (line 16), and then call $GetCode$ to compute the labeling l_r for each $r \in \mathcal{R}$ (line 17). The minimum l_r in lexicographic order is the canonical labeling for T (line 18). The procedure $GetCode$ (lines 3 to 13) first puts the ID of the current node u into the labeling. If u has children, it appends a delimiter “|”, and then recursively calls $GetNode$ to construct the label $l(v)$ of each child node v (lines 7 to 8). The label of v is appended with respect to the ordering of $l(v)$ (lines 9 to 11).

Once the lattice is generated and duplicates are removed, each node is labeled with a SQL query corresponding to the node. Since Phase 0 is performed offline, the SQL query in each node has an uninstantiated “where” clause. More specifically, the join conditions (e.g., `Item`.cid = `Color`.id) in the “where” clause are present, but the keywords (e.g., `Color`.name contains “saffron” OR `Color`.synonym contains “saffron”) can be added to it only at run-time, once user's keyword query is available. Keyword query is accepted in the next phase.

2.3 Keyword Based Pruning (Phase 1)

Once the user inputs the keyword query K, we *map* each keyword to a relation using an inverted index over the data. A keyword k_i can be *mapped to* a relation R if k_i occurs in some tuple in R. Recall that we generated copies $R_1 \ldots R_{m+1}$ for each relation R in the database. If k_i maps to R, k_i is bound to one of the copies R_j of relation R. Additionally, we bind the *empty keyword* to the copy R_0 for each relation R in the database.

We do this because relations to which no keywords are bound can still contribute to valid relationships. For example, for the keyword query “red candle”, suppose that “red” is bound to the `Color` (C) table and “candle” is bound to the `Product Type` (P) table. While these two tables cannot be directly joined due to the lack of a key-foreign-key association, the `Items` (I) table can be used to form a path between them. The resulting join network is then $C_1 I_0 P_1$, which represents the query “Find all products where product type is *candle* and color is *red*”. The I_0 here is used to indicate that no keyword is bound to the `Items` table. This is analogous to a *free tupleset* in Discover [11].

At the base level, all the nodes that contain queries with copies of relations to which no keyword is bound are pruned. Their respective ancestors are also pruned. For the keyword query “red candle”, a sample lattice with the `Item` (I), `Color` (C), and `Product Type` (P) relations from the sample database in Figure 2 is presented in Figure 6. Upon using the inverted index “red” is bound to C_1 and “candle” to P_1. C_0, I_0 and P_0 are bound to the *empty keyword* and are not pruned. Only the shaded nodes in the lattice are retained. The remaining nodes are pruned.

In our implementation, we handle cases where keywords can have multiple interpretations by dealing with one interpretation at a time. Additionally, if a keyword does not occur anywhere in the database, the system displays all such keyword(s) and does not investigate the query any further. This is in accordance with “and” semantics for keyword search.

At the end of Phase 0, each node is labeled with a SQL query with an uninstantiated “where” clause. Once we bind keywords to copies of relations, the “where” clauses of the queries in each remaining node in the lattice can then be instantiated. We now have an instantiated, pruned lattice for the keyword query K.

2.4 Finding Answer and Non-Answer Query Nodes (Phase 2)

Once the lattice has been pruned based on keyword query K, the next step is to find nodes that contain queries that correspond to answer queries and non-answer queries. To do this, first we introduce some terminology.

- *Total/Partial Node*: A node can be total or partial. A node N is said to be total if its query contains tables corresponding to every keyword k_i in K. Otherwise N is said to be partial. Since we assume “and” semantics for keyword search, only a total node can contain an answer query. [2]

[2] We further note that totality decreases by moving down in the

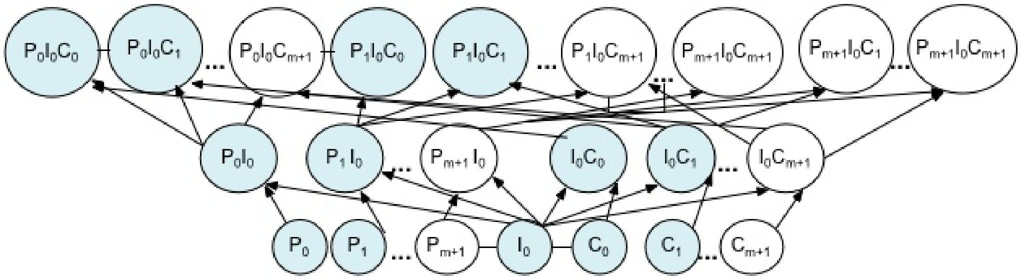


Figure 6: Sample lattice for the query "red candle". The un-shaded nodes are pruned.

- *Alive/Dead Node*: A node N is said to be alive if its query returns at least one tuple upon execution. If the query returns zero tuples, N is said to be dead. Typically a node can be classified as dead or alive only after executing its underlying structured query. As we shall see later in this section, in many cases, using the lattice structure helps us classify a node without actually executing its query.

- *Possibly Alive Node*: This node has not yet been classified as dead or alive. In the beginning, all the nodes in the pruned lattice are possibly alive.

- *Minimal-Total Node (MTN)*: A node N is said to be minimal-total, if N is total and no descendant of N is total. MTNs correspond to candidate networks in KWS-S systems [11], and contain answer and non-answer queries.

In Phase 2, we prune the lattice even further by only retaining MTNs and their descendants. To continue with our example, the node marked $P_1I_0C_1$ is the only MTN in the lattice in Figure 6. (None of the other shaded nodes are total.) We are now left with the task of classifying MTNs as dead or alive and explaining the reason(s) for the dead MTNs. We do this using Maximal Partial Alive Nodes (MPANs).

- *Maximal Partially Alive Node (MPAN)*: A node N is said to be a MPAN of a MTN M if it is both partial and alive, and if there exists no other node $N' \in \mathcal{D}esc(M)$ such that $N \in \mathcal{D}esc(N')$ and N' is alive.

There can be multiple reasons for a non-answer query. For example, the SQL query q_2 in Example 1 for the keyword query "saffron scented candle", where saffron is a scent, could be a non-answer query due to several reasons — the store carries products that are saffron scented but are not candles, they only carry unscented candles, they carry scented candles but none of them are saffron-scented or maybe they only carry products that are neither saffron-scented nor candles. The options that an administrator needs to explore in order to determine why q_2 is a non-answer query can get dauntingly large for manual debugging. Given that each keyword may have multiple interpretations (e.g., saffron could be a color or a scent), this task gets even more daunting.

We display the maximal alive query because we know that all its descendants are alive (i.e., if "find scented candles" returns some result tuples, then both "find scented products" and "find candles" will also return some result tuples). In Example 1, "find scented candles" and "find saffron scented products" are both MPANs of q_2. Collectively they convey that while the store does not carry saffron-scented candles, it does carry scented candles and other saffron-scented products. Notably, since all possible reasons for a non-answer query are sub-queries of the non-answer query, they can be systematically explored using our proposed lattice structure.

In the final phase of our system and in the rest of this section we discuss lattice traversal strategies to efficiently determine dead MTNs and their respective MPANs.

2.5 Lattice Traversal (Phase 3)

One approach to classifying the MTNs as dead or alive and finding MPANs is to simply execute the SQL queries for all the nodes in $\mathcal{L}$. However, this may not be necessary. Since each MTN is derived from its descendants, we can use the following two rules to avoid executing many of the SQL queries in $\mathcal{L}$.

Node Classification Rules:
• **(R1)** Node N is alive $\Rightarrow$ All $\mathcal{D}esc(N)$ are alive.
• **(R2)** If any $N' \in \mathcal{D}esc(N)$ is dead $\Rightarrow$ N is dead.

The descendants of a node in the lattice represent sub-queries of the node. R1 says that if a node is alive, all its sub-queries should also return some tuples when executed. R2 says that if a node is dead, all the queries of which it is a sub-query will also return no tuples. Next, utilizing the above two rules we propose traversal strategies that classify MTNs and find MPANs in the lattice.

2.5.1 Bottom-Up/Top-Down Traversal

In the bottom-up (BU) strategy, we classify one MTN at a time and traverse the sub-lattice consisting of the MTN and its descendants from the single-table level up. At each level we evaluate the SQL query corresponding to each node. If a node N is dead (i.e., its SQL query returns no result tuples), all the nodes in $\mathcal{A}sc(N)$, including the MTN, can be marked as dead (by R2). If a node is alive, it is marked as a potential MPAN until one of its ancestors is found to be alive. This process is repeated for all the MTNs until they are all classified as dead or alive and the corresponding MPANs are found.

This strategy performs well when the MTNs and MPANs corresponding to dead MTNs are found at lower levels of the lattice. In this case BU can avoid executing several expensive SQL queries at higher levels of the lattice. For keywords where MTNs and MPANs are found at higher levels, a better approach might be a top-down traversal of the lattice.

Top-down (TD) traversal is similar to BU, except that we traverse the sub-lattice for each MTN from its highest level down to the single-table level. We evaluate the SQL query corresponding to each node at each level. If a node N is alive, all the nodes in $\mathcal{D}esc(N)$ can be marked as alive (by R1). This is done till all the nodes in the lattice are classified and all the MPANs are found.

lattice, because the descendant sub-queries of a query q in general refer to fewer tables than q itself. This allows us to speak of *minimal* total nodes as in the following.

Algorithm 3: Bottom-Up with Reuse Approach

```
1  Input: $SG$, schema graph; $K = \{k_1, k_2, \ldots, k_N\}$, keyword query; $\mathcal{L}$, lattice; $M$, set of MTNs found during Phase 2
2  Output: $A$, set of alive MTNs; $D$, set of dead MTNs; $P$, set of corresponding MPANs for each dead MTN
3  GetBaseNodes($K$, $\mathcal{L}$):
4  $baseNodes \leftarrow \emptyset, nonKeywords \leftarrow \emptyset$
5  foreach $k \in K$ do
6      $T \leftarrow GetBaseTables(k)$
7      if $T == \emptyset$ then
8          $nonKeywords.add(k)$
9      else
10         $baseNodes.add(\mathcal{L}.GetBaseNodes(T))$
11     if $nonKeywords \neq \emptyset$ then
12         Display $nonKeywords$
13         $baseNodes \leftarrow \emptyset$
14 return $baseNodes$
15
16 Main:
17 $M' \leftarrow M, currLevel \leftarrow 1$
18 foreach $m \in M$ do
19     MP[$m$]← $GetDescendants(m)$ // potential MPANs
20 $B \leftarrow GetBaseNodes(K, \mathcal{L}), curr \leftarrow B, next \leftarrow \emptyset$
21 if $B == \emptyset$ then
22     return
23 while $currLevel \leq maxJoins + 1$ and $M' \neq \emptyset$ do
24     foreach $node \in curr$ do
25         $isAlive \leftarrow true$
26         if $node \notin B$ and $execSQL(node).nTuples == 0$ then
27             $isAlive \leftarrow false$
28         if $node \in M$ then
29             $M' \leftarrow M' - node$
30         if $isAlive == true$ and $node \notin M$ then
31             $next \leftarrow next \bigcup \mathcal{L}.GetParentNodes(node)$
32             foreach $m \in M$ do
33                 if $node \in$ MP[$m$] then
34                     $\mathcal{L}.RemoveAllDesc$(MP[$m$],$node$)
35         else if $isAlive == false$ then
36             $MarkAsDead(\mathcal{L}.GetAscNodes(node))$
37             if $node \notin M$ then
38                 foreach $m \in M$ do
39                     if $node \in$ MP[$m$] then
40                         $\mathcal{L}.RemoveAllAsc$(MP[$m$],$node$)
41                         $\mathcal{L}.Remove$(MP[$m$],$node$)
42             else
43                 $P[node]$← MP[$node$] // MPANs
44         $curr \leftarrow next, next \leftarrow \emptyset$
45     $currLevel \leftarrow currLevel + 1$
```

2.5.2 Bottom Up and Top Down with Reuse

We found that there is usually substantial overlap between the descendants of each MTN. Based on this observation, we modify BU and TD to process all the MTNs and their descendants simultaneously. We find that we can substantially reduce the redundancy in executing SQL queries corresponding to the common descendants of the MTNs. The corresponding algorithms are termed bottom-up with reuse (BUWR) and top-down with reuse (TDWR). The details of BUWR are presented in Algorithm 3. TDWR is very similar to BUWR so we do not elaborate on it any further.

Algorithm 3 works as follows. It first finds the descendants for the MTNs in M, which are all the potential MPANs (lines 18 to 19). It then traverses the lattice $\mathcal{L}$ in a bottom-up manner. For each keyword k, *GetBaseNodes* (lines 3 to 14) collects the base nodes (i.e., tables) in the lattice containing the keyword k. If some keyword is not contained by any base table, then this is reported to the user (lines 11 to 13) and no further exploration is needed (lines 21 to 22). Otherwise, answers and non-answers will be reported to the user by climbing up the lattice $\mathcal{L}$ (lines 23 to 45). For each node $node$ at the current level, the algorithm first checks its aliveness (if not known yet) by executing the SQL query associated with it (lines 25 to 27). If $node$ is alive, and it is not a MTN in M, then we can remove its descendants from candidate MPANs of each MTN $m \in M$, since they must be alive and cannot be MPANs because of the aliveness of $node$ (lines 30 to 34). On the other hand, if $node$ is dead, then all of its ancestors must be dead (line 36). If $node$ is not a MTN, once again we can remove $node$ and its ancestors from candidate MPANs of each MTN $m \in M$, since they cannot be MPANs because of their deadness (lines 37 to 41). Otherwise, $node$ is a dead MTN, and we need to report its MPANs (lines 42 to 43). Note that the MPANs must be those candidates that are still remaining in $\mathrm{MP}[node]$. This is because, due to the nature of bottom-up traversal, the aliveness of each member in $\mathrm{MP}[node]$ must have been either explicitly (by executing the corresponding SQL query) or implicitly (by using the two node classification rules R1 and R2) checked and hence known before $node$ is examined. After checking all the nodes at the current level, the algorithm can proceed to the next level (line 44). The next level only needs to include the parents of the alive nodes at the current level (line 31), since ancestors of dead nodes must also be dead and therefore can be excluded from further examination.

While we find that these approaches perform well in general, like any bottom-up or top-down approach, they suffer poor performance in certain cases. For example, TD will perform poorly if many MPANs are present at the lowest level in the lattice. On the contrary, BU will perform poorly if many MTNs at higher levels of the lattice are alive. In the rest of this section, we propose a score-based greedy approach, with the goal of avoiding the worst-case performance of both BU and TD.

2.5.3 Score Based Heuristic for Traversal

Given that the main advantage of the lattice structure is reuse amongst MTNs, the goal of this approach is to evaluate nodes in an opportunistic manner with the goal of minimizing the number of evaluated SQL queries. We do this by assigning a score to each unevaluated node in the lattice. This score indicates the amount of reduction in the search space that would result from evaluating this node. In other words, we evaluate the nodes that are most likely to influence the classification of other nodes first. Table 1 summarizes the notation used in the following discussion.

We start by investigating the effect that evaluating a node n_j in $\mathcal{L}$ has on the remaining nodes in the search space $\mathcal{S}(m_i)$ of each $m_i \in \mathcal{M}$. $\mathcal{S}(m_i)$ contains potential MPANs of m_i with unknown status. SQL queries might be required to determine aliveness of the nodes in $\mathcal{S}(m_i)$. Initially, $\mathcal{S}(m_i) = \mathcal{D}esc^{+}(m_i) = \{m_i\} \bigcup \mathcal{D}esc(m_i)$. Figure 7 demonstrates how n_j and m_i and their descendants could overlap. We next consider the cases when n_j is alive or dead.

- **If the current node n_j is alive** :

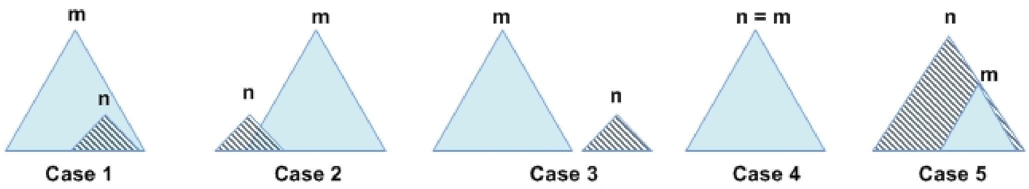


Figure 7: Ways in which a minimal complete node m, an unexplored node n, and their respective descendants may overlap.

Notation	Description
$\mathcal{N}$	$\mathcal{N} = \{n_1 \ldots n_q\}$, the set of nodes in the lattice $\mathcal{L}$
$\mathcal{M}$	$\mathcal{M} = \{m_1 \ldots m_p\}$, the set of MTNs
$\mathcal{P}(m_i)$	the set of MPANs for each dead $m_i \in \mathcal{M}$
$\mathcal{S}(m_i)$	the search space for each $m_i \in \mathcal{M}$ to find $\mathcal{P}(m_i)$
$\mathcal{D}esc(n)$	the set of descendants of the node n in $\mathcal{L}$
$\mathcal{A}sc(n)$	the set of ancestors of the node n in $\mathcal{L}$
$\mathcal{D}esc^+(n)$	$\mathcal{D}esc^+(n) = \{n\} \cup \mathcal{D}esc(n)$
$\mathcal{A}sc^+(n)$	$\mathcal{A}sc^+(n) = \{n\} \cup \mathcal{A}sc(n)$

Table 1: Notation used in the score-based heuristic

Case 1 ($n_j \in \mathcal{D}esc(m_i)$): If the current node n_j is a descendant of an MTN m_i, then all descendants of n_j are also alive. Since n_j is alive, these alive descendants cannot be maximal (i.e., cannot be in $\mathcal{P}(m_i)$), and thus can be removed from the search space $\mathcal{S}(m_i)$ as well.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \mathcal{D}esc^+(n_j).$$

Case 2 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) \neq \emptyset$ and $n_j \notin \mathcal{D}esc(m_i)$): Let n_k be the root node of the intersection $\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i)$. The descendants of n_k can also be removed from $\mathcal{S}(m_i)$ because they cannot be MPANs in $\mathcal{P}(m_i)$.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \mathcal{D}esc^+(n_k).$$

Case 3 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) = \emptyset$): The intersection of the descendants of n_j and m_i is empty. This implies that n_j has no impact on the search space of m_i.

Case 4 ($n_j = m_i$): Here n_j is the MTN. $\mathcal{S}(m_i) = \emptyset$.

Case 5 ($m_i \in \mathcal{D}esc(n_j)$): This case cannot occur because it implies that m_i is not minimal and hence not an MTN.

- **If the current node n_j is dead** :

Case 1 ($n_j \in \mathcal{D}esc(m_i)$): In this case, all nodes in $\mathcal{A}sc(n_j)$ are also dead and therefore can be removed from $\mathcal{S}(m_i)$.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \mathcal{A}sc^+(n_j).$$

Case 2 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) \neq \emptyset$ and $n_j \notin \mathcal{D}esc(m_i)$): No change in $\mathcal{S}(m_i)$.

Case 3 ($\mathcal{D}esc(n_j) \cap \mathcal{D}esc(m_i) = \emptyset$): No change in $\mathcal{S}(m_i)$.

Case 4 ($n_j = m_i$): m_i is the MTN.

$$\mathcal{S}(m_i) = \mathcal{S}(m_i) - \{n_j\}.$$

Case 5 ($m_i \in \mathcal{D}esc(n_j)$): This case cannot occur because it implies that m_i is not minimal and hence not an MTN.

We define $\mathcal{S}^a_{exp}(m_i)$ to be the expected search space of m_i if n_j is alive and $\mathcal{S}^d_{exp}(m_i)$ to be the expected search space of m_i if n_j is dead. Now, if p_a is the average probability that a node in the graph is alive, we define the score for n_j to be:

$$Score(n_j) = \sum_{m_i \in \mathcal{M}} p_a \cdot |\mathcal{S}^a_{exp}(m_i)| + (1-p_a) \cdot |\mathcal{S}^d_{exp}(m_i)|. \quad (1)$$

Intuitively, this score measures the expected size of the search space given the information that n_j is alive/dead.

We give some further analysis to the score so defined. Let

$$\mathcal{C}over(n) = \{n\} \cup \mathcal{D}esc(n) \cup \mathcal{A}sc(n)$$

be the *coverage* of a node n. We can then express $\mathcal{S}^a_{exp}(m_i)$ and $\mathcal{S}^d_{exp}(m_i)$ more explicitly:

$$\begin{aligned} \mathcal{S}^a_{exp}(m_i) &= \mathcal{D}esc^+(m_i) - \mathcal{D}esc^+(n_j) \\ &= (\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)) \\ &\cup \ (\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)) \end{aligned}$$

and

$$\begin{aligned} \mathcal{S}^d_{exp}(m_i) &= \mathcal{D}esc^+(m_i) - \mathcal{A}sc^+(n_j) \\ &= (\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)) \\ &\cup \ (\mathcal{D}esc^+(m_i) \cap \mathcal{D}esc(n_j)). \end{aligned}$$

Since $\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)$ and $\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)$ are disjoint, we have

$$\begin{aligned} |\mathcal{S}^a_{exp}(m_i)| &= |\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)| \\ &+ \ |\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)|, \end{aligned}$$

and similarly,

$$\begin{aligned} |\mathcal{S}^d_{exp}(m_i)| &= |\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)| \\ &+ \ |\mathcal{D}esc^+(m_i) \cap \mathcal{D}esc(n_j)|. \end{aligned}$$

Therefore, according to Equation (1), we have

$$\begin{aligned} Score(n_j) &= \sum_{m_i \in \mathcal{M}} |\mathcal{D}esc^+(m_i) - \mathcal{C}over(n_j)| \\ &+ \ p_a \cdot \sum_{m_i \in \mathcal{M}} |\mathcal{D}esc^+(m_i) \cap \mathcal{A}sc(n_j)| \\ &+ \ (1-p_a) \cdot \sum_{m_i \in \mathcal{M}} |\mathcal{D}esc^+(m_i) \cap \mathcal{D}esc(n_j)|. \end{aligned}$$

Based on the above equation, we can see that the score actually takes three factors into consideration:

(1) *the descendants of the MTNs that are not covered by n_j (the 1st summand)*: these nodes must be explored no matter whether n_j is alive or dead;

(2) *the descendants of the MTNs that are ancestors of n_j (the 2nd summand)*: these nodes are explored when n_j is alive;

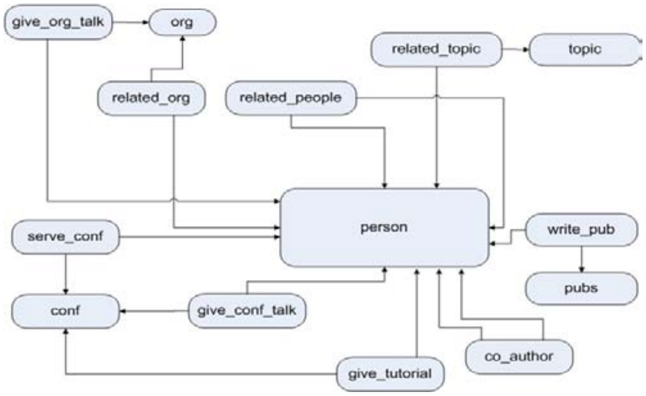


Figure 8: Relational schema for the DBLife dataset.

(3) *the descendants of the MTNs that are descendants of* n_j *(the 3rd summand)*: these nodes are explored when n_j is dead.

Hence, a smaller score means a smaller expected search space. We then use a simple greedy strategy based on this score to traverse the lattice — each time we pick the node n_j with the minimum score, check its aliveness, and eliminate other nodes from the lattice according to this information as before. This algorithm terminates when all nodes have been classified and the MPANs for all dead MTNs have been found.

The remaining issue is how to determine the probability p_a. We note here that setting p_a affects performance, not correctness. Estimating the probability value p_a accurately requires evaluating all the queries and finding out what percentage of them are empty. However, our experiments show that the simple assumption that p_a = 0.5 works surprisingly well. Nonetheless, it is still interesting future work to explore lightweight estimation approaches for p_a.

3. EVALUATION

In this section we evaluate our proposed approaches. We ran our experiments using PostgreSQL 8.3.6 on an Intel(R) Core(TM) 2 Duo 2.33 GHz system with 3GB of RAM. We implemented all the query processing algorithms in Java, and used JDBC to connect to the database. We further implemented the inverted indexes over the data using Lucene [18]. We evaluated the proposed approach over a 40MB snapshot of the DBLife [7] dataset that has 801,189 tuples in 14 tables (Figure 8). Note that in the DBLife schema, keywords are contained in 5 entity tables, namely, `Person`, `Publication`, `Conference`, `Organization`, and `Topic`. The 9 relationship tables connect the entity tables but do not contain any text attributes.

3.1 Lattice Generation

In Figure 9(a), we look at the number of nodes in the lattice. Having several copies of each table adds to the number of seed tables and consequently to the number of nodes in the lattice. As expected, the number of nodes grows exponentially as the level (and number of joins) increases. It is thus important to have efficient traversal and pruning strategies. Figure 9(a) shows the number of nodes generated and the number of duplicate nodes in the lattice. On average 11.7% of the generated nodes were removed due to duplicate elimination (note that the Y-axis is in log scale).

Next, in Figure 9(b), we look at the time spent in generating the lattice. We vary the level on the X-axis and measure the time taken on the Y-axis. We observe that lattice generation completes in less than 100 seconds, even for a lattice with 7 levels (i.e., 6 joins). We note that this is a one-time cost, and is done offline.

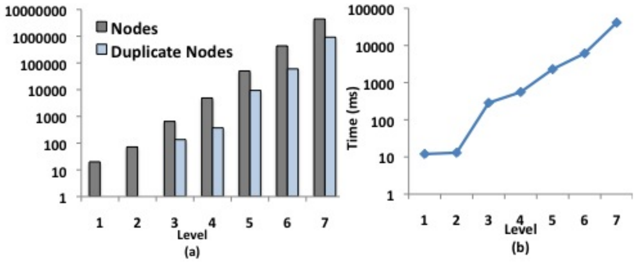


Figure 9: (a) The number of nodes generated in the lattice at each level and the number of duplicates are shown. As expected, the number of nodes in the lattice grows exponentially. On an average 11.7% of the nodes were pruned due to duplicate elimination. (b) The time spent in generating the lattice is shown. We note that the lattice is generated *offline*.

3.2 Keyword Queries

Table 2 lists the keyword queries we used in our following experiments. DBLife has a star schema, with the `Person` table at the center. As a result, queries with many person names (e.g., Q3) often lead to many MTNs. Q7, Q9, and Q10 do not contain any person names and Q8 contains the term "Washington" which occurs in the `Person`, `Publication`, and `Organization` tables. Q4 and Q6 lead to empty queries at the two-table level, but MTNs are found at higher levels as KWS-S explores relationships with more joins.

ID	Keyword Query
Q1	Widom Trio
Q2	Hristidis Keyword Search
Q3	Agrawal Chaudhuri Das
Q4	DeRose VLDB
Q5	Gray SIGMOD
Q6	DeWitt tutorial
Q7	Probabilistic Data
Q8	Probabilistic Data Washington
Q9	SIGMOD XML
Q10	Stream data histograms

Table 2: Keyword queries

3.3 Keyword Based Pruning (Phase 1) and Finding MTNs (Phase 2)

The next step involves mapping user's keyword query to schema terms and is performed online. This primarily involves lookups over the inverted indexes on the data. For the 10 testing queries, the time to map the keywords to schema terms varied between 7 ms and 66 ms with an average time of 26 ms.

Next, we measured the number of nodes in the lattice that remain in the lattice upon the introduction of keywords. We note that keyword-based pruning reduced the number of relevant nodes by 98% on average. Once the keywords are mapped to schema terms, the next step is to find the MTNs. This process took up to 23 ms for the 10 queries, in a lattice for level = 5. The number of MTNs ranged from 3 to 85.

Figure 10 summarizes these results. It also shows the number of unique descendants for the MTNs. We note that Q3 and Q8 have lower number of unique descendants, allowing higher possibility of reuse, as we will show later. We also computed these statistics for level = 7, and observed that the number of nodes after pruning varied from 277 to 18,904 with an average of 9,226 nodes, a reduction of 94.3% from the 161,440 nodes in the original lattice. The number of MTNs ranged from 35 to 1,418. This shows that even though a large number of lattice nodes are generated, phases 1 and 2 can prune the lattice substantially.

3.4 Comparison of Traversal Strategies

The goal of our traversal strategies is to efficiently determine if an MTN is alive or dead and to find the MPANs for the dead MTNs. We compared the five strategies for lattice traversal from end-to-end. Figure 11 shows the number of SQL queries that needed to be executed by each traversal approach for level = 5. Figure 12 shows the times taken by each approach for the corresponding queries. Note that both BUWR and TDWR perform better than their respective counterparts without reuse. This is especially true for Q3 and Q8, because for these queries, the total number of unique descendants are much smaller than that with duplicates. The number of SQL queries executed for Q3 is shown in Table 4. Q2 leads to only 3 MTNs, all of which are alive (i.e., Q2 has 0 MPANs). One of these 3 MTN queries is a join between the `Person` and `Publication` tables, with the keywords occurring in many tuples. This query takes around 20 seconds to execute. The proposed score-based heuristic (SBH) approach performs well in almost all cases, owing to its opportunistic choice of nodes during the pruning/traversal process. Further, we also note that TD and TDWR perform better than BU and BUWR respectively. Next, we explain the reason for the performance difference.

3.5 Impact of MTN and MPAN Distributions

For the DBLife dataset, we find that as the maximum level of the lattice increases, BU and BUWR generally perform poorly when compared to TD, TDWR, and SBH. This is because even keyword queries that return no answers at lower levels might return answers at higher levels. These answers correspond to relationships with many hops. Since the number of nodes in the lattice grows exponentially as the number of joins increases, TD and TDWR have a better pruning effect than BU and BUWR.

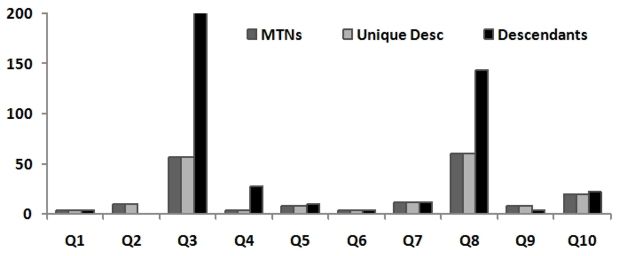


Figure 10: Keyword queries enable substantial pruning of the lattice (98% on an average). The number of MTNs and their descendants and unique descendants are presented to show the extent of overlap between the nodes in the lattice.

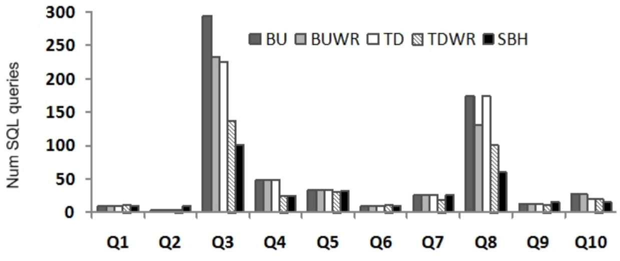


Figure 11: The number of SQL queries generated by the system for each keyword query.

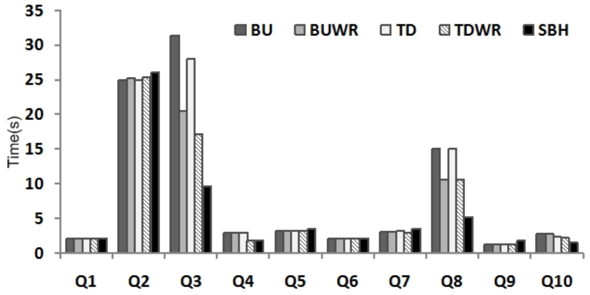


Figure 12: The time taken to execute all the SQL queries for each keyword query.

In Table 3, we present the distributions of MTNs and MPANs at levels 3, 5, and 7. Note that as most of the MTNs and MPANs are concentrated at higher levels, TD and TDWR perform better than BU and BUWR. We find that SBH performs well regardless of the distribution of MTNs.

	MTNs			MPANs		
Q	L3	L5	L7	L3	L5	L7
Q1	1	6	41	0	4	34
Q2	0	3	35	0	0	0
Q3	0	85	1418	0	94	1584
Q4	4	20	144	8	28	130
Q5	4	24	164	2	10	42
Q6	1	6	41	2	4	18
Q7	2	14	92	4	12	70
Q8	0	31	451	0	87	1172
Q9	0	8	40	0	4	4
Q10	0	6	83	0	10	92

Table 3: Distributions of MTNs and MPANs at levels 3, 5, and 7 for the 10 keyword queries (L3 is short for "Level 3", and so forth). Note that most of the MTNs and MPANs are concentrated at higher levels.

3.6 Performance at Higher Levels

We now investigate how the approaches perform as we vary the maximum level of the lattice. Table 4 shows the number of SQL queries executed for Q3 when increasing the maximum level from 3 to 7. As before, we note that the number of SQL queries executed increases as the maximum level is increased. We also note that reuse-based approaches perform better by executing fewer queries — BUWR executes 28% fewer queries than BU, while TDWR executes 52% fewer queries than TD, at level 7. Further, TDWR performs better than BUWR, owing to the presence of a large number of MPANs and MTNs at higher levels of the lattice. Finally, we note that SBH provides substantial reduction (79% reduction compared to BU) in the number of queries executed at higher levels of the lattice.

3.7 Performance Improvement with Reuse

We were interested in investigating the extent of the overlap between the descendants of each MTN. Increased overlap would allow more reuse, decrease the number of SQL queries executed and consequently improve runtime performance. Figure 13 shows the percentage of reuse, i.e., $100 * (1 - \frac{N_u}{N})$, where N_u is the

Level	BU	BUWR	TD	TDWR	SBH
3	0	0	0	0	0
5	294	233	225	136	101
7	5036	3624	3866	1818	1026

Table 4: Number of SQL queries executed in all the traversal techniques for Q3 with lattice level = 3, 5, 7. SBH provides substantial reduction in the number of queries executed at higher levels of the lattice. The approaches with reuse perform better than their respective counterparts without reuse.

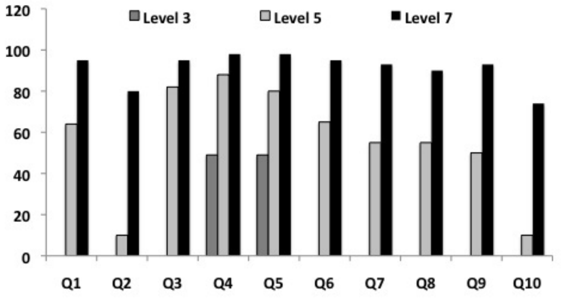


Figure 13: Percentage of reuse for the 10 keyword queries. While reuse is keyword dependent, it increases as the number of joins increases.

number of unique descendants of MTNs, and N is the total number of descendants of MTNs. The percentage of reuse for levels 3, 5, and 7 is shown. As expected, reuse increases as more joins are allowed. At level 3, only queries Q4 and Q5 show some overlap. However, we see substantial overlap between the descendants of the MTNs at levels 5 and 7. Specifically, Q2 and Q10 show a steep increase in overlap from level 5 to level 7. This increase is reflected in the performance of the reuse-based approaches and helps explain the performance of SBH over the other traversal approaches.

3.8 Comparison with Other Alternatives

Our proposed approach is not the unique one that can address the non-answer exploration problem. We therefore further compared our approach with other alternatives. Here we consider a simple indicator that is correlated with the "work" required to diagnose a non-answer. Our intent is to explore a simple quantitative metric that captures the intuition for why we think our approach may be helpful. Specifically, we compare our approach with the following two alternatives:

- *Return Nothing* (RN): Return nothing to the user for non-answers. This is the standard, existing KWS-S approach. To address the "why not" question, a developer would likely repeatedly submit modified queries by removing keywords from the original query. For instance, if the original keyword query were "k_1 k_2 k_3", then a user trying to understand the reason for the non-answer might additionally submit the queries: "k_1 k_2", "k_1 k_3", "k_2 k_3", "k_1", "k_2", and "k_3".
- *Return Everything* (RE): Do not build the lattice and return MPANs of the non-answers. Instead, explore alive descendants at runtime. For each descendant, issue the associated SQL query to determine its aliveness.

RN requires additional user effort to submit more queries. Both RE and our proposed approach remove this burden from the user. Meanwhile, the set of alive descendants returned by RN is both incomplete and redundant. It is incomplete, since only *minimal* alive descendants can be returned by existing KWS-S systems. That is, each leaf node of a candidate network is required to be bound by a keyword. As a result, alive descendants, including some MPANs, that do not satisfy this requirement will not appear in the results and hence are missing. It is also redundant, since some of the alive descendants returned may belong to answers (i.e., alive MTNs) but not non-answers (i.e., dead MTNs). Both incompleteness and redundancy are unsatisfactory for the purpose of debugging non-answers. On the other hand, RE returns the complete set of alive descendants. However, it is still redundant, since the aliveness of some descendants can be automatically inferred based on the aliveness of the others. Compared with RE, with the help of the lattice structure, our proposed approach can rule out this redundancy without sacrificing the completeness.

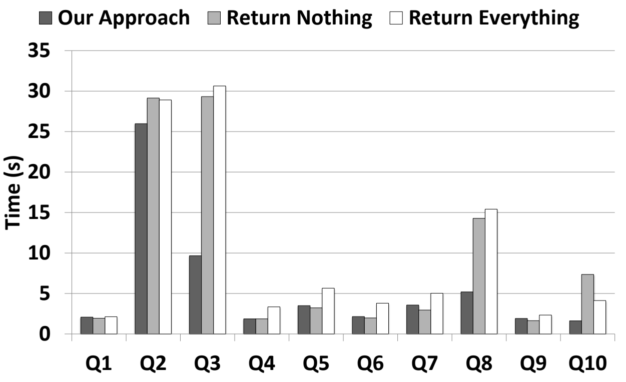


Figure 14: Response time when lattice level = 5

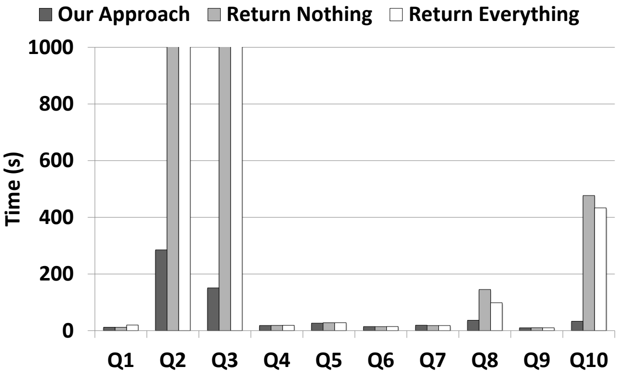


Figure 15: Response time when lattice level = 7

We further tested the system response time when the three approaches were adopted, in terms of the total execution time of the SQL queries issued. Figure 14 and 15 present the results for lattice levels 5 and 7. Our approach substantially reduces the response time for the more complicated queries Q2, Q3, Q8, and Q10, which contain three keywords, while the other queries only contain two. The improvement is more dramatic when multi-way joins (i.e., higher lattice levels) are allowed. For example, the response times are reduced by 84% and 99% for the two most costly queries Q2 and Q3, when the lattice level is 7 (i.e., up to 6 allowable joins).

4. RELATED WORK

Although we are not aware of any previous work that pertains to non-answers in the KWS-S context, the related work for the approaches and ideas presented in this paper is extensive.

Banks [1, 14], DBXplorer [2], and DISCOVER [11] are seminal papers in KWS-S. While Banks operates on a data graph, our paper is geared towards approaches like DISCOVER and DBXplorer, which use the underlying schema graph to explore relationships between the keywords. Several other KWS-S systems have been proposed over the years as well (see Yu et al. [25] for an extensive survey). Notably, Markowetz et al. [19] explored efficient generation and evaluation of candidate networks and briefly mentioned purging dead tuples in their paper about keyword search over streaming data. The Helix system [22] proposed a rule-based method for mapping keyword queries to structured queries. They automatically mined and manually tuned a set of patterns from query logs and mapped each pattern to a template query. A query was mapped to the best template once it arrived at runtime. Our static lattice structure is somewhat analogous to these templates. EASE [16] extensively leveraged offline computation to speed up runtime performance but did not consider the problem of non-answers. KWS-S-F [6] also leveraged offline computation but did not deal with non-answers. In addition, lattice structure has also been used in relaxing selection and join queries in relational databases to help users find more results [15].

We drew inspiration from Why Not [5] and Provenance of Non-Answers [12]. This work addressed non-answers in the context of single SQL queries and did not deal with KWS-S systems. Huang et al. [12] and Herschel et al. [8] provided tuple insertions or modifications that would yield the missing tuples. Chapman and Jagadish [5] tried to find the manipulation that led to a non-answer query. Tran and Chan [24] generated a modified query whose results included the user-specified missing tuple(s). In contrast, in a KWS-S system we cannot rely on user input, given that the user is assumed to be schema agnostic and may not even be aware that the KWS-S system is executing structured queries. Our notion of MPANs is similar to that of a frontier picky manipulation [5] in spirit. However, we feel that MPANs are more suited for the keyword search context; they represent the subset of keywords that would render a dead relationship alive. Work on lineage and provenance including [4, 20] influenced the lattice structure and use of MPANs to explain non-answer queries in KWS-S.

While we focused on pure relational database techniques in this paper, there has also been work on mapping sets of structured tuples to virtual documents and then applying information retrieval techniques to find results that match the keywords (e.g., the EKSO system [23]). However, this idea relies on inverted indices built over materialized views. In typical industry systems, these indices are updated only at some pre-determined time-intervals (mostly, on a nightly basis). This then implies that they may suffer from severe data staleness issues, for in the daily use of a relational database, any changes to the underlying data can impact answers and non-answers to keyword queries almost immediately. Moreover, it actually cannot work for non-answers if only "and" semantics is considered. Using "and" semantics, non-answers would never be displayed to the user. Nonetheless, this raises a very interesting point that there might be an alternative way to deal with the problem: replace "and" semantics by "or" semantics, though it seems to be equivalent to the "Returning Nothing" strategy discussed in Section 3.8 and thus suffers from issues such as incompleteness and redundancy. In fact, Hristidis et al. have considered the "or" semantics in KWS-S systems [10]. However, their focus was to efficiently present the end users with a list of top-k matches for moderate values of k. In contrast, our goal is to enable system developers to debug non-answers so providing top-k matches is insufficient. Nevertheless, this merits further exploration but it is out of the scope of this paper.

5. CONCLUSION

In this work we take a first step towards building a KWS-S system that exposes non-answer queries to system developers for the purpose of debugging the system. Given the exponential complexity of answer generation in KWS-S, exposing and explaining non-answer queries is a time-consuming process. We leveraged offline computation to generate a lattice structure and exploited the overlap between the queries to efficiently determine non-answer queries and their closest alive sub-queries.

While we concentrated on improving the performance of discovering and investigating non-answer queries in the KWS-S domain, this work opens up a couple of interesting directions for future work. For instance, debugging is often an interactive process and it is worth studying how to combine the search for MPANs with user intervention. Meanwhile, pushing user-defined constraints into the search procedure might greatly prune the search space and therefore significantly improve the efficiency. All of these are interesting questions that deserve further investigation.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. This work was supported in part by NSF Grant III-1018792 and by a Google Focus Award.

6. 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. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] O. Benjelloun, A. D. Sarma, C. Hayworth, and J. Widom. An introduction to ULDBs and the Trio system. *IEEE Data Eng. Bull.*, 29(1):5–16, 2006.

[5] A. Chapman and H. V. Jagadish. Why Not? *SIGMOD '09*.

[6] 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*.

[7] DBLife. http://dblife.cs.wisc.edu.

[8] M. Herschel, M. A. Hernández, and W. C. Tan. Artemis: A system for analyzing missing answers. *PVLDB*, 2(2):1550–1553, 2009.

[9] HP Autonomy. http://www.autonomy.com/.

[10] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

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

[12] J. Huang, T. Chen, A. Doan, and J. Naughton. On the provenance of non-answers to queries over extracted data. *VLDB '08*.

[13] IBM Coremetrics. http://www-01.ibm.com/software/marketing-solutions/coremetrics/.

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

[15] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[16] 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*.

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

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

[19] A. Markowetz, Y. Yang, and D. Papadias. Keyword search over relational tables and streams. *ACM Trans. Database Syst.*, 34(3).

[20] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. Why so? or Why no? functional causality for explaining query answers. *CoRR*, abs/0912.5340, 2009.

[21] Orcale Endeca. http://www.oracle.com/us/products/applications/commerce/endeca/.

[22] S. Paparizos, A. Ntoulas, J. Shafer, and R. Agrawal. Answering web queries using structured data sources. In *SIGMOD '09*, pages 1127–1130, New York, NY, USA, 2009. ACM.

[23] Q. Su and J. Widom. Indexing relational database content offline for efficient keyword-based search. In *IDEAS*, pages 297–306, 2005.

[24] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *SIGMOD '10*, pages 15–26, New York, NY, USA, 2010. ACM.

[25] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.