Graph-driven Exploration of Relational Databases for Efficient Keyword Search

Roberto De Virgilio
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
dvr@dia.uniroma3.it

Antonio Maccioni
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
maccioni@dia.uniroma3.it

Riccardo Torlone
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
torlone@dia.uniroma3.it

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 by representing relational data as graph and by combining progressively 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, at the same time, 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, because the vast majority of users has little or no familiarity with computer technology. This need has originated a large set of proposals of non-conventional methods for accessing structured and semi-structured data. Among them, several studies have focused on the adoption of a keyword-based strategy for retrieving information stored in relational databases, with the goal of freeing the users from the knowledge of query languages and/or the organization of data [12, 13, 15].

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* $\{t_3, t_{14}, t_{16}\}$, *which involve the given keywords.*

R_1:Employee

	ename	department
t_1	Zuckerberg	CS
t_2	Brown	CS
t_3	Lee	CS
t_4	Ferrucci	IE

R_2: WorksIn

	employee	project
t_5	Zuckerberg	x123
t_6	Brown	cs34
t_7	Lee	cs34
t_8	Ferrucci	m111

R_3: Project

	id	pname	leader
t_9	x123	Facebook	Zuckerberg
t_{10}	cs34	Watson	Ferrucci
t_{11}	ee67	LOD	Lee
t_{12}	m111	DeepQA	Ferrucci

R_4: SkilledIn

	person	skill
t_{13}	Brown	Algorithms
t_{14}	Lee	Java

R_5: Skill

	sname	type
t_{15}	Algorithms	theoretical
t_{16}	Java	technical

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

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, (ii) ranking of the answers according to some relevance criteria, and (iii) only the top-k answers are selected and 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* [2, 16, 17, 19] and *schema-free* [13, 14, 6]. 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 inspira-

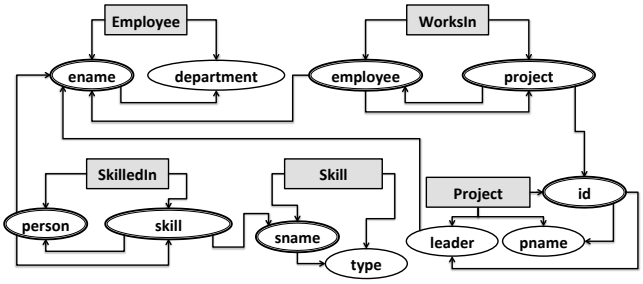

Figure 2: An example of schema graph $\mathcal{SG}$

tion 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. A relevant feature of our approach is that, as suggested in [18], it 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, 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 tool for keyword-based search over relational databases that implements the technique described in this paper. This tool has been used to perform several experiments over an available benchmark [4] that have shown a marked improvement over other approaches in terms of both effectiveness and efficiency.

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. The experimental results are reported in Section 4 and, in Section 5, we discuss related works. Finally, in Section 6, 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 **d** in terms of a pair of graphs $\langle \mathcal{SG}, \mathcal{DG} \rangle$ representing the schema and the instance of **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.*

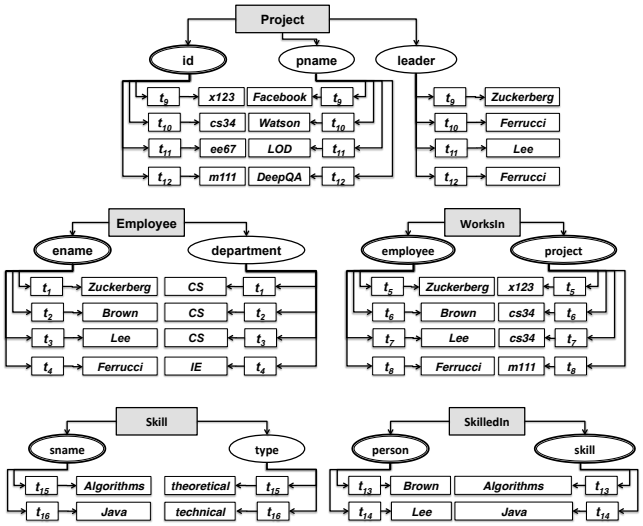

Figure 3: An example of data graph $\mathcal{DG}$

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.

$[cl_{Java}]$:

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

$[cl_{CS}]$:

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

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

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:

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

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.*

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) [11].

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 [5]. This is captured by a scoring function that simply returns cardinality of S.

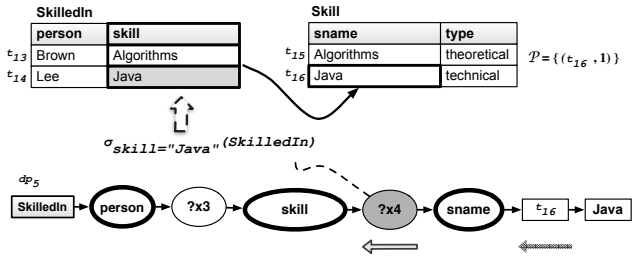

Figure 5: Backward exploration at work for dp_5 (selection)

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

Given a keyword-based query Q, our technique consists of two main phases, *clustering* and *building*. They guarantee a monotonic construction of the answers (i.e. the answer generated in 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.

3.1 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. There is one cluster for each keyword $q_i \in Q$. In particular, we start from each data path $R \rightarrow A \rightarrow tid \rightarrow v$ such that $q_i \approx v$. Then we generate the data paths following the route of each schema path sp ending into the attribute A. 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.

3.2 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

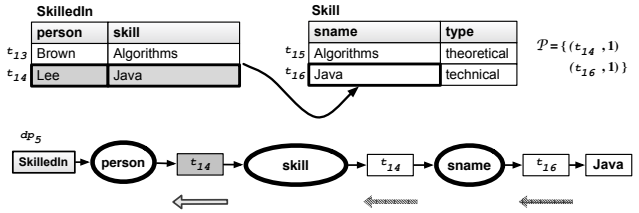

Figure 6: Backward exploration at work for dp_5 (projection)

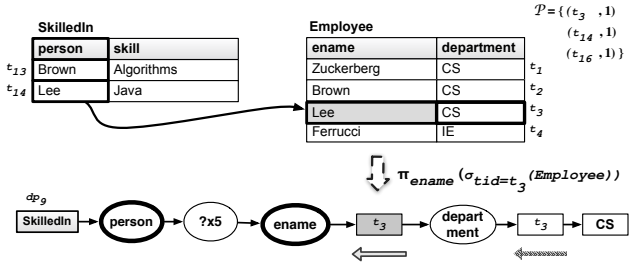

Figure 7: Backward exploration at work for dp_9 (projection)

an answer. The search proceeds in this way with longer data paths that follow in the clusters. In detail, following the Algorithm 1, 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 of answers. 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 with the clusters in Figure 2, 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, through the procedure `validCombinations`, before combining paths if all those paths cross a common table. 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

Algorithm 1: 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;
```

set $\mathcal{S}$. This means to instantiate a set set of data paths $DP = \{dp_1, \ldots, dp_n\}$ and verifying if the results have a tuple in common. This evaluation is performed by the procedures `backward_exploration` and `forward_exploration`, as follows. 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. Each data path dp of a combination is analysed independently from the others, i.e. in our example dp_5 and dp_9. They are navigated backward starting from the last node. In other terms we follow the

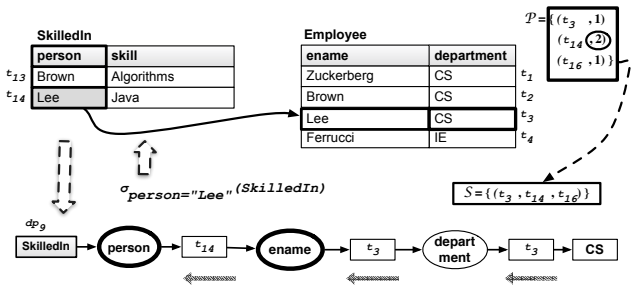

Figure 8: Backward exploration at work for dp_9 (selection)

$[cl_{Brown}]$:

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

$[cl_{Ferrucci}]$:

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

Figure 9: Clusters of data paths for $Q_2 = \{Brown, Ferrucci\}$

foreign and primary key constraints contrariwise. The `backward_exploration` procedure takes as input a data path dp to analyse, the query Q, a map $\mathcal{P}$ and a set $\mathcal{C}_d$ of conditions, whose functionality will be described in the forward exploration. Given dp_5, we start from the node *Java*, we meet the tid t_{16} and the algorithm updates $\mathcal{P}$ inserting the pair $\{t_{16}, 1\}$. Then, we proceed until the variable x_4 is encountered (Figure 5).

According to the information carried by this data path, the only possible substitution for x_4 is t_{14}, that is the tid of the tuple that has as `SkilledIn.skill` the same value occurring in `Skill.sname` of the tuple with tid t_{16}, i.e. *Java*. In this case we are following a foreign key constraint and we extract the new tid by a simple selection. As shown in Figure 6, 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. In this case we are following a primary key constraint and the procedure extracts the data value associated to the attribute A of the same tuple with a simple projection. The exploration of dp_5 terminates. Similarly we explore dp_9. In Figure 7, we start from the data value *CS*, we insert t_3 in $\mathcal{P}$ and then we meet the variable x_6. It belongs to the same relation `Employee` of t_3. Therefore x_6 corresponds to t_3 and it is extracted by a projection.

Finally, we meet the variable x_5 as depicted in Figure 8. Since we are following a foreign key constraint, we execute the selection $\sigma_{person="Lee"}(SkilledIn)$ and we retrieve the tid t_{14}. 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: in this case the tids in $\mathcal{P}$ represent an answer to insert in $\mathcal{S}$; in our example we have the answer $\{t_3, t_{14}, t_{16}\}$.

Forward Exploration. If in the backward exploration we find a multiple substitutions for some variable the analysis of the current data path stops. 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 **d** more times, similarly to schema free approaches.

Therefore, the backward exploration determines a condition γ in terms of a triple $\langle R, A, v \rangle$. 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 $\mathcal{C}_d$. Starting from the information captured by the conditions we use a *forward* strategy, where data paths are

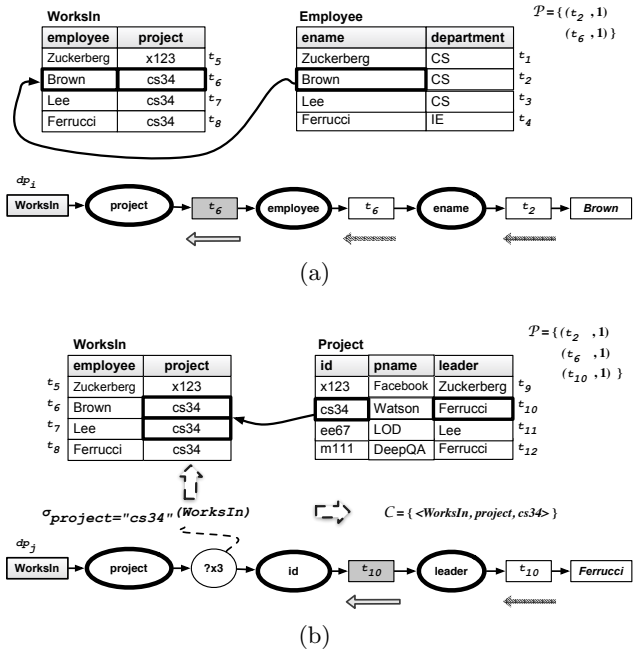

Figure 10: Backward exploration at work for Q_2

navigated forward using all the tids retrieved in the first step as substitutions for the remaining variables.

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*. We obtain the two clusters depicted in Figure 9. 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 10 we depict the backward exploration at work to process the query 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

$$\sigma_{employee="Brown"}(WorksIn)$$

it is possible to instantiate the variables x_1 and x_2 with t_6 in dp_i, as shown in Figure 10.(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 $\sigma_{project="cs34"}(WorksIn)$, 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 $\gamma_1 = \langle WorksIn, project, "cs34" \rangle$ in the set $\mathcal{C}_d$. To retrieve S_1 the forward exploration has to disambiguate between t_6 and t_7. 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 $S_1 = \{t_2, t_6, t_{10}\}$.

In general, 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 **d**. Therefore the forward exploration provides the *selection* step. In $\mathcal{C}_d$, we search multiple conditions involving the same relation R, i.e. $\gamma_1 = \langle R, A_1, v_1 \rangle$, $\gamma_2 = \langle R, A_2, v_2 \rangle$, ..., $\gamma_n = \langle R, A_n, v_n \rangle$, and then we check if these multiple conditions can retrieve a new tid t_y in R.

Note that the forward navigation has been already exploited in data graph algorithms [9, 12] 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.

4. 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. [4]. It satisfies criteria and issues [3, 20] from the research community to standardize the evaluation of keyword search techniques. In [4], 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 [4] 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

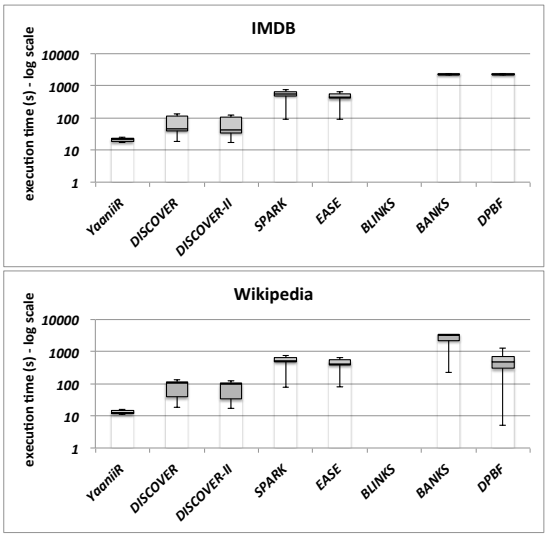

Figure 11: Performance comparison with schema-free approaches

in the path. Schema paths are retrieved by the computation of the metadata (schema) of **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 **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 **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

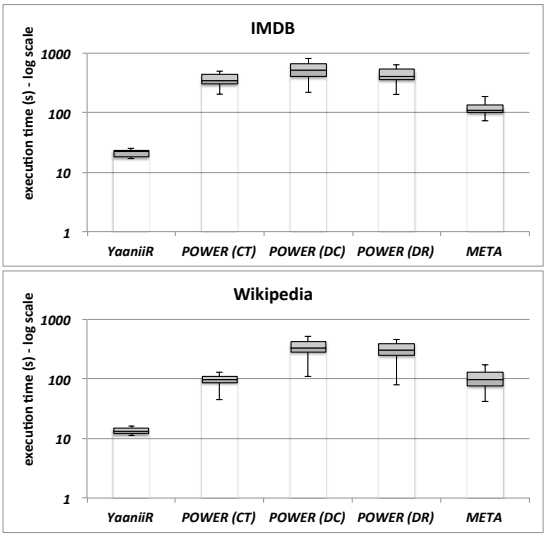

Figure 12: Performance comparison with schema-based approaches

schema-free approaches: SPARK [16], EASE [14], and BLINKS [9], DPBF [7], DISCOVER [11] and the refined version DISCOVER-II [10]. Moreover we made a comparison with schema-based approaches: POWER [18], using all the algorithms under the three semantics – connected tree (CT), distinct core semantics (DC), distinct root semantics (DR)[1] – and META [2].

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 [4], 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 11 and Figure 12 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

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

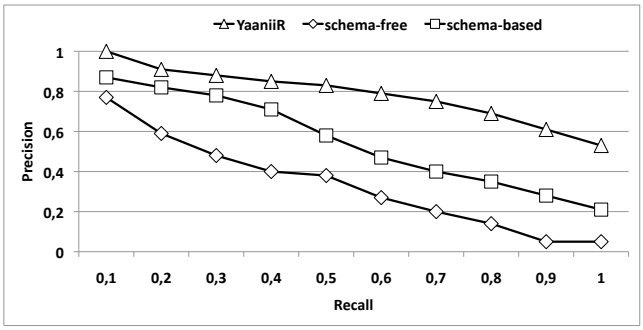

Figure 13: Precision-Recall curves

spend much time traversing a large number of tuples (nodes) and computing and ranking the candidates to be (in case) answers.

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 [18] 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.

Effectiveness. We have also evaluated the effectiveness of results. 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 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.

5. RELATED WORK

The common assumption made by the various proposals to keyword search over relational databases is that an answer is a *joining tuple tree*(JTT) in which the nodes represent tuples and the edges represent references between them, according to the foreign keys defined on the database schema. 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 [11, 16, 17] 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 [17] 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 [11]. Moreover, empty results can occur and this can make the process inefficient and introduce noise in the final result. 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.

Schema-free approaches. Schema-free approaches [6, 7, 13, 14] 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 [6], 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* [8] of $\mathcal{G}$. This problem is known to be NP-Complete [8]. Therefore, the various proposals rely on complex heuristics aimed at generating approximations of Steiner trees. 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., [1, 4]), 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 [18] argue that the main drawback of existing approaches is the limited use of the functionality of the RDBMS in which data is stored. The work in [1] 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.

6. 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.

7. REFERENCES

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

[2] S. Bergamaschi, E. Domnori, F. Guerra, R. T. Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *SIGMOD*, pages 565–576, 2011.

[3] Y. Chen, W. W. 0011, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD*, pages 1005–1010, 2009.

[4] J. Coffman and A. Weaver. An empirical performance evaluation of relational keyword search techniques. *TKDE*, 99(PrePrints):1, 2012.

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

[6] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *VLDB*, 1(1):1189–1204, 2008.

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

[8] 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.

[9] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[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*, pages 670–681, 2002.

[12] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

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

[14] 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*, pages 903–914, 2008.

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

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

[17] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIGMOD*, pages 605–616, 2007.

[18] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD*, pages 681–694, 2009.

[19] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *ICDE*, pages 724–735, 2009.

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