Keyword Search in Heterogeneous Data Sources

Felipe Cordeiro, Helena Galhardas, Julien Leblay, Ioana Manolescu, Tayeb Merabti

Keyword Search in Heterogeneous Data Sources

Felipe Cordeiro[1], Helena Galhardas[2], Julien Leblay[3], Ioana Manolescu[1], and Tayeb Merabti[1]

[1] Inria and Ecole Polytechnique, France, first.last@inria.fr
[2] INESC-ID and IST, Univ. Lisboa, Portugal, helena.galhardas@tecnico.ulisboa.pt
[3] AIST, Japan, julien.leblay@aist.go.jp

Abstract. Data journalism is the field of investigative journalism work based first and foremost on digital data. As more and more of human activity leaves strong digital traces, data journalism is an increasingly important trend. Important journalism projects increasingly involve diverse data sources, having heterogeneous data models, different structures, or no structure at all; the Offshore Leaks is a prime example.
Inspired by our collaboration with Le Monde, a leading French newspaper, we designed a novel content management architecture, together with an algorithm for exploiting such heterogeneous corpora through keyword search: given a set of search terms, find links between them within and across the different datasets which we interconnect in a graph. Our work recalls keyword search in structured and unstructured data, but data heterogeneity makes it computationally harder. We analyze the performance of our algorithm on real-life datasets.

Keywords: data journalism · keyword search · heterogeneous data sources

1 Introduction

Data journalists often have to handle sets of different data structures, which they obtain from official organizations or from their sources, extract from social media, receive via email or create themselves (typically Excel or Word-style) etc. For instance, journalists from Le Monde newspaper want to retrieve *connections between elected people at Assemblée Nationale and companies that have outposts outside of France*; such a query can be answered currently at a high human effort cost, by inspecting e.g. a JSON list of Assemblée elected officials (available from NosDeputes.fr) and manually connecting the names with those found in a national registry of companies. This huge effort may still miss connections that could be found if one added information about policians' and business people's spouses, information sometimes available in public knowledge bases such as DBPedia, or in a journalists' personal notes.

No single query language can be used on such heterogeneous data; instead, we study methods to query the corpus by specifying some keywords and asking for all the connections that exist, in one or across several datasources, between these keywords. This problem has been raised by our collaboration with Les Décodeurs, Le Monde's fact-checking team[4], with whom we collaborate within

[4] http://www.lemonde.fr/les-decodeurs/

the ContentCheck collaborative research project[5]. Our study is novel with respect to the state of the art (Section 7) as we are the first to consider that an answer may span over a federation of datasets of different data models, with very different or even absent internal structure (the latter is true for text data). For instance, a national company registry is typically relational, contracts or political speeches are text, social media content typically comes as JSON documents, and open data is often encoded in RDF graphs.

Answering such queries raises several technical challenges. First, there are **many**, **structurally heterogeneous**, independently-produced data sources. Second, answers may require **interconnecting data from several sources**, e.g., the history of a company, the Wikipedia page of a person, and the public information available on the company and its CEO. Finally, a staple of professional journalism is to be able to show evidence for published news. Thus, it is important to be able to **show where each piece of information in an answer came from and how the connections were created**. To address these challenges, this work makes the following contributions:

- a *graph representation* of a set of interconnected heterogeneous datasets;
- a *score function*, based on the novel notion of *edge specificity*, used to rank answer trees to return them in order of decreasing interesting;
- the first cross-data source *query answering algorithm* which, given a keyword query, returns relevant answer trees spanning over any subset of datasets.

We have implemented these contributions into a system, and present experimental results validating its practical usefulness. A previous version of our system had been demonstrated in [3]. Since then, we have completely re-engineered the source registration and devised a better query algorithm.

This document is organized as follows. Section 2 provides the formal problem statement. The technical contributions of our work are described in Section 3 where we introduce the algorithms, data structures and information extraction methods. Section 5 details our query answering algorithm, while Section 6 presents experimental results. Section 7 positions our work with respect to related works, then we conclude.

2 Formal problem statement

Our problem is formalized as follows. We consider the following *data models*: Relational (including SQL, web tables, CSV, etc.), RDF, JSON, HTML and text. A dataset DS in our context is either a relational database, or an RDF graph, a JSON file, an HTML file, or a text file. Let A be an alphabet of words.

We define an **integrated graph** $G = (N, E)$ where N is a set of nodes and E the set of edges. We have $E \subseteq N \times N \times A^* \times [0, 1]$, where A^* denotes the set of (possibly empty) sequences of words, and the value in $[0, 1]$ is the *confidence*, reflecting the probability that the relationship between two nodes actually holds. Each node $n \in N$ has a label $\lambda(n) \in A^*$ and similarly each edge e has $\lambda(e) \in A^*$. We use ϵ to denote the empty node label.

Section 2.1 explains how each type of dataset yields nodes and edges in the integrated graph. Then, Section 2.2 introduces a special kind of graph edges,

[5] https://team.inria.fr/cedar/contentcheck/

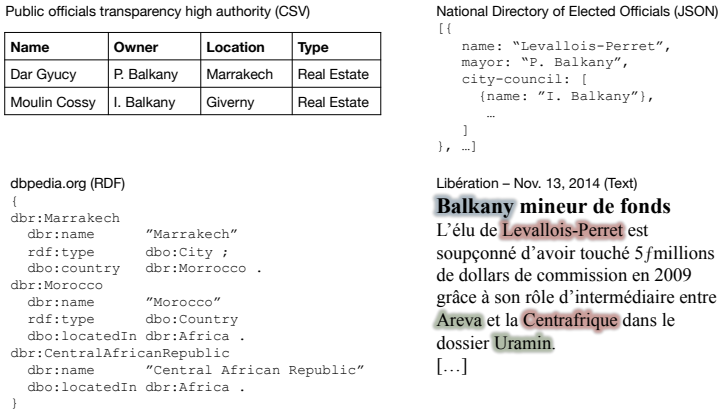

Fig. 1. Data set collection $\mathcal{D}$. Starting from the top left, clockwise: a table with assets of public officials, a JSON listing of France elected officials, an article from the newspaper Libération with entities highlighted, and a subset of the DBPedia RDF knowledge base.

which connect nodes from the same or from different datasets, based on their similarity. Section 2.3 formalizes our search problem.

2.1 Mapping each dataset to the graph

First, we introduce a **dataset node** n_{DS_i} for each dataset DS_i. Second, for any dataset D, and any constant c (e.g., literal, number, etc.) appearing once or several times in the dataset, a single node n_c is created, such that $\lambda(n_c) = c$. This node will have incoming edges —from "parent" node(s)— reflecting each of its occurrences; how these edges are created depends on the data model of D. We present the remaining nodes and edges created in the graph out of each distinct data model below. We also describe how we *rely on entity extraction to identify possible occurrences of structured data items within text*; this enables in particular to exploit text datasets alongside structured ones. For illustration, Figure 1 shows a set of datasets, while Figure 2 shows the nodes and edges of the integrated graph resulting from them.

Relational. Let $R(a_1, \ldots, a_m)$ be a relation (table), which we view as a relational dataset DS. A node n_{DS} is created to represent R (yellow node with label hatvp.csv in Figure 2). Let $t = (v_1, \ldots, v_m)$ be a tuple in R. A node n_t is created for the tuple t with a label giving it a unique **id**, and an edge from n_{DS} to n_t with confidence 1 and label $origDS$. Such edges are *virtual* (we do not materialize them), and instead simply store for each node id, the id of the dataset it comes from. *Here and below, some edges between a dataset and its nodes are ommitted to avoid clutter.* For each non-null attribute in t, an edge goes from n_t to each node corresponding to the node representing the value v_i in R, with confidence 1 and label a_i[6] (for example, the edge labeled owner). To keep the graph readable,

[6] We do not create graph edges from null-valued attributes because such attributes do not correspond to actual data items and thus cannot lead to connections.

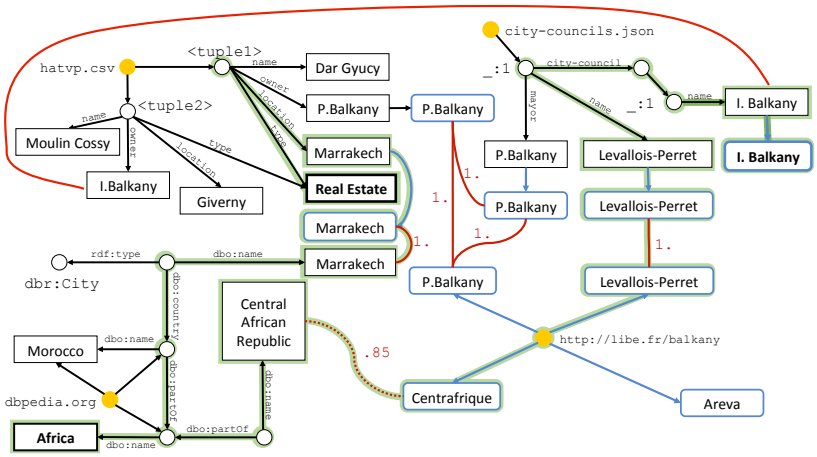

Fig. 2. Integrated graph corresponding to the datasets of Figure 1. An answer to the keyword query {"I. Balkany", Africa, Estate} is highlighted in light green; the three keyword matches in this answer are shown in bold.

confidence values of 1 are not shown. Moreover, for any two relations R, R' for which we know that attribute a in R is a foreign key referencing b in R', and for any tuples $t \in R$, $t' \in R'$ such that $t.a = t'.b$, the graph comprises an edge from n_t to $n_{t'}$ with confidence 1.

RDF. The mapping from an RDF graph to our graph is the most natural. Each node in the RDF graph becomes a node in G and each RDF triple edge becomes an edge in E with confidence 1 and the RDF edge label. As above, a virtual edge is created from n_{DS} (yellow node labeled dbr:City) to each node.

JSON. A set of nodes is created out of the JSON document D recursively starting from the root (yellow node with label city-councils.json in Figure 2), as follows. (i) From a JSON constant value c, a single node n_c is created for all occurrences in D sharing the same *value and path from the root*. (ii) From a simple map of the form $\{name : value\}$, two nodes are created, n_1 and n_2, together with an edge labeled *name* from n_1 to n_2 with confidence 1. The label λ of the node n_1 is an empty string (also denoted ϵ). (iii) From a more general map of the form $\{n_1 : v_1, n_2 : v_2 \ldots n_m : v_m\}$, we create $m+1$ nodes: n_1 labeled ϵ represents the map, and it has m outgoing edges, labeled $n_1, \ldots, n_m$ leading to the nodes that are created respectively from the values $v_1, \ldots, v_m$. In Figure 2, we have three outgoing edges named mayor, name and city−council. (iv) From an array of the form $[v_1, v_2, \ldots, v_m]$, we also create $m+1$ nodes, one for the array, labeled ϵ, and one for each non-null array element. The edge leading from the array node to each child node has an empty label.

HTML. The conversion of an HTML document is handled in a very similar way to JSON, reflecting the particular semantics of HTML links.

Plain text, and entity extraction. Let D be a text dataset. We rely on *entity extraction* models which, given a text from A^*, extract a set $\mathcal{E}$ of entity occurrences (or entities, in short) present in the text. In G, an occurrence n_E of an entity $E \in \mathcal{E}$ is represented by a node n_E (yellow node with label http://libe.fr/balkany in the figure), together with an edge leading from n_D (the node representing the text dataset) to n_E; the edge has an empty label and also carries the confidence (between 0 and 1) of the extraction. In Figure 2, the blue, round-corner rectangles Centrafrique, Areva, P. Balkany, Levallois-Perret correspond to the entities extracted from the original text document.

We apply entity extraction as stated above, not only on text documents but also *on any text (whether occurring as a tuple attribute, an RDF literal, or a text JSON node)* whose length is above a fixed threshold t_{ext}. Figure 2 shows extracted entities as blue, round-corner boxes, children of the text nodes they are extracted from (for example, text nodes from the JSON data source). Thus, a G node whose text label contains an identified entity occurrence is, in G, the parent of the node corresponding to the extracted entity occurrence.

2.2 Same-as edges

At the heart of our query approach are nodes that appear (with identical or strongly similar labels) several times in G. To assess if two nodes $n_1 \in D_1, n_2 \in D_2$ (where $D_1 = D_2$ or $D_1 \neq D_2$) have similar labels, we rely on:

- a set of *selectors* which, for each kind of node (e.g., extracted Person entity, extracted Organization entity, untyped literal, number, etc.) determine the nodes from G with which it makes sense to compare a node of the given kind. For instance, Person entities are only compared with Person etc.
- a set of *similarity functions* which we choose based on the entity matching literature, and which are applied in each comparison setting, e.g., one is used to compare Person entities, another to compare numbers etc.

Selectors and similarity functions are detaile in our technical report.

When comparing the labels of two nodes n_1, n_2, using a similarity function, if the returned value is 1.0, we say that the two nodes are *equivalent* (represented by the red edges in Figure 2); if the similarity is a value $T < s < 1.0$ for a fixed threshold T, then the two nodes are *similar* (represented by the red, dashed edge with confidence 0.85 in the figure). In both cases, we add an edge labeled *sameAs*, connecting the two nodes. The edge confidence is 1.0 if the nodes are equivalent, and a lower c corresponding to the similarity between the labels, if the nodes are similar. If $s < T$, no connection is recorded between n_1 and n_2; their similarity is considered too low. In our example, sameAs edges are red.

2.3 Search problem

Let W be the set of *keywords*, obtained by stemming the word set A; a *search query* is a set of keywords $Q = \{w_1, ..., w_m\}$, where $w_i \in W$. We define an *answer tree* (AT, in short) as a tree of G edges (*considered undirected*, that is: $n_1 \xrightarrow{a} n_2 \xleftarrow{b} n_3 \xrightarrow{c} n_4$ is a sample AT), such that for all $w_i \in Q$, there is either:

- a node n_i appearing in some edge of the AT such that $w_i \in \lambda(n_i)$; or

– an edge $n_i \xrightarrow{a} n_j$ in the AT such that $w_i \in \lambda(a)$.

We treat G edges as undirected when defining the AT in order to enable more query results, on a graph built out of heterogeneous content whose structure we do not control. For instance, consider a query consisting of the keywords k_1, k_4 such that $k_1 \in \lambda(n_1)$ and $k_4 \in \lambda(n_4)$ on the four-nodes sample AT introduced above. If our ATs were restricted to the original direction of G edges, the query would have no answer; ignoring the edge directions, it has one.

Further, we are interested in *minimal* answer trees, that is: removing an edge from the tree should make it lack one or more of the query keywords w_i. In Figure 2, a green-highlight answer tree connects the nodes labeled Africa, Real Estate and Levalois-Perret (in bold), respectively.

Several minimal answer trees may exist in G for a given query. We describe a *scoring function* which assigns a higher value to more interesting answer trees in Section 4 . Thus, our problem can be stated as follows:

PROBLEM STATEMENT Given the datasets $DS_1, \ldots, DS_n$ and a query Q, return the k highest-score minimal answer trees.

An AT may potentially span over the whole graph, (also) because it can traverse G edges in any direction; this makes the problem challenging.

3 Building and storing the integrated graph

We now explain how we compute and store the elements of our graph, in particular sameAs edges which enable interconnecting datasets (Section 3.2), and entities we extract to enrich the graph (Section 3.3).

3.1 Basic data structures

Nodes and edges To each node $n \in G$ we associate (i) a type (whether it stands for a dataset, an entity, a literal, a number, an URI etc.), and (ii) a globally unique ID. These are stored in a relation $N(ID, type, \lambda)$ which also stores for each node n its label $\lambda(n)$. Similarly, we store edges in a collection $E(ID, n_1.ID, n_2.ID, \lambda, c)$ where $n_1.ID, n_2.ID$ are the identifiers of the nodes defining the edge, λ is its label and c is its confidence between 0 and 1.

Index We build an *inverted index* accounting for the presence of (stemmed) keywords in our dataset. Specifically, the index $I(ID, kwd)$ associates a keyword $w \in W$ with the IDs of the G nodes (or edges) where it appears[7]. Following a common optimization in information retrieval systems, we only index noun words from text, based on the observation that non-noun words (e.g., verbs, pronouns etc.) are pretty common and may appear in many (text) datasets without this corresponding to a data connection. We also decompose URIs based on the separators present therein, and index the domain names as well as any nouns recognized in the URIs, as keywords associated to an URI node.

For instance, in our example dataset (Figure 2), the index contains entries for "L. Balkany", "Morocco", "Africa", "Areva" etc.

[7] Each index entry also contains an attribute which states whether ID refers to a node or edge. For simplicity, we omit this attribute from our discussion.

3.2 Treatment of similar nodes

A naïve way to represent equivalent and similar nodes in the graph would consist of adding, between each such two nodes, an edge carrying a *sameAs* label, with the confidence value c. This naïve approach has the drawback of storing $O(p^2)$ edges for equivalent p nodes; if the label is very frequent, e.g., "France" in a French personnel dataset, this number can quickly explode.

Instead, we take a more elaborate approach, as follows:

(i) If $n_1, \ldots n_p$ are equivalent nodes (i.e., with equal labels) found *in the same* dataset D, we only create *one* node with that label, e.g., n_1, assuming that historically, this has been the first node with that label encountered when registering D. Then, in any edge resulting from D, any node n_i with $1 < i \leq p$ is replaced with n_1. This reflects the idea that the same constant in the same dataset typically has the same significance, e.g., in an employee file, a zipcode such as "91200", or an email such as "a@b.org" appearing multiple times usually mean the same thing. However, this may also introduce some confusions, e.g., if name="Marie" and street="Marie" appear in different places in a JSON dataset, this is more a coincidence than a reliable connection. To avoid such unwanted node unifications, for datasets that are *hierarchical (tree-structured)* before this transformation, e.g., JSON or relational, we use the path from the dataset root to each node as a supplementary criterium for deciding when to unify identical-label nodes. Thus, two same-label nodes on the path .employees.employee.address.zipcode are unified into one, whereas nodes on the different paths .employees.employee.name and .employees.employee.address.street (the two "Marie" above) are not.

(ii) If $n_1, \ldots, n_p$ are equivalent nodes found *in different datasets* $D_1, \ldots, D_k$, we arbitrarily choose one (say, n_1, assuming D_1 was registered before all the others) and consider it the *representative* of $n_1, n_2, \ldots, n_p$. Thus, we add a column to the N relation storing nodes defined in Section 3.1, turning it into $N(ID, type, \lambda, rep)$. This incurs a constant storage overhead for every node, instead of $O(p^2)$ *sameAs* edges for each set of p identical-labeled nodes. If $p = 1$, i.e., if a node's label is not shared by any other node, the node represents itself.

For node pairs that are similar, we store explicitly the *sameAs* edges with a confidence value representing the similarity. This is necessary because one cannot infer such an edge if it is not stored. For instance, if $sim(n_1, n_2) = 0.85$ and $sim(n_2, n_3) = 0.81$, it is generally not possible to infer the value of $sim(n_1, n_3)$; we need to explicitly store the respective *sameAs* edge with the similarity value serving as the confidence c. For instance, in Figure 2, *sameAs* edges with a confidence of 1.0 are shown between the two nodes labeledMarrakech, the three P. Balkany nodes etc.; another with confidence 0.85 connects Central African Republic to Centrafrique (the labels are not identical).

3.3 Information extraction

We apply Named Entity Extraction to identify, within text, people, places, or organizations. In our sample graph, this leads to all the *blue, round-noded boxes*: each of them is an entity, added in the graph as a child of the text node inside which it was recognized. When an entity, e.g., P. Balkany, is recognized in a text

node, the keywords contained in the entity name are indexed as labeling the entity (the text node "transfers" the keywords to its entity child). This allows us to return answer trees as meaningful (containing as many entities) as possible to user queries. We are currently developing a tool to extract relationships between entities in a given sentence to enrich the graph with more useful connections.

4 Scoring Answer Trees

Section 4.1 introduces a metric on edges, which will be used to favor, in AT, edges that are "rare" for both nodes they connect; we also explain how to compute and maintain this metric as datasets are added to the graph. Section 4.2 introduce our AT scoring function, motivated by our data journalism scenarios.

4.1 Edge specificity

For a given node n and label l, let $N^l_{\rightarrow n}$ be the number of l-labeled edges entering n, and $N^l_{n\rightarrow}$ the number of l-labeled edges outgoing n. The **specificity** of an edge $e = n_1 \xrightarrow{l} n_2$ is defined as: $s(e) = 2/(N^l_{n_1\rightarrow} + N^l_{\rightarrow n_2})$. $s(e)$ is 1.0 for edges that are "unique" for both their source and their target, and decreases when the edge does not "stand out" among the edges of these two nodes. For instance, the city council of Levalois-Perret comprises only one mayor (and one individual cannot be mayor of two cities in France, because he has to inhabit the city where he runs for office). Thus, the edge from the city council to P. Balkany has a specificity of $2/(1.0 + 1.0) = 1.0$. In contrast, there are 54 countries in Africa (we show only two), and each country is in exactly one continent; thus, the specificity of the dbo:partOf edges in the DBPedia fragment, going from the node named Morocco (or the one named Central African Republic) to the node named Africa is $2/(1 + 54) \simeq .036$.

How to compute edge specificity? When registering the first dataset D_1, computing the specificity of its edges is easy. However, when registering subsequent datasets D_2, D_3 etc., if some node, say $n_2 \in D_2$ is found to be equivalent to a node $n_1 \in D_1$, both the D_1 edges adjacent to n_1 and the D_2 edges adjacent to n_2 are attached to the same conceptual node, representing n_1, n_2 and all (possible, future) nodes that are equivalent to them. Therefore, the D_1 edges adjacent to n_1 and the D_2 edges adjacent to n_2 should be reflected in the specificity of each of these edges, thus, the specificity of D_1 edges needs to be *recomputed* when a node in a source added after D_1 is equivalent to one of its nodes.

Below, we describe an efficient incremental algorithm to (re)compute specificity. We introduce two notations. For any edge e, we denote $N^e_{\rightarrow\bullet}$, respectively, $N^e_{\circ\rightarrow}$ the two *numbers out of which the specificity of e has been most recently computed*[8]. Specifically, $N^e_{\rightarrow\bullet}$ counts l-labeled edges incoming to the target of e, while $N^e_{\circ\rightarrow}$ counts l-labeled edges outgoing the source of e. In Figure 3, if e is the edge $x \xrightarrow{l} n_1$, then $N^e_{\rightarrow\bullet} = 3$ (blue edges) and $N^e_{\circ\rightarrow} = 1$, thus $s(e) = 2/4 = .5$.

[8] This can be either during the first specificity computation of e, or during a recomputation, as discussed below.

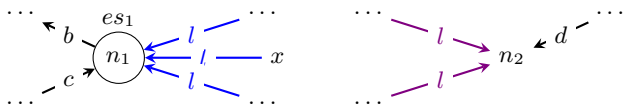

Fig. 3. Illustration for specificity (re)computation. The specificity of the edge $x \xrightarrow{l} n_1$, $s(e)$ is initially computed out of the blue edges; when n_2 joins the equivalence set es_1, it is recomputed to also reflect the violet edges.

Let $n_1 \in D_1$ be a node, es_1 be the set of all nodes equivalent to n_1, and $n_2 \in D_2$ be a node in a dataset we currently register, and which has just been found to be equivalent to n_1, also.

Further, let l be a label of an edge incoming or outgoing (any) node from es_1, and/or n_2. We denote by $N^l_{\rightarrow es_1}$ the sum $\sum_{n \in es_1}(N^l_{\rightarrow n})$ and similarly by $N^l_{es_1 \rightarrow}$ the sum $\sum_{n \in es_1}(N^l_{n \rightarrow})$; they are the numbers of l-labeled outgoing (resp., incoming) l-labeled edges of any node in es_1. When n_2 joins the equivalence set es_1 of n_1 (see Figure 3):

1. If $N^l_{\rightarrow es_1} \neq 0$ and $N^l_{\rightarrow n_2} \neq 0$, the specificity of every l-labeled edge e *incoming* either a node in es_1 or the node n_2 must be recomputed.

Let e be such an *incoming* edge labeled l. When n_2 is added to the set es_1, the specificity of e becomes $2/((N^e_{\rightarrow \bullet} + N^l_{\rightarrow n_2}) + N^e_{\circ \rightarrow})$, to reflect that n_2 brings more incoming l-labeled edges. This amounts to $2/(3+2+1) = .33$ in Figure 3: the violet edges have joined the blue ones.

Following this adjustment, the numbers out of which e's specificity has been most recently computed are modified as follows: $N^e_{\rightarrow \bullet}$ becomes $N^e_{\rightarrow \bullet} + N^l_{\rightarrow n_2}$, thus $3 + 2 = 5$ in Figure 3; $N^e_{\circ \rightarrow}$ remains unchanged.

2. If $N^l_{\rightarrow es_1} = 0$ and $N^l_{\rightarrow n_2} \neq 0$, the specificity of every l-labeled edge e *incoming* n_2 does not change when n_2 joins the equivalence set es_1.

3. If $N^l_{\rightarrow es_1} \neq 0$ and $N^l_{\rightarrow n_2} = 0$, the newly added node n_2 does not change the edges adjacent to the nodes of es_1, nor their specificity values.

The last two cases, when $N^l_{es_1 \rightarrow} \neq 0$ and $N^l_{n_2 \rightarrow} \neq 0$, respectively, $N^l_{es_1 \rightarrow} = 0$ and $N^l_{n_2 \rightarrow} \neq 0$, are handled in a similar manner.

This above procedure is quite efficient because it is *local*, i.e., it only needs, for a given label l, the number of edges adjacent to the new node n_2 which joins the equivalence set es_1, and the number of edges adjacent to a node from es_1.

Concretely, to (re)compute specificity, we add to the relation E introduced in Section 3.1 three attributes: $N^e_{\rightarrow \bullet}$, $N^e_{\circ \rightarrow}$ and s, the last-computed specificity.

4.2 Answer trees scoring

We use a score function to compare the quality of answers to a same query Q.

First, the score needs to reflect the quality of the answer tree, that is, the extent to which it matches the query. A second component of the score is inspired by the particular applications we consider: intuitively, an answer tree that goes through *rare, important* graph edges is preferable. Formally, let $t = (N, E)$ be an answer tree to a query $Q = \{w_1, \ldots, w_n\}$. The **connection score** $conns(t)$ is a measure of the interest of t *independently of the query* Q. It is defined as:

$$conns(t) = \beta \cdot \prod_{e \in E} c(e) + (1 - \beta) \cdot \frac{1}{|t|} \cdot \sum_{e \in E} s(e)$$

where $0 \leq \beta \leq 1$, $c(e)$ is the confidence attached to the edge e (Section 2), $|t|$ is the number of edges in t, and $s(e)$ is the specificity of e (Section 4.1).

The **matching score** $ms(t)$ of an AT t is a $|Q|$-dimensional vector, whose elements are computed as follows. For each keyword $w_i \in Q$, the i-th element of the vector is the mean of the matching scores for each t node or edge on which w_i matches. This node (or edge) matching score for a query keyword w_i is computed by applying the similarity function used for sameAs computation (Section 2.2) to w_i and $\lambda(n)$ (respectively, $\lambda(e)$ on an edge).

Comparing two answer trees Let t_1 and t_2 be two answer trees to a same query Q. First, if $ms(t_1)$ contains less zeros than $ms(t_2)$, t_1 is preferred; we prefer ATs matching as many keywords as possible. Then, if t_1 and t_2 match the same number of keywords, we compute

$$score(t, Q) = \alpha \cdot ms(t, Q) + (1 - \alpha) \cdot conns(t)$$

for $t \in \{t_1, t_2\}$, and prefer the one with the highest score.

5 Answering keyword queries

We now present our approach for computing query answers, based on the integrated graph. Section 5.1 discusses the consequences of these choices on our search problem, which is significantly harder than those previously studied. Section 5.2 describes our search algorithms.

5.1 Search space and complexity

Given a graph G with weights (costs) on edges, and a set of G nodes $n_1, \ldots, n_m$, the *Steiner Tree Problem (STP)* [7] consists of finding the smallest-cost tree in G that connects all the nodes together. We could answer our queries by solving one STP problem for each combination of nodes matching the keywords $w_1, \ldots, w_m$. However, there are several obstacles left: ($\diamond$) STP is a known NP-hard problem in the size of G, denoted $|G|$; ($\triangleright$) as we consider that each edge can be taken in the direct or reverse direction, this amounts to "doubling" every edge in G. Thus, our search space is $2^{|G|}$ larger than the one of the STP, which is daunting even for "small" graphs of a few hundred edges; ($\triangleleft$) we need the k smallest-cost trees, not just one; ($\circ$) each keyword may match several nodes, not just one.

The closely related *Group STP* (GSTP, in short) [7] is: given *m sets of nodes* from G, find the minimum-cost subtree connecting one node from each of these subtrees. GSTP does not raise the problem ($\circ$), but still has all the others.

STP and GSTP assume the tree cost is *monotonous*, that is: for any query Q and all trees $T, \hat{T}$ where T is a subtree of $\hat{T}$ it follows score(T) $\leq$ score($\hat{T}$). This is naturally satisfied if the cost is the addition of edge weights. A last obstacle blocking the reduction of our problem to (G)STP is: ($\square$) our score function is not monotonous, as illustrated in Figure 4, where on each edge, c is the confidence and s is the specificity. Denoting T the four-edge tree rooted in n_1, the connection score of $T' = T \cup \{n_4 \rightarrow n_6\}$ is smaller than that of T alone (see the connection score function presented in Section 4.2). If we assume that T' has the same matching score as T, the global score of T' is smaller than that of T, contradicting the monotonicity assumption.

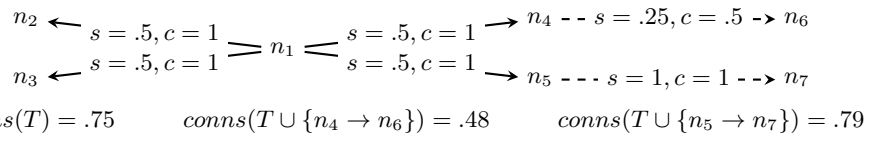

Fig. 4. Example: $conns(\cdot)$ is non-monotonous. T is the four-edges tree rooted in n_1.

In the literature, (G)STP has been addressed making various simplifying assumptions that do not hold in our context. For instance: the quality of a solution exponentially decreases with the tree size, thus search can stop when all trees are under a certain threshold [2]; edges are considering in a single direction [19,5,14]; the cost function is monotonous [4] etc. While these assumptions reduce the computational cost, they are all contradicted by concrete examples we encountered while analyzing our journalist partner's datasets. Algorithms which find *bounded approximations*, i.e., (G)STP trees solutions whose cost is at most p times higher than the optimal cost, e.g., [8,9] are not applicable, either: in a data journalism context, journalists want to be the first to uncover an interesting data connection, not one "at most p times less interesting".

The above search space analysis shows that any connected subtree of G could be part of the answer. Since they cannot be all enumerated, neither can we adopt prior simplifications and restrictions, our approach is to *enumerate G subtrees, starting from the smallest ones*, in a given *time budget*, and striving to find also *multi-dataset answers*; when the time budget is exhausted, we return the best k answers found. Enumerating small trees first is both a practical decision (we use them to build larger ones) and fits the intuition that we shouldn't miss small answers that a human could have found manually. Many-dataset answers interest us since they avoid most manual labor to the journalists, by interconnecting multiple data sources, of potentially different formats.

We detail the first known algorithm for keyword search over multiple datasets.

5.2 Grow and Aggressive Merge (GAM) Algorithm

Our first algorithm uses some concepts from the prior literature [4,11] while exploring much more trees. Specifically, it starts from the sets of nodes $N_1, \ldots, N_m$ where the nodes in N_i all match the query keyword w_i; each node $n_{i,j} \in N_i$ forms a 1-node partial tree. For instance, in Figure 2, 1-node trees are built from the nodes with boldface text, labeled "Africa", "Real Estate" and "I. Balkany". Then, two transformations can be applied. GROW(t, e), where t is a tree, e is an edge *adjacent to the root* of t, and e does not close a loop with a node in t, creates a new tree t' having all the edges of t plus e; the root of the new tree is the other end of the edge e. For instance, starting from the node labeled "Africa", a GROW can add the edge labeled `dbo:name`. MERGE(t_1, t_2), where t_1, t_2 are trees with the same root, whose other nodes are disjoint, and having matching disjoint sets of keywords, creates a tree t'' with the same root and with all edges from t_1 and t_2. Intuitively, GROW moves away from the keywords, to explore the graph; MERGE fuses two trees into one that matches more keywords than both t_1 and t_2. It has been shown [11] that using MERGE steps helps reduce the search effort, as it allows partial trees to "meet halfway" on a common root node.

The changes we bring to adapt the algorithm to our harder problem (bidirectional edges and multiple interconnected datasets) are as follows.

(*i*) We allow GROW to traverse an edge both going from the source to the target, and going from the target to the source. For instance, the type edge from "Real Estate" to <tuple1> is traversed target-to-source, whereas the location edge from <tuple1> to "Real Estate" is traversed source-to-target.

Our next modification is to enable *traversing a dataset that does not match any keyword*, such as `http://libe.fr/balkany` in our example. Observe that search starts from the nodes matching query keywords, and GROW and MERGE do not go across datasets. This would miss answers (such as the one in Figure 2) which require such a traversal. A naïve solution is to allow GROW to also add a sameAs edge to the root of a tree. However, this can be very inefficient. Consider three nodes m, m' and m'', all equivalent and connected through strong sameAs edges (e.g., the three "P. Balkany" nodes): a GROW step could add one sameAs edge, the next GROW could add another on top of it etc. Very frequent entities (e.g., "France" in our French journalistic datasets) lead to a very high number of sameAs edges, which successive GROW would chain in all possible paths. This is wasteful, because any two nodes connected by a path of strong sameAs edges, are also connected by one such edge. The key observation here is that we *want* such edges to be used by our algorithm, while we want to avoid *exploring* them like any other edge. Thus:

(*ii*) We extend GROW to add to a tree t rooted in n_1 from dataset D_1, a strong sameAs edge going to a node n_2 from a distinct dataset D_2, *and in the same step*, an edge that is not a strong sameAs, going from n_2 to some other node n_3 in D_2; the resulting tree is rooted in n_3. We call this extension GROWACROSS. It prevents adding a second strong sameAs edge by "moving away" the tree root from n_2 to another node n_3, not equivalent to it. In our example, GROWACROSS goes from the central Marrakech entity node (round-corners blue box) extracted from the string node just above it, from the hatvp.csv dataset, to the Marrakech node from the RDF graph dbpedia.org and from there, to the node at its left.

Order of exploration How to interleave GROW, GROWACROSS and MERGE? We decide to apply in sequence: one GROW or GROWACROSS (see below), leading to a new tree t, immediately followed by all the MERGE operations possible on t. Thus, we call our algorithm **G**row and **A**ggressive **M**erge (GAM, in short). We merge aggressively in order to detect as quickly as possible when some of our trees, merged at the root, are a solution.

We also need to decide what GROW (or GROWACROSS) to apply at a certain point. For that, we use a *priority queue* Q in which we add (tree, edge) entries: for GROW, with the notation above, we add the (t, e) pair, while for GROWACROSS, we add the tree t' rooted at n_2 (thus, already containing the strong sameAs edge between n_1 and n_2), together with the edge $n_2 \to n_3$, i.e., the first half of GROWACROSS is already performed. In both cases, when a (t, e) pair is extracted from Q, we just extend t with the edge e (adjacent to its root), leading to a new tree t_G, whose root is the other end of the edge e. Then we aggressively merge t_G with all compatible trees explored so far, finally we fetch from G the (data

or sameAs) edges adjacent to t_G's root and add to Q more (tree, edge) pairs to be considered further during the search. The algorithm then picks the highest-priority pair in Q and reiterates; it stops when Q is empty (or, e.g., at a timeout).

The last parameter impacting the exploration order is the priority used in Q: at any point, Q gives the highest-priority (t, e) pair, which determines the operations performed next. Trees matching *many query keywords* are preferable, to go toward complete query solutions; at the same number of matched keywords, *smaller trees* are preferable in order not to miss small solutions (similar to a breadth-first search); finally, among (t_1, e_1), (t_2, e_2) with the same number of nodes and matched keywords, we prefer the pair with the *higher specificity edge*.

6 Experimental evaluation

We have implemented our approach in **Java 1.8**; the code consists of 155 classes and amounts to 31200 lines. We relied on **Postgres 9.6.5** to store our integrated graph, **TreeTagger** served as a POS (part-of-speech) tagger, and **StanfordNLP**, we trained a model for the French language based on a corpus of press articles[9], for named entity extraction (Section 3.3). Experiments ran on a MacBook Pro 10.12.6, with a 2.8 GHz Intel i7 and 16 GB RAM.

Parameter setting We have used the following values for our thresholds and coefficients: t_{ext} (Section 2.1) is set to 10 characters; T (Section 2.2) is set to 0.5; α, which balances confidence vs. specificity in an AT score (Section 4), is set to 0.5 (e give a higher weight to the confidence because they are multiplied, and thus that part of the score tends to 0 very fast), while β is set to 0.8.

Corpora First, we built a corpus of **113 HTML articles** (1.4MB) crawled from the French online newspaper Mediapart with the search keywords "gilets jaunes" (Yellow Vests, a protest movement in France over the last year). Together, these articles lead to a graph G_1 of $E_1 = 41195$ edges and $N_1 = 27112$ nodes; among these, $N_1^P = 1357$ people, $N_1^L = 1346$ locations and $N_1^O = 501$ organizations, many of which appearing several times, e.g., France (136 times), Emmanuel Macron (47 times), etc. On this corpus, our queries (based on frequent terms and names involved in the events) look for connections that can be made, e.g., when a journalist writes separately about two different topics, or when a title connects a person with a place, or when two people p_1 and p_2 co-occur, in separate documents, with a location l etc.

Second, we used a manually built **French politics knowledge base (RDF graph)** of **38108** triples, comprising parties, politicians, and their Twitter accounts; we got the latter from Le Monde, turned them into RDF and enriched them with DBPedia education and spouse information about the politicians, when available. This lead to a graph G_2 of $E_2 = 45378$ edges, $N_2 = 19192$ nodes, including $N_2^P = 1784$ people, $N_2^L = 238$ locations and $N_2^O = 18$ organizations.

Queries and results Figure 5 shows for a set of queries of 1 to 4 keywords: the number of answers N_{AT}, the number of trees N_T, among which we distinguish the number of trees created by GrowAcross (N_{GA}), respectively, by Grow (N_G) and Merge (N_M), the time to the first solution, and the total time (in

[9] http://catalog.elra.info/en-us/repository/browse/ELRA-W0073/

Query	N_{AT}	N_T	N_{GA}	N_G	N_M	Time to 1st	Total time
Macron	50	150	0	0	0	1	47
Christophe Dettinger	29	29921	28196	988	37	333	2533
Etienne Chouard Rodrigues	1	61647	57721	3671	255	1207	10000
Thierry-Paul Valette Drouet	4	89682	5698	3962	22	3595	10000
Mélenchon Aubry	50	83938	82513	1368	57	37	4333
Castaner flashball	46	157039	151585	5408	46	412	10000
Drouet Levavasseur	50	32493	30919	1523	51	359	2006
Dupont-Aignan Chalençon	50	17754	16122	1582	50	80	1174
Maquet Demarthe	50	821	0	771	50	371	2463
Hollande76 TousHollande	7	3612	4	3601	7	674	10000
Fougerat Proust	5	694	0	644	50	9	1729
Barnier Faure	50	960	2	908	50	19	1541
Barnier Balas	50	754	2	702	50	438	2211
Gilles Lebreton Nicolas Bay	50	3748	14	2774	960	7673	8849

Fig. 5. Experiment results on Mediapart (top) and French politics (bottom) corpora. milliseconds). The executions were limited to **10 seconds** or **50 answers found**, whichever came first; thus, some queries did not run to completion, which is expected given the huge search space size (Section 5.1). Naturally, 1-keyword queries did not need Grow, GrowAcross nor Merge. Many answers are found within 10 seconds; the first are found faster, depending (also) on their size. A majority of trees is generated by GrowAcross, showing the importance and impact of interconnecting datasets. As expected, the running time is strongly correlated with the number of explored trees. Very little GrowAcross happens on G_2, since it originates from just 1 RDF dataset; its few (463) strong sameAs edges are between entities found by the extraction. In contrast, G_1 has 9345 such edges, significantly adding to the search effort. The interconnection between datasets opens opportunities, while increasing the computational effort. Note that each query ran in isolation; caching partial trees and sharing them across executions to reduce the running time is one direction for future work.

7 Related work and Conclusions

Keyword search (KS) in relational databases is studied in [12,15,16,17,18,19]. In these works and ours, primary key-foreign key (PK-FK) connections add edges to the graph; [16] also establishes links based on similarity (or equality) of constants appearing in different relational attributes. Our problem is harder since our trees can traverse edges in both directions, and paths can be (much) longer than those based on PK-FK alone. [18] proposes to incorporate user feedback through active learning to improve the quality of answers in a relational data integration setting. KS works over RDF graphs [5,14] traverse edges in their direction only. Thus, the diameter of the graph that needs to explored is limited. Our bidirectional search leads to a much more complex task.

Broadly, working with heterogeneous datasets is a goal of data integration *Mixed-instance querying* either through structured languages like XRQ, or through keywords like in S4 [2], are technical tools to exploit multiple data sources. In [13,1], data sources and files are only referenced and query/search

interfaces are accessed at query times (a.k.a *pay-as-you-go data integration*). GOODS [10] is a concrete implementation of a *dataspace* which was first introduced in [6] to designate "a large number of diverse, interrelated data sources".

In this paper, we have considered the problem of answering keyword queries on a set of heterogeneous data sources. We model all sources as a graph, introduced edge specificity and a score function based on it, and described the first known algorithm for finding answers under a time budget in such a context. Our experiments demonstrate the feasibility of our approach.

Acknowledgments L. Elbert studied score function properties. C. Conceição trained the NER extractor for French texts.

References

1. Alagiannis, I., Idreos, S., Ailamaki, A.: H2o: A hands-free adaptive store. In: SIGMOD. No. EPFL-CONF-198683 (2014)
2. Bonaque, R., Cautis, B., Goasdoué, F., Manolescu, I.: Social, structured and semantic search. In: EDBT (2016)
3. Chanial, C., Dziri, R., Galhardas, H., Leblay, J., Nguyen, M.H.L., Manolescu, I.: CONNECTIONLENS: Finding connections across heterogeneous data sources (demonstration). VLDB (2018)
4. Ding, B., Yu, J.X., Wang, S., Qin, L., Zhang, X., Lin, X.: Finding top-k min-cost connected trees in databases. In: ICDE (2007)
5. Elbassuoni, S., Blanco, R.: Keyword search over rdf graphs. In: CIKM. ACM, New York, NY, USA (2011)
6. Franklin, M., Halevy, A., Maier, D.: From databases to dataspaces: A new abstraction for information management. SIGMOD Record **34**(4) (Dec 2005)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co. New York (1990)
8. Garg, N., Konjevod, G., Ravi, R.: A polylogarithmic approximation algorithm for the group Steiner tree problem. In: SIAM (1998)
9. Gubichev, A., Neumann, T.: Fast approximation of Steiner trees in large graphs. In: CIKM (2012)
10. Halevy, A., Korn, F., Noy, N.F., Olston, C., Polyzotis, N., Roy, S., Whang, S.E.: Goods: Organizing google's datasets. In: SIGMOD. ACM (2016)
11. He, H., Wang, H., Yang, J., Yu, P.S.: BLINKS: ranked keyword searches on graphs. In: SIGMOD (2007)
12. Hristidis, V., Papakonstantinou, Y.: DISCOVER: keyword search in relational databases. In: VLDB (2002)
13. Idreos, S., Kersten, M.L., Manegold, S.: Database cracking. In: CIDR (2007)
14. Le, W., Li, F., Kementsietsidis, A., Duan, S.: Scalable keyword search on large RDF data. IEEE Trans. Knowl. Data Eng. **26**(11) (2014)
15. de Oliveira, P., da Silva, A.S., de Moura, E.S.: Ranking candidate networks of relations to improve keyword search over relational databases. In: IEEE (2015)
16. Sayyadian, M., LeKhac, H., Doan, A., Gravano, L.: Efficient keyword search across heterogeneous relational databases. In: ICDE (2007)
17. Vu, Q.H., Ooi, B.C., Papadias, D., Tung, A.K.H.: A graph method for keyword-based selection of the top-k databases. In: SIGMOD (2008)
18. Yan, Z., Zheng, N., Ives, Z.G., Talukdar, P.P., Yu, C.: Active learning in keyword search-based data integration. VLDB J. **24**(5) (2015)
19. Yu, J.X., Qin, L., Chang, L.: Keyword search in relational databases: A survey. IEEE Data Eng. Bull. **33**(1) (2010)