ROMA TRE
UNIVERSITÀ DEGLI STUDI

Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79
00146 Roma, Italy

Incremental Keyword Search in Relational Databases

Roberto De Virgilio, Antonio Maccioni, Riccardo Torlone

RT-DIA-204-2013 **March 2013**

Università Roma Tre,
Via della Vasca Navale, 79
00146 Roma, Italy

ABSTRACT

Keyword-based search is becoming the standard way to access any kind of information and it is considered today an important add-on of relational database management systems. The approaches to keyword search over relational data usually rely on a two-step strategy in which, first, tree-shaped answers are built by connecting tuples matching the given keywords and, then, potential answers are ranked according to some relevance criteria. In this paper, we illustrate a novel technique to this problem that aims, rather, at generating directly the best answers. This is done progressively by combining the shortest join paths that involve the tuples relevant to the query. We show that, in this way, answers are retrieved in order of relevance and can be then returned as soon as they are built. The approach does not require the materialization of ad-hoc data structures and avoids the execution of unnecessary queries. A comprehensive evaluation demonstrates that our solution strongly reduces the complexity of the process and guarantees an high level of accuracy.

1 Introduction

Today, everyone can access an incredibly large quantity of information and this requires to rethink the traditional methods and techniques for querying and retrieving data, mainly because the vast majority of users has little or no familiarity with computer systems. This need has originated a large set of proposals of non-conventional methods to access structured and semi-structured data. Among them, several approaches have focused on the adoption of IR strategies [24] for keyword-based search on the top of traditional database management systems, with the goal of freeing the users from the knowledge of query languages and/or the organization of data [4, 14, 16, 17, 19].

Example 1 *Let us consider the relational database in Figure 1 in which employees with different skills and responsibilities work in projects of an organization. A keyword-based query over this database searching for experts of* Java *in the* CS *department could simply be:* $Q_1 = \{Java, CS\}$. *A possible answer to* Q_1 *is the set of joining tuples* $\{t_3, t_{14}, t_{16}\}$ *that includes the given keywords.*

Usually, keyword-based search systems over relational data involve the following key steps: (i) generation of tree-shaped answers (commonly called *joining tuple trees* or JTT) built by joining the tuples whose values match the input keywords, and (ii) ranking of the answers according to some relevance criteria. At the end, only the top-k answers are returned to the users. The core problem of this approach is the construction of the JTT's. In this respect, the various approaches proposed in the literature can be classified in two different categories: *schema-based* [1, 3, 15, 14, 20, 21, 23] and *schema-free* [4, 12, 16, 17, 18, 10, 9]. Schema-based approaches usually implement a middleware layer in which: first, the portion of the database that is relevant for the query is identified, and then, using the database schema and the constraints, a (possibly large) number of SQL statements is generated to retrieve the tuples matching the keywords of the query. Conversely, schema-free approaches first build an in-memory, graph-based, representation of the database, and then exploit graph-based algorithms and graph exploration techniques to select the subgraphs that connect nodes matching the keywords of the query.

In this paper, we present a novel technique to keyword-based search over relational databases that, taking inspiration from both the schema-based and the schema-free approaches, aims at generating progressively the most relevant answers, avoiding the selection of bunches of potential answers followed by their ranking, as it happens in other approaches. As suggested in [22], our approach exploits only the capabilities of the underlying RDBMS and does not require the construction and maintenance of ad-hoc, in-memory data structures. Moreover, by avoiding redundant accesses to data, we are able to keep the computational complexity of the overall process linear in the size of the database. In a graph-oriented vision of the database, the basic idea is to search and combine incrementally the shortest paths of joining tuples that are relevant to the query. This is done by first identifying all the paths in the relational schema involving attributes linked by primary and foreign keys. Then, without building in-memory graph-shaped structures, such paths are enriched with data by traversing them backward. This step only requires simple selection and projection operations. If the backward navigation is not able to generate an answer, the paths are navigated forward using all the information retrieved in the backward phase, without further accessing the database. We show that, in this way,

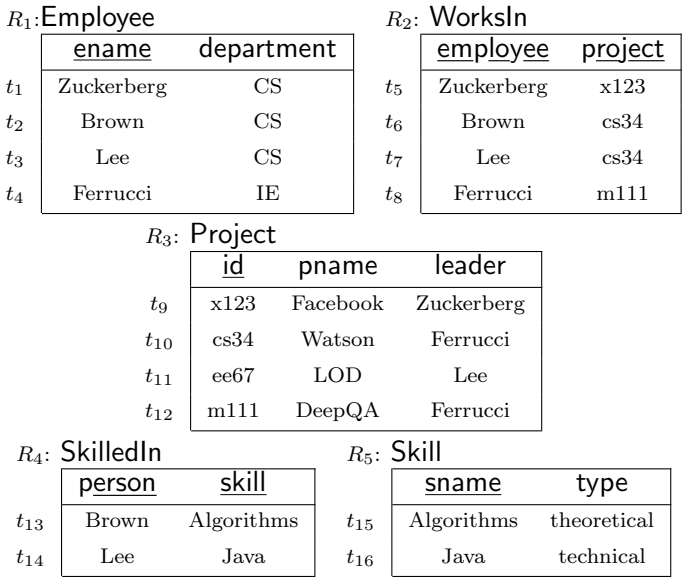


Figure 1: An example of relational database: schema and its data

answers are retrieved in order of relevance. This eliminates the need to compare answers and allows us to return the results to the user as soon as they are built.

To validate our approach, we have developed a system for keyword-based search over relational databases that implements the technique described in this paper. This system has been used to perform several experiments over an available benchmark [6] that have shown a marked improvement over other approaches in terms of both effectiveness and efficiency. This demonstrates that our approach is able to generate relevant answers while reducing the complexity of the overall process.

Outline. The rest of the paper is organized as follows. Section 2 introduces a graph-based data model that we use throughout the paper. In Section 3, we describes in detail our incremental method for building top-k answers to keyword-based queries. An analysis of the computational complexity and of the monotonicity of the approach is faced in Section 4. The experimental results are reported in Section 5 and, in Section 6, we discuss related works. Finally, in Section 7, we sketch conclusions and future work.

2 Preliminaries

2.1 A graph data model over relational data

In our approach, we model a relational database $\mathbf{d}$ in terms of a pair of graphs $\langle \mathcal{SG}, \mathcal{DG} \rangle$ representing the schema and the instance of $\mathbf{d}$, respectively. We point out however that only $\mathcal{SG}$ will be materialized while $\mathcal{DG}$ is just a conceptual notion.

Definition 1 (Schema Graph) *Given a relational schema $\mathcal{RS} = \langle \mathcal{R}, \mathcal{A} \rangle$, where $\mathcal{R}$ is a set of relation schemas and $\mathcal{A}$ is the union of all attributes of $\mathcal{R}$, a schema graph $\mathcal{SG}$ for $\mathcal{RS}$ is a directed graph $\langle V, E \rangle$ where $V = \mathcal{R} \cup \mathcal{A}$ and there is an edge $(v_1, v_2) \in E$ if one of the following holds: (i) $v_1 \in \mathcal{R}$ and v_2 is an attribute of v_1, (ii) $v_1 \in \mathcal{A}$ belongs to a key of a relation $R \in \mathcal{R}$ and v_2 is an attribute of R, (iii) $v_1 \in \mathcal{A}$, $v_2 \in \mathcal{A}$ and there is a foreign key between v_1 and v_2.*

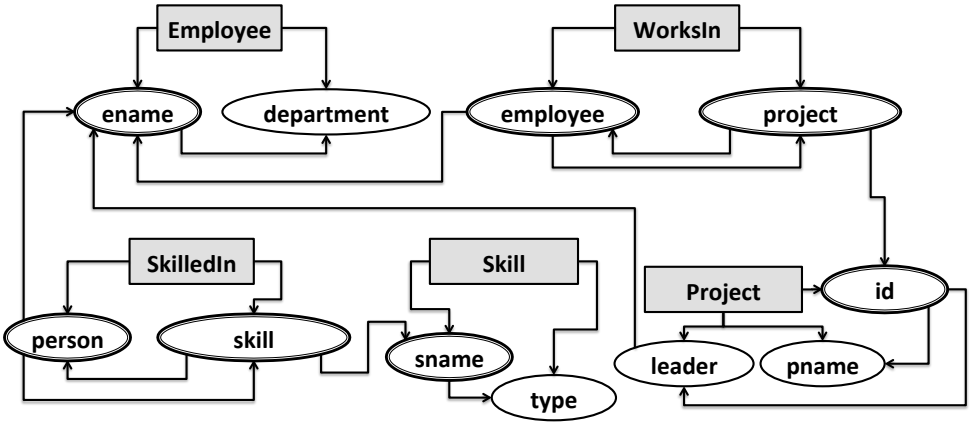


Figure 2: An example of schema graph $\mathcal{SG}$

For instance, the schema graph for the relational database in Figure 1 is reported in Figure 2. In a schema graph the sources represent the tables of a relational database schema (grey nodes) and the paths represent the relationships between attributes according to primary and foreign keys. The double-marked nodes denote the keys of a relation.

Definition 2 (Schema Path) *A schema path in a schema graph $\mathcal{SG} = \{V, E\}$ is a sequence $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_f$ where $(v_i, v_{i+1}) \in E$ and v_1 is a relation node.*

An example of schema path for the schema graph in Figure 2 is *SkilledIn* $\rightarrow$ *skill* $\rightarrow$ *sname*.

Let us now fix an injective function denoted by idx that maps each tuple to a *tuple-id* (tid for short).

Definition 3 (Data Graph) *Given a relational database instance $\mathcal{I} = \langle \mathcal{R}, \mathcal{A}, I, \mathcal{D} \rangle$, where I is the set of all tids and $\mathcal{D}$ is the set of all data values occurring in the database, a data graph $\mathcal{DG}$ on $\mathcal{I}$ is a directed graph $\langle V, E \rangle$ where $V = \mathcal{R} \cup \mathcal{A} \cup I \cup \mathcal{D}$ and there is an edge $(v_1, v_2) \in E$ if one of the following holds: (i) $v_1 \in \mathcal{R}$ and v_2 is an attribute of v_1, (ii) $v_1 \in \mathcal{A}$ belongs to a key of a relation R and v_2 is the tid of a tuple for R, (iii) v_1 is a tid in I and v_2 is a value of a tuple t such that $v_1 = \text{idx}(t)$.*

Figure 3 shows the data graph on the database of Figure 1. Note that we assume, for the sake of simplicity, that each relation has an explicit attribute for its tids.

We now introduce the notion of data path. Intuitively, while a schema path represents a route to navigate relational data for query answering, a data path represents an actual navigation through data to retrieve the answer of a query.

Definition 4 (Data Path) *Given a schema path $sp = R \rightarrow A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_k$ the data path dp following sp is the path $R \rightarrow A_1 \rightarrow \tau_1 \rightarrow \ldots A_k \rightarrow \tau_k \rightarrow v$, where: (i) each τ_i denotes either a variable denoting a tid or the tid of a tuple belonging to the relation involving A_i and (ii) v is a value belonging to the tuple with tid τ_k.*

Let us consider again the example in Figure 3. The data path that follows the schema path $sp =$ *SkilledIn* $\rightarrow$ *skill* $\rightarrow$ *sname* is the following:

$$dp_1 : SkilledIn \rightarrow skill \rightarrow x_1 \rightarrow sname \rightarrow t_{15} \rightarrow Algorithms$$

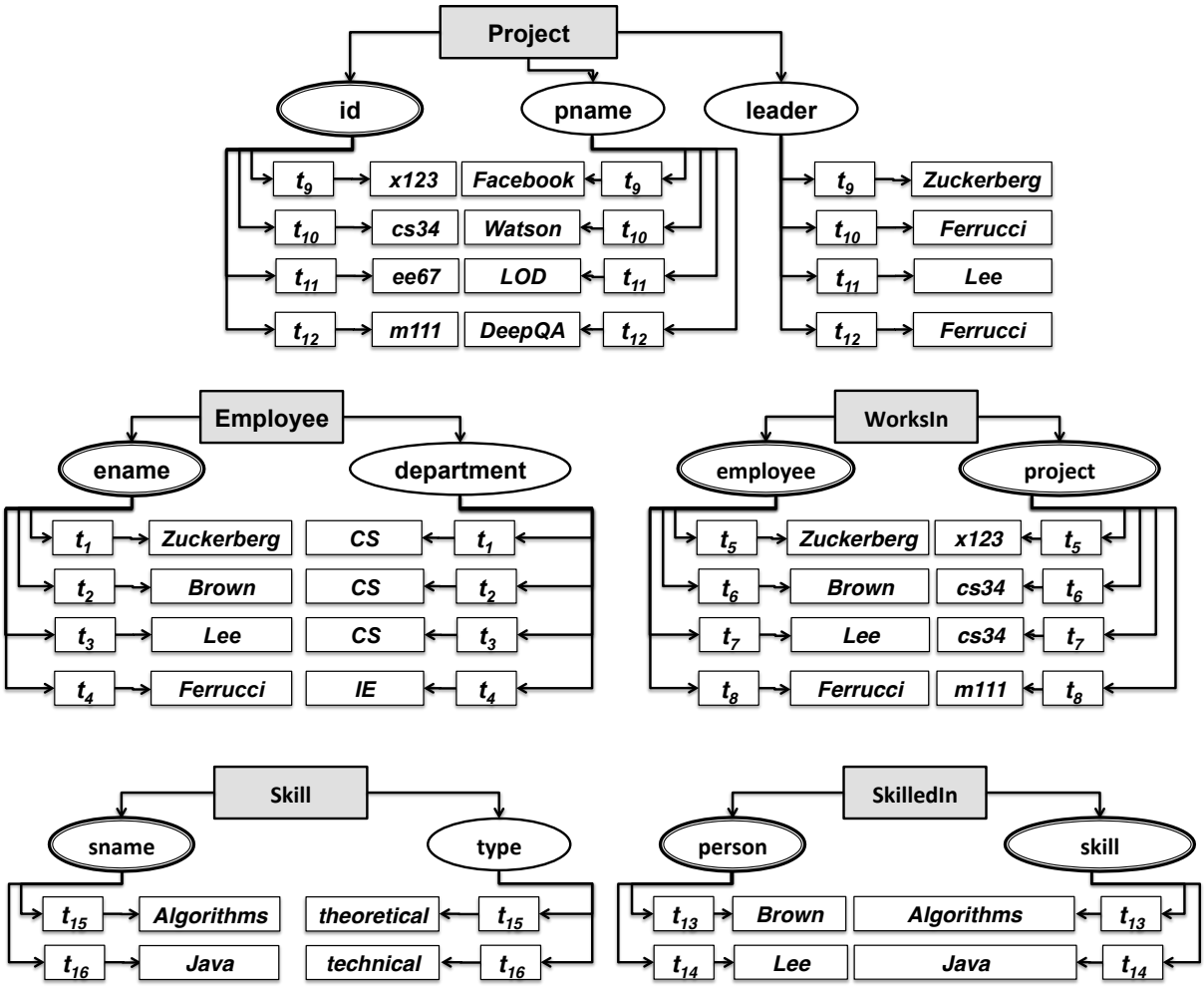


Figure 3: An example of data graph $\mathcal{DG}$

Basically, this path describes the fact that the *sname* of the tuple with tid t_{15} is related to the *skill* of a tuple x_1 in relation *SkilledIn*.

An *instance* of a data path dp is a function ϕ that associates a tid with each variable occurring in dp. As an example, an instance of the data path dp_1 above associates t_{13} with x_1.

2.2 Answers to a keyword-based query

We consider the traditional Information Retrieval approach to value matching adopted in full text search and we denote the matching relationship between values with $\approx$. We have used standard libraries for its implementation and since this aspect is not central in our approach, it will not be discussed further. Given a tuple t and a value v, we then say that t matches v, also denoted for simplicity by $t \approx v$, if there is a value v' in t such that $v \approx v'$.

Definition 5 (Answer) *An answer to a keyword-based query Q is a set of tuples S such that: (i) for each keyword q of Q there exists a tuple t in S that matches q and (ii) the tids of the tuples in S occur in a set of data path instances having at least one tid in common.*

$[cl_{Java}]$:

$$\left(\begin{array}{l} dp_1 : \texttt{SkilledIn} \rightarrow \texttt{SkilledIn.skill} \rightarrow \texttt{t}_{14} \rightarrow Java \\ dp_2 : \texttt{Skill} \rightarrow \texttt{Skill.sname.} \rightarrow \texttt{t}_{16} \rightarrow Java \\ dp_3 : \texttt{SkilledIn} \rightarrow \texttt{SkilledIn.person} \rightarrow x_1 \rightarrow \texttt{SkilledIn.skill} \rightarrow \texttt{t}_{14} \rightarrow Java \\ dp_4 : \texttt{SkilledIn} \rightarrow \texttt{SkilledIn.skill} \rightarrow x_2 \rightarrow \texttt{Skill.sname} \rightarrow \texttt{t}_{16} \rightarrow Java \\ dp_5 : \texttt{SkilledIn} \rightarrow \texttt{SkilledIn.person} \rightarrow x_3 \rightarrow \texttt{SkilledIn.skill} \rightarrow x_4 \rightarrow \texttt{Skill.sname} \rightarrow \texttt{t}_{16} \rightarrow Java \end{array}\right)$$

$[cl_{CS}]$:

$$\left(\begin{array}{l} dp_6 : \texttt{Employee} \rightarrow \texttt{Employee.department} \rightarrow \texttt{t}_1 \rightarrow CS \\ dp_7 : \texttt{Employee} \rightarrow \texttt{Employee.department} \rightarrow \texttt{t}_2 \rightarrow CS \\ dp_8 : \texttt{Employee} \rightarrow \texttt{Employee.department} \rightarrow \texttt{t}_3 \rightarrow CS \\ \ldots \\ dp_9 : \texttt{SkilledIn} \rightarrow \texttt{SkilledIn.person} \rightarrow x_5 \rightarrow \texttt{Employee.ename} \rightarrow x_6 \rightarrow \texttt{Employee.department} \rightarrow \texttt{t}_3 \rightarrow CS \\ \ldots \end{array}\right)$$

Figure 4: Clusters of data paths for $Q_1 = \{Java, CS\}$

An example of answer, with reference to the query $Q_1 = \{Java, CS\}$, is the set of tids $\{t_3, t_{14}, t_{16}\}$ that are contained in the instances of the set $\{dp_5, dp_9\}$ of data path in Figure 4.

Note that we assume the AND semantics for the keywords in Q. Note also that our notion of answer basically corresponds to the notion of joining tuple tree (JTT) [15].

As usual, an answer S_1 is considered more relevant than another answer S_2 if S_1 is "more compact" than S_2 since, in this case, the keywords of the query are closer between each other [8]. This is captured by a scoring function that simply returns cardinality of S.

Problem Statement. Given a relational database **d** and a keyword search query $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$, where each q_i is a keyword, we aim at finding the top-k ranked answers $S_1, S_2, \ldots, S_k$.

3 Path-oriented Search

3.1 Overview

Given a keyword-based query Q, our technique consists of two main phases:

Clustering. In the first phase all the data paths having an ending node that matches one of the keywords in Q are generated and grouped in clusters, one for each keyword. The clusters are kept ordered according to the length of the data paths, with the shortest paths coming first. As an example, given the query $Q_1 = \{\texttt{Java}, \texttt{CS}\}$ and the relational database in Figure 1, we obtain the clusters shown in Figure 4.

Building. The second phase aims at generating the most relevant answers by combining the data paths generated in the first step. This is done iteratively by picking, in each step, the shortest data paths from each cluster: if there is an instance of these data paths having a tuple in common, we have found an answer. The search proceeds in this way with longer data paths that follow in the clusters. As an example, given the cluster in Figure 4, the first answer is obtained by combining the instances of dp_5 and dp_9 that associate t_{14} to both x_3 and x_4 in dp_5, and t_{14} to x_5 and t_3 to x_6 in dp_9. Since these two instances share the tid t_{14} they form an answer.

The most tricky task of the whole process occurs in the second phase: it consists of instantiating a set of data paths $DP = \{dp_1, \ldots, dp_n\}$ and verifying if the results have a tuple in common. We have adopted here a strategy that tries to minimize the number of database accesses, as follows.

1. Each data path $dp \in DP$ is navigated backward starting from the last node. For instance, given dp_5, we start from the node *Java* and proceed until the variable x_4 is encountered. According to the information carried by this data path, the only possible substitution for x_4 is the tid of the tuples that have as `SkilledIn.skill` the same value occurring in `Skill.sname` of the tuple with tid t_{16}, i.e. *Java*. In the database of Figure 1 we have $x_4 = t_{14}$. It turns out that $x_3 = t_{14}$ as well since x_4 and x_3 refer to the same tuple in the `SkilledIn` relation. If we find a multiple substitutions for some variable the analysis of the current data path stops.

2. If the backward navigation produces a single substitution ϕ for all the variables occurring in the data paths (note that each variable occurs in a single data path only) and the same tid occurs in each of $\phi(dp_1), \ldots, \phi(dp_n)$, then we have found an answer.

3. If the backward navigation is not able to generate an answer, the data paths are navigated forward using all the tids retrieved in the first step as substitutions for the remaining variables. As an example, during the backward phase, dp_5 is completely instantiated but it is not possible to find a substitution for x_5 in dp_9. Then, in the forward navigation, we substitute x_5 with the tid used for instantiating x_3 and x_4 in dp_5. In Section 3.3 we will provide more examples of this kind.

This process guarantees a *monotonic* construction of the answers (i.e. the answer of the i-th step is always more relevant than that of the $i+1$-th step) and a linear time complexity with respect to the size of the input. This makes possible to return answers as soon as they are computed.

The rest of the section describes our technique in more detail.

3.2 Clustering

Before computing answers, we organize the paths into clusters, as shown in the Algorithm 1. Intuitively we create a cluster of data paths dp for each keyword $q_i \in Q$ (lines 2-3). In particular we start from each data path $R \rightarrow A \rightarrow tid \rightarrow v$ such that $q_i \approx v$ (line 4). Then we generate the data paths following the route of each schema path sp ending into the attribute A (line 5) using `last`(sp), that returns the final node of sp, and using `generate`(sp, tid, v) that generates a data path including variables in sp and ending sp with tid and v (line 6). Each data path is implemented as an array of tokens, while each cluster is a priority queue, where the priority is in inverse proportion to the length of each data path. Finally, $\mathcal{CL}$ is a simple set of clusters.

3.3 Building

Once we have clustered the data paths in $\mathcal{CL}$, we combine them to provide the top-k answers to Q. Our building algorithm is an iterative process which incrementally computes

Algorithm 1: Clustering

Input : Query Q, schema graph $\mathcal{SG}$, data graph $\mathcal{DG}$.
Output: The set of clusters $\mathcal{CL}$.

```
1 CL ← ∅;
2 foreach q_i ∈ Q do
3     cl_i ← ∅;
4     foreach R → A → tid → v in DG : q_i ≈ v do
5         foreach sp in SG : last(sp) = A do
6             dp ← generate(sp,tid,v);
7             cl_i.enqueue(dp);
8     CL ← CL ∪ {cl_i};
9 return CL;
```

the answers. Differently from existing schema-based approaches, we do not formulate SQL join queries from the analysis of the schema of **d**, possibly introducing unnecessary overhead. Furthermore, contrary to schema-free approaches, we do not provide graph exploration techniques to analyse graph-shaped ad-hoc data structures. Our approach follows an hybrid strategy: similarly to schema-based approaches, we use schema information to generate the data paths of interest for the query. Then similarly to schema-free approaches we explore the most promising data paths of each cluster: in this case we operate simple SQL statements, i.e. selections and projections on the relations of **d**, to instantiate the variables of the data paths. Finally, we incrementally generate an answer. We are more efficient and effective than graph exploration because every data path is computed independently and we don't have to query relations (or portions of relations) that are not interesting for the final answer (i.e. no space and time overhead).

In detail, following the Algorithm 23, we extract all top data paths (i.e. the shortest ones) from each cluster into a set DP (lines 5-6). This task is supported by the procedure `dequeueTop`. Then we generate all possible combinations C of paths within DP (line 12) in order to find the best candidates to be answers (i.e. the task is performed by the procedure `validCombinations`). Each combination c is a connected directed graph that has to contain exactly one data path from each cluster: two paths from the same cluster cannot belong to the same combination and all clusters have to participate in each combination, i.e. AND-semantics. We try to combine paths with the same length. However two clusters could provide their longest paths with different length. In this case, to satisfy the AND-semantics, if a cluster cl_i becomes empty then we re-enqueue those data paths $dp \in DP$ such that $dp \rhd cl_i$ (lines 9-11). Note that, given a cluster cl_i corresponding to a keyword q_i, if $q_i \approx \texttt{last}(dp)$ then we denote $dp \rhd cl_i$. In this case we combine also data paths with different length. For instance referring to our example, at the first running of the algorithm we have to combine dp_1, dp_2 from cl_{Java} with dp_6, dp_7, dp_8 from cl_{CS}. To avoid a possible exponential number of combinations and useless path processing, we check before combining paths if all those paths cross a common table, also through its attribute nodes (i.e. the task is performed by the procedure `validCombinations`). For instance referring to Figure 4, dp_5 and dp_9 can be combined since they both cross the table `SkilledIn`: the former through the attributes `skill` and `person`, the latter through only

Algorithm 2: Building

Input : The clusters $\mathcal{CL}$, a query Q, the number k.
Output: The set of answers $\mathcal{S}$.

```
1  finished ← false;
2  S ← ∅;
3  while ¬finished do
4      DP ← ∅;
5      foreach cl_i ∈ CL do
6          DP ← DP ∪ dequeueTop(cl_i);
7      if CL = ∅ then finished ← true;
8      else
9          foreach cl_i ∈ CL: cl_i = ∅ do
10             foreach dp ∈ DP: dp ▷ cl_i do
11                 cl_i.enqueue(dp);
12     C ← validCombinations(DP );
13     foreach c ∈ C do
14         P ← ∅; C_d ← ∅;
15         foreach dp ∈ c do
16             is_sol ← backward_exploration(dp, Q, P, C_d);
17         if is_sol then
18             S.enqueue(P.keys);
19         else if forward_exploration(P, C_d) then
20             S.enqueue(P.keys);
21         if |S| = k then
22             return S;
23 return S;
```

`person`. This is a necessary condition for finding a common tid node. Intuitively the best answer contains tuples strictly correlated, e.g., a tuple containing all the keywords or tuples directly correlated by foreign key constraints. Referring to our example there is no valid combination in the first two runs of the algorithm. Therefore we have to extract longer data paths from $\mathcal{CL}$ and we find the first valid combination that is $c = \{dp_5, dp_9\}$. Now we have to verify if c brings an answer: if the test is positive, we extract all tids of c, i.e. the answer S_i, to include in the set $\mathcal{S}$. This evaluation is performed by the procedures `backward_exploration` and `forward_exploration`, described in the Algorithm 27 and in the Algorithm 18. Such procedures keep a map $\mathcal{P}$ where the key is a tid and the value is the number of occurrences of the tid in the combination c. If c brings an answer, then S_i is the set of keys extracted from $\mathcal{P}$ (line 18 and line 20). The building ends when we computed k answers (line 22) or the set $\mathcal{CL}$ is empty (line 7).

Backward Exploration. This procedure analyses a data path dp backwards: in other terms we follow the foreign and primary key constraints contrariwise. For instance let us consider Figure 5 that depicts the backward exploration at work on the combination $c = \{dp_5, dp_9\}$. Each data path is analysed independently from each other, i.e. in our example dp_5 and dp_9. The `backward_exploration` procedure takes as input a data path dp to analyse, the query Q, and the map $\mathcal{P}$. Moreover the procedure takes as input a set $\mathcal{C}_d$ of conditions whose functionality will be described in the forward exploration. Therefore we start the exploration of dp from the data value v (line 1). When we meet a tid t_y we insert it in $\mathcal{P}$ (line 3 and line 14), i.e. if t_y already exists in $\mathcal{P}$ then we increment the value associated to t_y in $\mathcal{P}$. Moreover, we maintain the information about the corresponding relation R of t_y from dp (line 2), i.e. the procedure `relation` is responsible of this task.

For instance Figure 5.(a) depicts the exploration of the path dp_5; in this case starting from *Java* we meet the tid t_{16}. The algorithm updates $\mathcal{P}$ inserting the pair $\{t_{16}, 1\}$. When we meet a variable x, we can have the following: if the attribute A associated to x belongs to the same relation R containing t_y then we are following a primary key constraints (line 25), i.e. x is a value associated to t_y, otherwise we are following a foreign key constraints - $R \neq$ `relation`(x,dp) - (line 10), i.e. x is a value associated to a tid related to t_y. In Figure 5.(b), the variable x_4 is associated to the attribute `skill` belonging to the relation `SkilledIn`, while t_{16} is in the table `Skill`. In this case we are following a foreign key constraint: from the relation R of the variable x the `backward_exploration` procedure extracts the tuple having the data value v associated to the attribute A, i.e. it is a simple selection $\sigma_{A=v}(R)$ (line 13). In Figure 5.(b) we have $\sigma_{skill=\text{“}Java\text{”}}(SkilledIn)$ resulting the tid t_{14} as shown in Figure 5.(c). Once we retrieve the new tid, we insert it into the map $\mathcal{P}$ and it becomes the new current tid t_y. As shown in Figure 5.(d), the attribute A associated to the variable x_3 belongs to the same relation R of t_y, that is `SkilledIn`. In this case we are following a primary key constraint: the `backward_exploration` procedure extracts the data value associated to the attribute A of the same tuple, i.e. it is a simple projection $\pi_A(\sigma_{tid=t}(R))$ on the attribute a of the tuple with id equal to t_y (line 25). In our example x_3 corresponds to t_{14} itself and we have $\pi_{person}(\sigma_{tid=t_{14}}(SkilledIn))$, that is the value *Lee*. The exploration of dp_5 terminates. Similarly we explore dp_9. In Figure 5.(e) we start from the data value *CS*. We insert t_3 in $\mathcal{P}$; then we meet the variable x_6: the associated attribute a is `ename` that belongs to the same relation `Employee` of t_3. Therefore x_6 corresponds to t_3 and we have the projection $\pi_{ename}(\sigma_{tid=t_3}(Employee))$, resulting the data value *Lee*, i.e. Fig 5.(f). Finally we meet the variable x_5, as shown in Figure 5.(g): the attribute a associated to x_5 is `person` that belongs to the relation `SkilledIn`. In this case we are

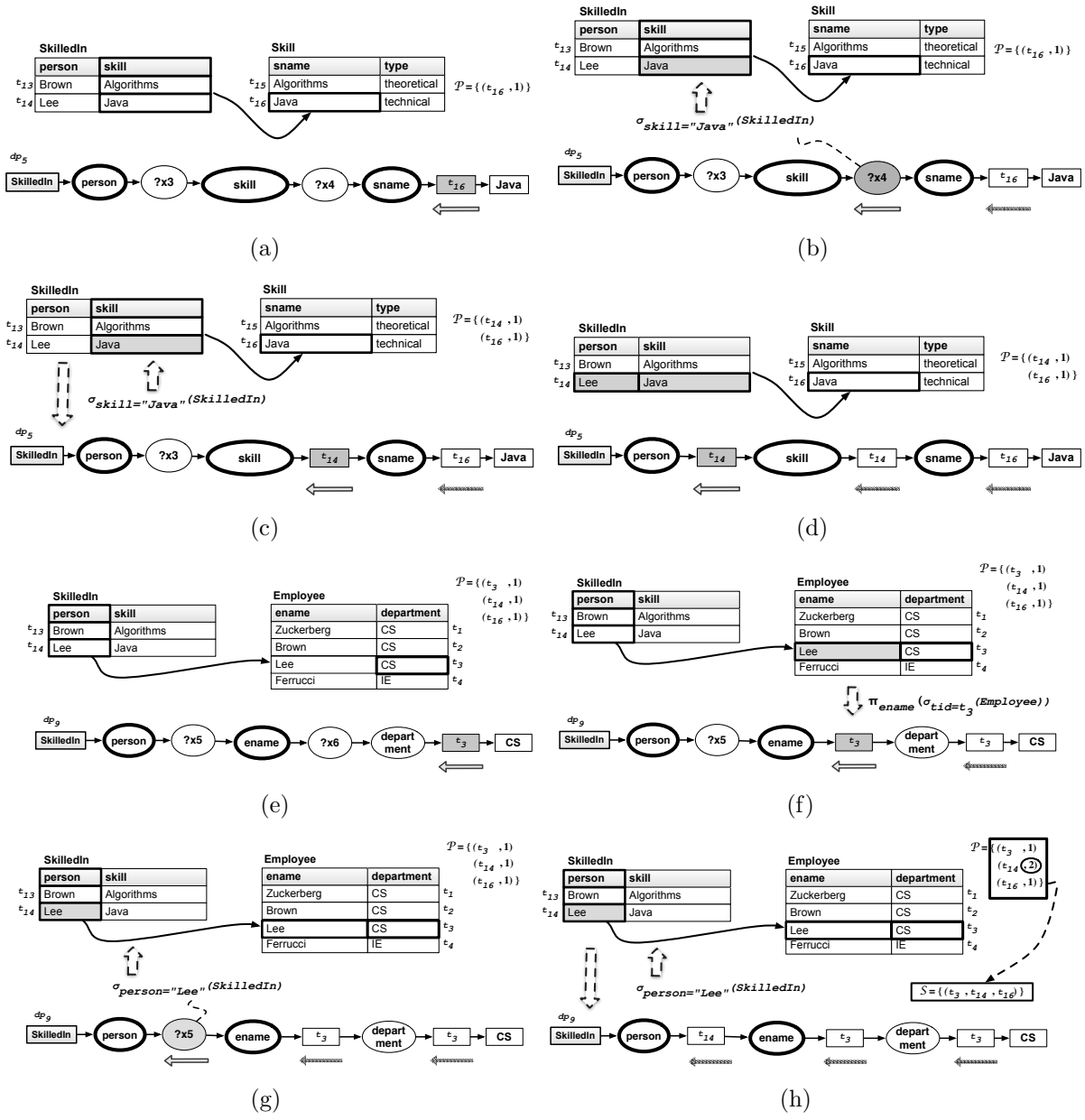


Figure 5: Backward exploration at work for Q_1

following a foreign key constraint; we execute the selection $\sigma_{person="Lee"}(SkilledIn)$ and we retrieve the tid t_{14}, as depicted in Figure 5.(h). In this case t_{14} exists in $\mathcal{P}$: we have to increment the value associated to t_{14} in $\mathcal{P}$. If $\mathcal{P}$ contains a pair $\{t, n\}$, where $n = |Q|$, then t_y represents the tuple able to reach all tuples matching the keywords of Q (line 15): in this case the tids in $\mathcal{P}$ represent an answer to insert in $\mathcal{S}$; in our example we have the answer $\{t_3, t_{14}, t_{16}\}$.

Algorithm 3: Backward Exploration

Input : A data path dp, the query Q, the map $\mathcal{P}$, the set $\mathcal{C}_d$.
Output: A boolean value.

```
1  i ← dp.length; v ← dp[i]; t_y ← dp[i − 1];
2  R ← relation(t_y,dp);
3  P.put(t_y, P.get(t_y)+1);
4  if P.get(t_y) = |Q| then
5      return true;
6  i ← i − 3;
7  while i > 2 do
8      A ← dp[i − 1];
9      x ← dp[i];
10     if R ≠ relation(x,dp) then
11         R ← relation(x,dp);
12         if |σ_{A=v}(R)| = 1 then
13             t_y ← idx(σ_{a=v}(R));
14             P.put(t_y, P.get(t_y)+1);
15             if P.get(t_y) = |Q| then
16                 return true;
17         else
18             if |σ_{A=v}(R)| > 1 then
19                 γ ←< R, A, v >;
20                 C_d ← C_d ∪ {γ};
21                 return false;
22             else
23                 return false;
24     else
25         v ← π_A(σ_{tid=t_y}(R));
26     i ← i − 2;
27 return false;
```

As shown in this example, in the building process we exploit simple operations of *selection* (σ) and *projection* (π) on limited groups of tuples (often only one) and we never utilize join ($\bowtie$) operations (possibly complex cartesian products), as competitor approaches. For instance let us consider Figure 6. The figure depicts the possible overhead introduced by competitors executing join operations; in this case the complete set of tuples

R': **(Employee $\bowtie$ SkilledIn) $\bowtie_{skill=sname}$ Skill**

ename	department	skill	sname	type
Brown	CS	Algorithms	Algorithms	theoretical
Lee	CS	Java	Java	technical

Figure 6: Schema-based approaches computation

$[cl_{Brown}]$:

$$\left(\begin{array}{l} dp_1 : \texttt{Employee} \rightarrow \texttt{Employee.name} \rightarrow \texttt{t}_2 \rightarrow Brown \\ dp_2 : \texttt{WorksIn} \rightarrow \texttt{WorksIn.employee} \rightarrow \texttt{t}_6 \rightarrow Brown \\ dp_3 : \texttt{SkilledIn} \rightarrow \texttt{SkilledIn.name} \rightarrow \texttt{t}_{13} \rightarrow Brown \\ \ldots \\ dp_i : \texttt{WorksIn} \rightarrow \texttt{WorksIn.project} \rightarrow x_1 \rightarrow \texttt{WorksIn.employee} \rightarrow x_2 \rightarrow \texttt{Employee.ename} \rightarrow \texttt{t}_2 \rightarrow Brown \\ \ldots \end{array}\right)$$

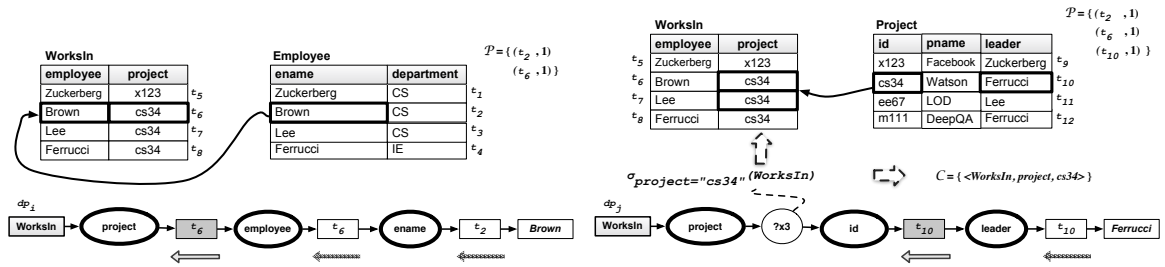
$[cl_{Ferrucci}]$:

$$\left(\begin{array}{l} dp_4 : \texttt{Employee} \rightarrow \texttt{Employee.ename} \rightarrow \texttt{t}_4 \rightarrow Ferrucci \\ dp_5 : \texttt{WorksIn} \rightarrow \texttt{WorksIn.employee} \rightarrow \texttt{t}_8 \rightarrow Ferrucci \\ dp_6 : \texttt{Project} \rightarrow \texttt{Project.leader} \rightarrow \texttt{t}_{10} \rightarrow Ferrucci \\ \ldots \\ dp_j : \texttt{WorksIn} \rightarrow \texttt{WorksIn.project} \rightarrow x_3 \rightarrow \texttt{Project.id} \rightarrow x_4 \rightarrow \texttt{Project.leader} \rightarrow \texttt{t}_{10} \rightarrow Ferrucci \\ \ldots \end{array}\right)$$

Figure 7: Clusters of data paths for $Q_2 = \{Brown, Ferrucci\}$

connecting the keywords of Q_1 is $\{t_2, t_3, t_{13}, t_{14}, t_{15}, t_{16}\}$, i.e. corresponding to the two rows in the table of Figure 6. Once analysed all the tuples in the set, the filters (`department` like '$\%CS\%$') and (`sname` like '$\%Java\%$') will reduce the result to the final answer $\{t_3, t_{14}, t_{16}\}$, i.e. the second row in the table of Figure 6.

Forward Exploration. In the Algorithm 27, the backward exploration stops when the selection $\sigma_{a=v}(R)$ retrieves more than one tuple (lines 18-21). In this case we would need to fork the exploration for each retrieved result: we could trigger a large number of branches and consequently explore all the database $\mathbf{d}$ more times, similarly to schema free approaches. Therefore, starting from the information captured by the backward exploration, we use a *forward* strategy. Forward navigation has been already exploited in data graph algorithms [13, 16] to improve backward explorations individuating connections from potential root nodes to keyword nodes. Similarly, our forward exploration supports the backward navigation, still preserving our competitive advantages: it does not require to keep extra information of the exploration and it only exploits *selection* (σ) and *projection* (π) operations. When the backward exploration identifies more than one tuple in a selection ($|\sigma_{a=v}(R)| > 1$), it does not insert any tid into $\mathcal{P}$, but it determines a condition γ in terms of a triple $\langle R, A, v\rangle$ (line 19). The condition says that a tuple in the relation R having the data value v associated to the attribute A is desired. All the conditions are kept in a set $\mathcal{C}_d$ (line 20). For instance, let us consider a second query $Q_2 = \{Brown, Ferrucci\}$. In this case we would retrieve information about *Brown* and how he is related to *Ferrucci*. Following the Algorithm 1, we obtain the two clusters depicted in Figure 7. In this case the first desired answer should be $S_1 = \{t_2, t_6, t_{10}\}$, i.e. *Brown* works in the *CS* department and he works in the *Watson* project with id *cs34* whose director is *Ferrucci*. In Figure 8 we depict the backward exploration at work to process the query Q_2.



(a) (b)

Figure 8: Backward exploration at work for Q_2

The first combination c useful to generate S_1 is (dp_i, dp_j). However, in this case the backward exploration is not able to provide an answer. Through the selection

$$\sigma_{employee=\text{“}Brown\text{”}}(WorksIn)$$

it is possible to instantiate the variables x_1 and x_2 with t_6 in dp_i, as shown in Figure 8.(a). In dp_j the variable x_4 is trivially instantiated with t_{10}, but the procedure stops when it tries to resolve the variable x_3. This is due to perform the selection $\sigma_{project=\text{“}cs34\text{”}}(WorksIn)$, resulting more than one tid, i.e. t_6 and t_7. At the end of the backward exploration we have $\mathcal{P} = \{(t_2, 1), (t_6, 1), (t_{10}, 1)\}$ and the condition $\gamma_1 = \langle WorksIn, project, \text{“}cs34\text{”}\rangle$ in the set $\mathcal{C}_d$. To retrieve S_1 the forward exploration has to disambiguate between t_6 and t_7.

As shown in the Algorithm 18, the forward exploration exploits the conditions in $\mathcal{C}_d$ retrieved in the backward exploration. The procedure performs two steps: *projection* and (in case) *selection*. In both steps, we explore data paths in forward since we start from a relation R of $\mathbf{d}$, we reach the data value v through the attribute a and we find the corresponding tid t_y (possibly) linked to the tuples retrieved from the backward exploration. Starting from the set $\mathcal{P}$, the first step to disambiguate the conditions in $\mathcal{C}_d$ is the *projection*. It checks if a condition γ in $\mathcal{C}_d$ can be satisfied ($\models$) by a tid t_y in $\mathcal{P}$, denoted by $t_y \models \gamma$ (line 3). If t_y satisfies the condition, we can increase the value of t_y in $\mathcal{P}$ (line 4) and probably find the answer (lines 5-6). Given the condition $\langle R, A, v\rangle$, the test $t_y \models \gamma$ is verified by checking $(\pi_a(t) = v) \wedge (t \in R)$, where $t_y = \text{idx}(t)$. For instance, recalling the condition $\gamma_1 = \langle WorksIn, project, \text{“}cs34\text{”}\rangle$, we have to check if there exists a tid in $\mathcal{P}$ belonging to the relation `WorksIn` and providing the data value *cs34* associated to the attribute `project`: in this case $t_6 \models \gamma_1$ since $\pi_{project}(t) = \text{“}cs34\text{”}$ and $t \in WorksIn$, i.e. $t_6 = \text{idx}(t)$. Since t_7 is not in $\mathcal{P}$, we do not consider it. Incrementing the value associated to t_6 in $\mathcal{P}$ we obtain the pair $(t_6, 2)$, i.e. we find the first answer $S_1 = \{t_2, t_6, t_{10}\}$.

The *projection* step could fail: a single condition γ is not able to disambiguate tuples, i.e. $\nexists t_y \in \mathcal{P} : t_y \models \gamma$. In this case we have to retrieve new tids from $\mathbf{d}$. Therefore the forward exploration provides the *selection* step. In $\mathcal{C}_d$, we search multiple conditions involving the same relation R, i.e. $\gamma_1 = \langle R, A_1, v_1\rangle$, $\gamma_2 = \langle R, A_2, v_2\rangle$, ..., $\gamma_n = \langle R, A_n, v_n\rangle$. Then we check if multiple conditions (lines 10-11) on the same relation R can retrieve a new tid t_y in R (line 13), i.e. we check if $t_y \in \text{idx}(\sigma_{(A_1=v_1)\wedge\ldots\wedge(A_n=v_n)}(R))$. The procedure `AND` is responsible for computing $cond := (a_1 = v_1) \wedge \ldots \wedge (a_n = v_n)$ from the set Γ of conditions involving the same relation R (line 12). If the test is satisfied we increment the value associated to t_y in $\mathcal{P}$ (line 15) and probably find the answer, i.e. the occurrence of t_y is equal to $|Q|$ (line 16).

Algorithm 4: Forward Exploration

Input : The map $\mathcal{P}$, the set $\mathcal{C}_d$.
Output: A boolean value.

```
// projection step
foreach γ ∈ C_d do
    if ∃t_y ∈ P : t_y ⊨ γ then
        P.put(t_y, P.get(t_y)+1);
        if P.get(t_y) = |Q| then
            return true;
// selection step
foreach R ∈ R do
    Γ ← ∅;
    foreach γ =< R', A', v' >∈ C_d : R = R' do
        Γ ← Γ ∪ {γ};
    cond ← AND(Γ);
    if |Γ| > |Q| and |σ_cond(R)| = 1 then
        t_y ← idx(σ_cond(R));
        P.put(t_y, P.get(t_y)+1);
        if P.get(t_y) = |Q| then
            return true;
return false;
```

4 General results

In this section we investigate the general properties of our approach. First of all, we show that, in the worst case, the computational complexity of our algorithm is linear in the number of tuples that match the given keywords. Then, we show that query answering is monotonic, that is, our technique is able to retrieve the top-k solutions in the first k iterations.

Computational Complexity.

Given a database **d**and a query Q, let $|Q|$ be the number of keywords in Q, T the number of tuples in **d**matching a keyword in Q, and $|\mathcal{R}|$ the number of relations in **d**.

We first note that the longest possible data path in the set of clusters $\mathcal{CL}$ has two nodes for each table and so it is linear in $|\mathcal{R}|$. We also note that the number d of data paths depends on the number of schema paths. By construction, the schema paths are at most $2^{|\mathcal{R}|}$ because every attribute node A of a relation R has at most two incoming edges from R. Consequently, we have that, in the worst case, $d = T \times 2^{|\mathcal{R}|}$ even if, in real world database schemas, the schema paths are much lower. Since it is quite natural to assume that $|\mathcal{R}|$ is fixed, we can assume that d is linear in T.

Now, the overall computational complexity is given by $O(\mathit{Clustering}) + O(\mathit{Building})$. In the Algorithm 1 (Clustering phase) every data path is inserted within one cluster. Since clusters are kept ordered according to the size of the data paths, we can implement efficiently a cluster as an ordered queue, which makes the clustering phase linear in the

number of inserted paths, i.e. $O(Clustering) \in O(d)$.

The complexity of Algorithm 23 (Building phase) is affected by the number of $|C|$ combinations of data paths that need to be computed. The worst case is when the data paths are equally distributed among the clusters. Therefore, the number of combinations is $O(|C|) \in \frac{O(d)}{O(|Q|) \times O(|\mathcal{R}|)}$. For every combination c, we need to compute $|Q|$ backward explorations (one for each data path) and, possibly, one forward exploration. Hence, we have that $O(Building) \in O(|C|) \times (O(Forward\_Exploration) + O(Backward\_Exploration) \times O(|Q|))$. The number of conditions in $\mathcal{C}_d$ are $|Q|$ at most, while the number of tuples in $\mathcal{P}$ is limited by the lengths of the data paths, i.e. $|\mathcal{R}| \times |Q|$ in the worst case. In the forward exploration, for every condition, the *projection step* of the algorithm executes a projection over every tuple in $\mathcal{P}$ and is therefore bounded by $O(|\mathcal{R}| \times |Q|)$. The *selection step* groups the conditions w.r.t. a common relation and computes a selection for each group. This can be done in $O(|Q|)$. Hence, the overall complexity of the forward exploration is $O(|\mathcal{R}| \times |Q|) + O(|Q|) \in O(|\mathcal{R}| \times |Q|)$. Since we can assume that, exploiting the indexes defined in $\mathbf{d}$, the execution of projections and selections made during this step requires constant time, we have that the complexity of the backward exploration is just given by the length of a data path, that is, it is $O(|\mathcal{R}|)$.

$$O(Building) \in \frac{O(d)}{O(|Q|) \times O(|\mathcal{R}|)} \times (O(|\mathcal{R}| \times |Q|) + O(|\mathcal{R}|) \times O(|Q|)) \in O(d)$$

Finally the overall process $O(Clustering) + O(Building)$ is $O(d) + O(d)$ and since the number of data paths is linear in T, it follows that our technique is linear in the size of the database.

Monotonicity. The basic intuition is that when a data path dp_i is analyzed, any data path dp_j contained in dp_i has been already analyzed in previous steps. Therefore an instance of dp_i retrieves more tuples (precisely a superset) than any instance of dp_j. Here we sketch a proof to demonstrate the monotonicity of the approach. The proof proceeds by induction on the number of iterations of the building algorithm. At the first iteration (the base case) the data paths contain one tuple and therefore the answers consist of single tuples containing all the keywords. In the rest of iterations the data paths involve more tuples: at the i-th iteration each instance of these paths contains at most i tuples. In an answer there are $|Q|$ data paths that share at least one node. Therefore, an answer contains $(i-1) \times |Q|$ tuples. At the $(i+1)$-th iteration, the tuples in an answer are $(i+1-1) \times |Q| = i \times |Q|$ that is always greater than $(i-1) \times |Q|$. This shows the monotonicity of our approach.

5 Experimental Results

We developed our approach in YaaniiR, a system for keyword search over relational databases. YaaniiR is implemented entirely with a procedural language for SQL. In particular PL/pgSQL since we used PostgreSQL 9.1 as RDBMS. In our experiments we used the only available benchmark, which is provided by Coffman et al. [6]. It satisfies criteria and issues [5, 25] from the research community to standardize the evaluation of keyword search techniques. In [6], by comparing the state-of-the-art keyword search systems, the authors provide a standardized evaluation on three datasets of different size and complexity: IMDb (1,67 million tuples and 6 relations), Wikipedia (206.318 tuples and 6 relations), and a third ideal counterpoint (due to its smaller size), Mondial (17.115

tuples and 28 relations). For each dataset, we run the set of 50 queries (see [6] for details and statistics). Experiments were conducted on a dual core 2.66GHz Intel Xeon, running Linux RedHat, with 4 GB of memory, 6 MB cache, and a 2-disk 1TB striped RAID array, and we used PostgreSQL 9.1 as RDBMS. We remark that we keep schema and instance of all datasets.

Implementation. The implementation plays an essential role in our framework. Here we provide some technical details in order to show the feasibility to implement keyword-based search functionality in a RDBMS and consequently to introduce an SQL keyword search operator. We implemented the algorithms of the paper by using only a procedural language for SQL and the RDBMS data structures. Similarly to all the approaches we employ inverted indices and full-text queries to have direct access to the tuples of interest. Modern RDBMSs already integrate general purpose full-text indices and related query operators. In some case they can be customized by the DB administrator and applied on a limited number of attributes, i.e. usually the attributes relevant to the user or containing text data. We implement schema and data paths as integer arrays, i.e. text values are encoded by hash functions provided by the RDBMS. Each element of the array corresponds to a node in the path. Schema paths are retrieved by the computation of the metadata (schema) of $\mathbf{d}$. The management of tuple-ids is already implemented in many RDBMSs. In our case, we use the PostgreSQL clause `WITH OIDS` updating the definition of a table, in case. It creates a column named `OID` containing the identifiers of the tuples. Each cluster is in practice a priority queue where the priority decreases with the increasing length of a path. A cluster is implemented with a table, having the length of the paths as indexed attribute. All the loops of the algorithms are supported by the definition and usage of cursors. In our implementation we apply a straightforward cache mechanism for the tuples. In the cache we trace the already accessed tuples. So before executing an access to the disk we search within the cache. In this way a tuple is accessed only once. Such simple mechanism speeds-up significantly the execution time.

Our algorithms have been implemented in terms of PL/pgSQL procedures to add in $\mathbf{d}$. Such procedures exploit a simple index based on the permanent table SG(*attribute,path*) and the procedure DG. The former stores all schema paths while the latter retrieve all data paths at runtime. In SG, *path* implements a schema path in terms of an array of hash numbers (i.e. hashing of table and attributes names in the schema) while *attribute* is the value of the ending node of the path implemented as a hash value (i.e. on *attribute* we define a B-tree index). An efficient implementation of a BFS traversal supports the computation of all schema paths (i.e. we compute all paths between tables and attributes, not only the shortest ones). The procedure DG, similarly, implements a data path in terms of an array of hash numbers and defines a `tsvector` value on all text attributes of $\mathbf{d}$ on which imposes a GIN index for full-text search. Such pre-configuration (e.g., the building of the SG table) is built efficiently: from few milliseconds on Mondial to a couple of minutes on IMDb and Wikipedia. The last datasets, i.e. IMDb and Wikipedia, present 516MB and 550 MB of size, respectively. The resulting index increases the starting data size of few MBs.

Performance. For query execution evaluation, we compared our system (YaaniiR), with the most related schema-free approaches: SPARK [20], EASE [18], and Blinks [13], DPBF [10], DISCOVER [15] and the refined version DISCOVER-II [14]. Moreover we made a comparison with schema-based approaches: Power [22], using all the algorithms under the three semantics – connected tree (CT), distinct core semantics (DC), distinct

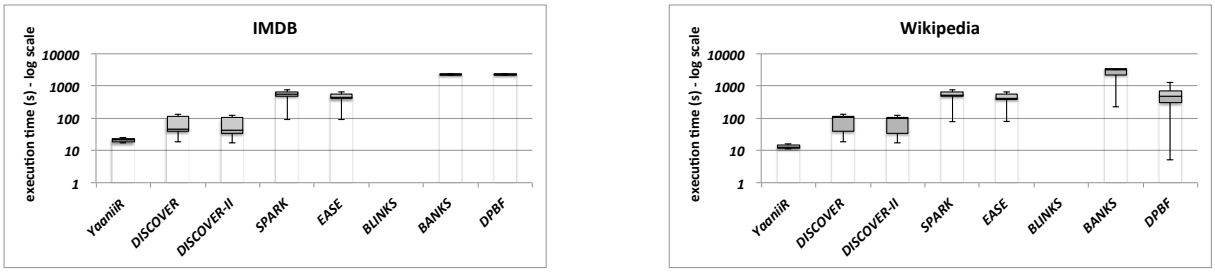


Figure 9: Performance comparison with schema-free approaches

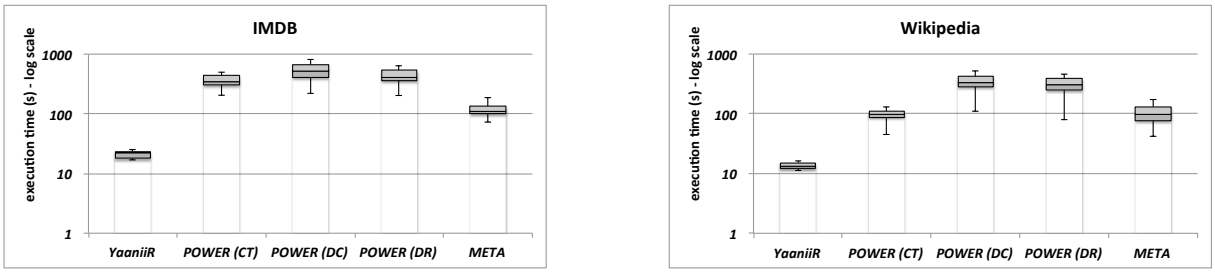


Figure 10: Performance comparison with schema-based approaches

root semantics (DR)[1] – and META [3].

We evaluated the execution time that is the time elapsed from issuing a query until an algorithm terminates. Such execution computes the top-100 answers. We performed *cold-cache* experiments (by dropping all file-system caches before restarting the systems and running the queries) and *warm-cache* experiments (without dropping the caches). We repeated all the tests three times and measured the mean execution times. For space constraints, we report only cold-cache experiments, but warm-cache experiments follow a similar trend. As in [6], we imposed a maximum execution time of 1 hour for each technique (stopping the execution and denoting a timeout exception). Moreover we allowed ≈5 GB of virtual memory and limit the size of answers to 5 tuples.

Figure 9 and Figure 10 show box plots of the execution times for all queries on each dataset w.r.t schema-free approaches and schema-based approaches, respectively. In general our system outperforms consistently all approaches. In particular the range in execution times for schema-free approaches is often several orders of magnitude: the performance of these heuristics varies considerably (i.e. the evaluation of the mean execution time cannot report such behavior). In the figures, we do not report box plots for BLINKS since it always required more than one hour or encountered an OutOfMemoryError. Similarly, DISCOVER, BANKS, DPBF failed many queries due to time out exception. SPARK and EASE perform worse but they completed most of the queries. Our system completed all 50 queries in each dataset without computing useless answers or set of tuples to combine. This is due to our incremental strategy reducing the space overhead and consequently the time complexity of the overall process w.r.t. the competitors that spend much time traversing a large number of tuples (nodes) and computing and ranking the candidates to be (in case) answers. In particular Figure 11 shows how much backward and forward strategies are involved in the execution time. In all datasets, most of the execution time is spent by the backward strategy: the forward navigation refines and

[1]We refer to the most efficient version of both DC and DR

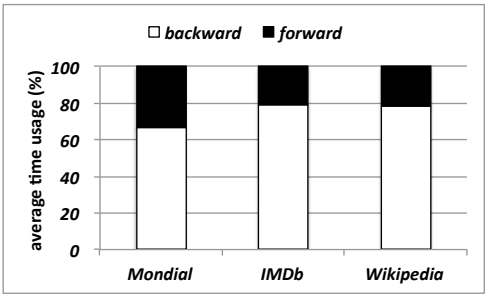


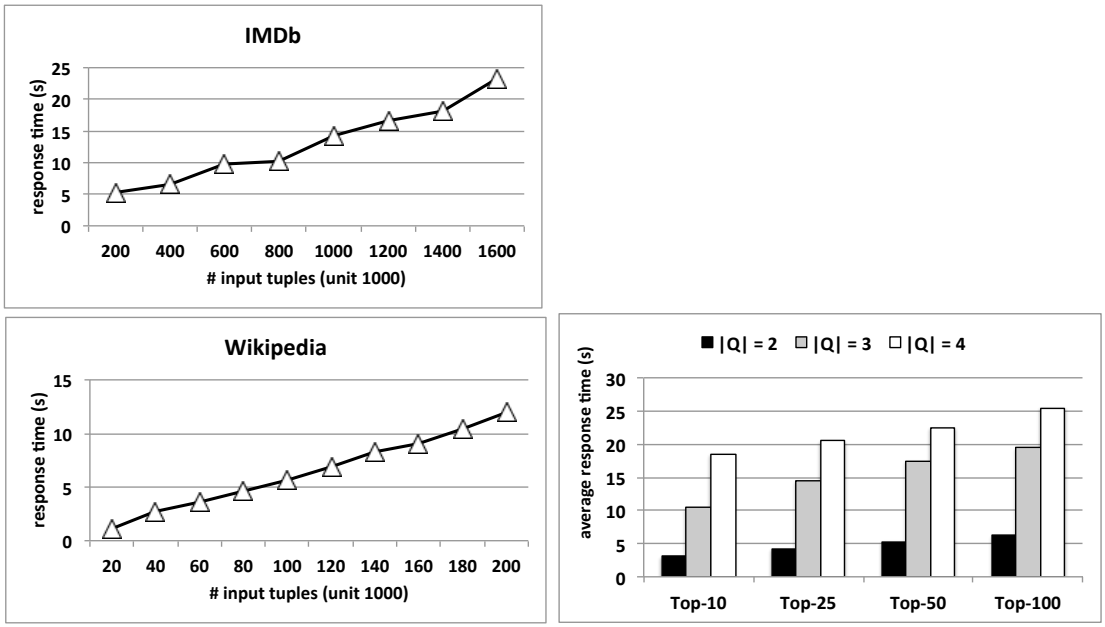
Figure 11: Average time usage of backward and forward in query execution

complete the work of the backward. In MONDIAL the forward is more involved due to the greater number of relations of the dataset (i.e. we have longer paths).

With respect to schema-based approaches, we implemented the three algorithms of POWER in Java 1.6 and JDBC to connect to PostgreSQL. In particular we used the same parameters for IMDB testing as described in [22] for all datasets. On the other hand, we used the implementation of META offered by the same authors. Also in this case, the results confirm the significant speed-up of our approach with respect to the others. In this case the number of tuples generated by the join operations is effective to generate the answers of interest, i.e. the cost to evaluate each candidate network is limited. The DC and DR algorithms perform worse due to the more complex technique to evaluate the candidate networks. In some queries, a larger number of keywords in Q increases the complexity to evaluate a candidate network and consequently the number of tuples to evaluate. In this context the CT algorithm and META are comparable while our system performs significantly better due to the lowest (or missing) overhead introduced in our incremental strategy. However schema-based approaches completed all 50 queries in each dataset and provide a more regular behavior in the execution time.

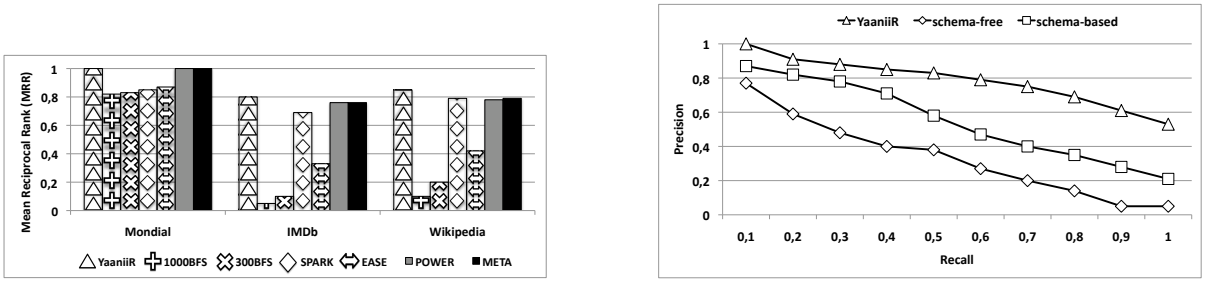
An evaluation of the scalability of our system is reported in Figure 12. In particular Figure 12.(a) reports the scalability of YAANIIR on both IMDB and WIKIPEDIA. It shows the average search time (s) to execute a query w.r.t. the database size, i.e. the number of tuples. The diagram confirms the studies of complexity in Section 3. Moreover we enriched such experiment by introducing also scalability with respect to the the average size of the query (i.e. $|Q|$), that is the number of keywords, as shown in Figure 12.(b). In particular we evaluate the impact of the number of keywords to find the top-k (i.e. k $\in \{10, 25, 50, 100\}$) answers. Also in this case the time grows linearly. The impact of query length is relevant when a higher k is used.

Effectiveness. We have also evaluated the effectiveness of results. Figure 13.(a) shows the mean reciprocal rank (MRR) of the queries for each system in any dataset. In particular we considered the average MRR over all the three algorithms of POWER. Due to the small size, all systems show comparable performance on the MONDIAL dataset. On the contrary, we have different results on IMDB and WIKIPEDIA. As we expected, BLINKS and EASE perform poorly on this task since they implement a proximity search strategy where the ranking is not able to distinguish answers containing a single node. SPARK performs well in average because it exploits an IR ranking strategy: usually IR-style search systems prefer larger results supporting the disambiguation of search terms. Both POWER and META introduce *noise* due to empty results or too exhaustive answers to the detriment of the specificity. Our strategy outperforms significantly all the others: this strategy is



(a) (b)

Figure 12: (a) Scalability w.r.t. #tuples and (b) Average Response Times w.r.t. the average size of Q



(a) (b)

Figure 13: MRR (a) and Precision-Recall curves (b)

able to return first the most relevant answer (i.e. MRR $\in [0.8, 1]$) in any case. Then, we measured the interpolation between precision and recall to find the top-10 answers, on the queries on all datasets. We compare our curve with the interpolated precision curves averaged over both schema-free and schema-based approaches. Figure 13.(b) shows the results. As to be expected, the precision of the other systems dramatically decreases for large values of recall. The overhead introduced by all competitors damages the quality of the results. On the contrary our strategies keeps values on the range [0.6,0.9]. Such result confirms the discussion of Section 3, that is the feasibility of our system that produces the top-k answers in linear time.

6 Related Work

The problem of keyword search over structured data arose a decade ago [1, 4], when it was clear that search engines were imposing a "de-facto" standard to the way to look for any

kind of information. From then, a lot of work has been done in this field. The common assumption made by the various proposals to keyword search over relational databases is that an answer is a *joining tuple tree* [15] (JTT) in which the nodes represent tuples and the edges represent references between them, according to the foreign keys defined on the database schema. Usually, it is assumed that a JTT must have a limited size. In particular, in the *distinct root semantics* a JTT is a tree with a certain radius (distance of nodes from the root) and each answer is identified by a unique tuple (the root of the tree) [9, 13, 18, 22]. Conversely, the *distinct core semantics* assumes that a JTT is a multi-center graphs, called communities [23]. The various approaches to keyword-based query answering are commonly classified into two categories, *schema-based* and *schema-free*, even if some recent works have questioned the state of the art and suggested alternative techniques to solve the problem. We discuss all of them in order.

Schema-based approaches. Schema-based approaches [1, 14, 15, 20, 21] make use, in a preliminary phase, of the database schema to build trees called *candidate networks* (CNs) whose nodes represent subsets of the tuples in a relation. CNs must be *complete* (i.e., involving all the keywords in the query) and *duplicate-free*. Duplicate elimination relies on graph isomorphisms, which requires a high computational cost. For this reason, in [21] the authors have proposed an approach to CN duplicate elimination that does not rely on graph isomorphism. CNs are then evaluated by means of a (possible large) number of SQL queries that, once submitted to the RDBMS, return the final JTTs. Unfortunately, it has been shown that finding the best execution plan from a set of CNs is an NP-Complete problem [15]. Moreover, empty results can occur and this can make the process inefficient and introduce noise in the final result. The technique proposed in [14] is based on a IR-style technique for the construction of JTTs. The score of an answer is evaluated by aggregating the scores of the tuples that matches the given keyword and by normalizing the result on the basis of the size of the tree. The approach in [20] is similar but candidate answers are monolithic documents assessed by a standard IR function. Our approach fits in this category in that we take advantage from database schema and constraints to build the data paths (see Definition 4) without accessing the database. However, we use those data path for building answers progressively, in order of relevance. This eliminates the need to compare answers and allows us to return the results as soon as they are built.

Schema-free approaches. Schema-free approaches [4, 9, 10, 12, 16, 17, 18] first build a graph-based representation $\mathcal{G}$ of the database in which the nodes of $\mathcal{G}$ represent the tuples of the database and its edges represent primary or foreign key constraints. Then, they make use of graph algorithms and graph exploration techniques to select the subgraphs of $\mathcal{G}$ that connect nodes matching the keywords of the query. Usually, apart from [9], all of them materialize $\mathcal{G}$ in main memory, which is clearly hard to scale. Query evaluation usually consists in finding a set of (minimal) *Steiner trees* [11] of $\mathcal{G}$. This problem is known to be NP-Complete [11]. Therefore, the various proposals rely on complex heuristics aimed at generating approximations of Steiner trees. In [4, 16], the authors propose backward and bidirectional graph exploration techniques that have polynomial data complexity. In [17], the authors show that the top-k answers in an approximate order can be computed linearly in k and polynomially in the size of the input. A practical implementation of this approach is illustrated in [12]. The approach in [10] computes approximate top-k answers making use of dynamic programming. Another approximate approach is provided by [18]: it is based on pre-computing and indexing all the maximal *r-radius* sub-graphs on disk. Then, *r-radius Steiner graphs* (a variant of Steiner trees) are computed by pruning the

maximal r-radius sub-graphs. We actually took inspiration from these approaches by modeling the problem in terms of graph search. However, we do not build in-memory graph-based structures and resort on a simple technique for building the answers that is linear in the size of the database and does not require complex graph algorithms of high computational cost.

New approaches. As observed by several authors (e.g., [2, 6, 7]), the solutions proposed so far are not efficient and reliable enough for a spread usage. Indeed, it should be mentioned that none of them has been implemented in a commercial system. The authors in [22] argue that the main drawback of existing approaches is the limited use of the functionality of the RDBMS in which data is stored. Along this line, Bergamaschi et al. [3] have proposed an approach that considers the metadata of the RDBMS and applies the Hungarian algorithm to generate the SQL queries needed to retrieve candidate answers. The work in [2] proposes to compute the answers within a time limit and to show to the user the unexplored part of the database, so that she can refine the results. We have indeed followed this clue in that our approach only relies on the capabilities of the underlying RDBMS.

7 Conclusion and Future Work

In this paper, we presented a novel approach to keyword search query over relational databases, by providing a linear strategy for top-k query answering. Such strategy enables the search to scale seamlessly with the size of the input. Experimental results confirmed our algorithms and the advantage over other approaches. This work now opens several directions of further research. From a theoretical point of view, we are investigating algorithms to keyword search over distributed environments, retaining the results achieved in this paper. From a practical point of view, we are widening optimization techniques to speed-up the query evaluation and to improve the effectiveness of the result, implementing an SQL operator.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[2] Akanksha Baid, Ian Rae, Jiexing Li, AnHai Doan, and Jeffrey F. Naughton. Toward scalable keyword search over relational data. *PVLDB*, 3(1):140–149, 2010.

[3] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Raquel Trillo Lado, and Yannis Velegrakis. Keyword search over relational databases: a metadata approach. In *SIGMOD*, pages 565–576, 2011.

[4] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.

[5] Yi Chen, Wei Wang 0011, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In *SIGMOD*, pages 1005–1010, 2009.

[6] Joel Coffman and Alfred Weaver. An empirical performance evaluation of relational keyword search techniques. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints):1, 2012.

[7] Joel Coffman and Alfred C. Weaver. A framework for evaluating database keyword search strategies. In *CIKM*, pages 729–738, 2010.

[8] Joel Coffman and Alfred C. Weaver. Learning to rank results in relational keyword search. In *CIKM*, pages 1689–1698, 2011.

[9] Bhavana Bharat Dalvi, Meghana Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *VLDB*, 1(1):1189–1204, 2008.

[10] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[11] M. R. Garey, R. L. Graham, and D. S. Johnson. The complexity of computing Steiner minimal trees. *SIAM Journal on Applied Mathematics*, 32(4):835–859, 1977.

[12] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, pages 927–940, 2008.

[13] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[14] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[15] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[16] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[17] Benny Kimelfeld and Yehoshua Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.

[18] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[19] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[20] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, pages 115–126, 2007.

[21] Alexander Markowetz, Yin Yang, and Dimitris Papadias. Keyword search on relational data streams. In *SIGMOD*, pages 605–616, 2007.

[22] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD*, pages 681–694, 2009.

[23] Lu Qin, Jeffrey Xu Yu, Lijun Chang, and Yufei Tao. Querying communities in relational databases. In *ICDE*, pages 724–735, 2009.

[24] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data(base) Engineering Bulletin*, 24(4):35–43, 2001.

[25] William Webber. Evaluating the effectiveness of keyword search. *IEEE Data Eng. Bull.*, 33(1):54–59, 2010.