Exploring heterogeneous data graphs through their entity paths

Nelly Barret[1][0000−0002−3469−4149], Antoine Gauquier[2], Jia Jean Law[3], and Ioana Manolescu[1][0000−0002−0425−2462]

[1] Inria and Institut Polytechnique de Paris, France
nelly.barret@inria.fr, ioana.manolescu@inria.fr
[2] Institut Mines Télécom, France antoine.gauquier@etu.imt-nord-europe.fr
[3] Ecole Polytechnique, France jia-jean.law@polytechnique.edu

Abstract. Graphs, and notably RDF graphs, are a prominent way of sharing data. As data usage democratizes, users need help figuring out the useful content of a graph dataset. In particular, journalists with whom we collaborate [4] are interested in identifying, in a graph, the *connections between entities*, e.g., people, organizations, emails, etc.
We present a novel, interactive method for exploring data graphs through *their data paths connecting Named Entities* (NEs, in short); each data path leads to a tabular-looking set of results. NEs are extracted from the data through dedicated Information Extraction modules. Our method builds upon the pre-existing ConnectionLens platform [4,5] and follow-up work on dataset abstraction [8,9]. The contribution of the present work is in the interactive and efficient approach to enumerate and compute NE paths, based on an algorithm which automatically recommends subpaths to materialize, and rewrites the path queries using these subpaths. Our experiments demonstrate the interest of NE paths and the efficiency of our method for computing them.

Keywords: Data graphs · Graph exploration · Information Extraction

1 Motivation and Problem Statement

Data graphs, including RDF knowledge graphs, as well as Property Graphs (PGs), are often used to represent data. More broadly, *any structured or semi-structured dataset can be viewed as a graph*, having: (*i*) an internal node for each structural element of the original dataset, e.g., relational tuple, XML element or attribute, JSON map or array, URI in an RDF graph; (*ii*) a leaf node for each value in the dataset, e.g., attribute value in a relational tuple, text node or attribute value in XML, atomic (leaf) value in a JSON document, or literal in RDF. The connections between the data items in the original dataset lead to edges in the graph, e.g. parent-child relationships between XML or JSON nodes, edges connecting each relational tuple node with their attributes, etc. In a relational database, primary/foreign keys may lead to more edges, e.g., the node representing an Employee tuple "points to" the Company tuple representing their employer. This graph view of a dataset has been introduced to support unstructured (keyword-based) search on (semi)structured data, since [3,15] and through many follow-up works [20].
Entity-rich graphs Building on this idea, the ConnectionLens system [4,5] has been developed to facilitate, for non-IT users such as data journalists, the exploration of

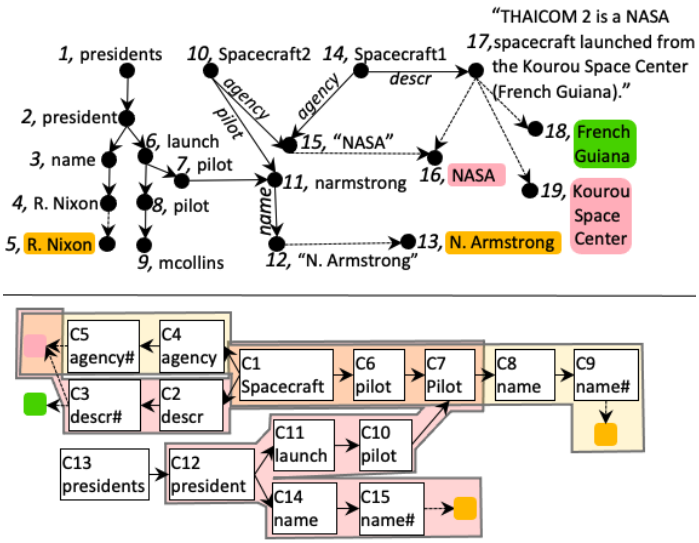

Fig. 1. Sample data graph (top), and corresponding collection graph (bottom) on which paths linking entities are explored (highlighted areas).

datasets of various models, including relational/CSV, XML, JSON, RDF, and PGs. ConnectionLens turns any (set of) datasets into a single graph as outlined above. For instance, the data graph at the top of Fig. 1 features RDF triples about NASA spacecrafts (labeled edges), and an XML document describing presidents who attended spacecraft launches (tree with labeled nodes and unlabeled edges). ConnectionLens also includes Information Extraction modules, which extract, from any leaf node in the data graph, Named Entities (People, Locations and Organizations) [5], as well as other types of entities that journalists find interesting: temporal moments (date, time), Website URIs, email addresses, and hashtags. We designate any of these pieces of information as *entities*, and we model them as extra nodes, e.g., in Fig. 1 (top), organizations appear on a pink background, people on yellow and locations on green, respectively. Each entity is extracted from a leaf text node, to which it is connected by a dashed edge. When an entity is extracted from more than one text node, the edges connecting it to those nodes increase the graph connectivity, e.g., "NASA" extracted from the nodes 15 and 17.

Goal: efficient, interactive exploration of entity connections Journalists are interested in *data paths ending in entity pairs of certain kinds* in a given dataset, e.g. in Fig. 1, "how people are connected to places?". When shown the set of corresponding labeled paths, users may pick one to *further explore*: how many pairs of entities are connected by each path?, which entities are most frequent?, etc. Note that it is important to consider paths *irrespective of the edges directions in the data graph*. This is because, depending on how the data is modeled, we may encounter $x \xrightarrow{\text{boughtProperty}} y \xrightarrow{\text{locatedIn}} c$, or $x \xleftarrow{\text{hasOwner}} y \xrightarrow{\text{locatedIn}} c$; both paths are interesting.

Challenges and contributions The analysis outlined above raises two challenges. First, it requires *materializing the entity pairs connected by the paths*, which may be very costly, if (*i*) the graph is large, and/or (*ii*) there are many paths (the latter is almost

always true, if the data is complex/heterogeneous, and/or if we allow paths to traverse edges in both directions). Second, the large number of paths may *overwhelm users*. Non-expert users, or users which are not familiar with the dataset structure, cannot be expected to state "only the paths that they would like to see", since they lack technical expertise and/or dataset knowledge. However, *if prompted by the system*, they can give valuable input on whether *certain links (or connections) are worth making*, or whether they are just spurious links that would generate uninteresting paths. An example of uninteresting path is: in a dataset of French national assembly members, all addresses are in France, thus the France named entity enables connecting any two people.

Our contributions towards addressing these challenges is as follows. (*i*) From the data, we *generate a small set of user questions*, which help us leverage the domain knowledge that users may have, to generate interesting paths (Sec. 2 and 3). (*ii*) To speed up the materialization of interesting paths, we *recommend a set of views (subpaths) to materialize*, and *rewrite each path query using these materialized views* (Sec. 4). This allows to identify subpaths that appear in more than one path, and compute the corresponding data paths only once, greatly improving performance (Sec. 5).

2 From datasets to data graphs

In order to propose a general approach that works on any data format (e.g. RDF, JSON, XML, etc.), we start by building a *data graph* out of each input dataset (Sec. 2.1). These graphs may be large, thus enumerating paths on them would be inefficient. Therefore, we leverage a more compact structure, namely a *collection graph* (Sec. 2.2). The graph representation and the collection graph are based on prior work [8,9]. Finally, we explain how we produce a single collection graph out of *several datasets* (Sec. 2.3), which journalists may need to exploit together in a particular investigation.

2.1 From a dataset to a data graph

We transform any dataset as a directed graph $G_0 = (N_0, E_0, \lambda_0)$ where N_0 is a set of nodes, E_0 is a set of vertices connecting N_0 nodes and λ_0 is a function assigning a label l to each node and edge (l may be empty). We map each data model on G_0 as follows.
RDF graphs are naturally mapped on G_0: each subject, respectively object, is turned into a node and an edge labelled with the property is connecting them.
An XML document can be naturally viewed as a tree, with element nodes having element and attribute children. XML elements may carry #ID attributes whose values uniquely identify them; other XML elements may carry #IDREF attributes, whose values act as "foreign keys" referring to other elements by their #ID value. ID-IDREF information can be supplied in an optional Document Type Description (DTD) or XML Schema (XSD); when these are not available, ID-IDREF links can be found by profiling [16,1] techniques, which we also implemented. In the graph representation of an XML document, ID-IDREF links lead to more edges between elements (thus, the graph is no longer a tree).
JSON documents are also modelled as trees, where each map, array, and leaf value is a node and parent-child edges are connecting them.

Named entities (NE) are extracted from each leaf of the data graph (RDF literal, XML text node or attribute value, or JSON value). For instance, in Fig. 1, the NASA organization (pink background) is extracted from both the RDF and XML datasets. Each extracted NE is materialized by a new node in the graph, connected via an extraction edge (dashed arrows in Fig. 1) to each leaf text node from which it has been extracted.

2.2 From a data graph to a collection graph

The collection graph is a compact representation of the data graph. It is based on an equivalence relation between the data graph nodes: the collection graph has exactly one node for each equivalence class of data graph nodes; further, whenever $n_1 \rightarrow n_2$ is an edge in the data graph, the collection graph comprises an edge $C_1 \rightarrow C_2$, where C_1, C_2 are the equivalence classes (also called collections) to which n_1, n_2 belong, respectively.

The most natural node equivalence relation differs across data models. Specifically, XML nodes we consider equivalent are elements with the same name; text nodes that are children of equivalent elements; and values of same-name attributes of equivalent elements. For instance, in Fig. 1, the pilot nodes 7 and 8 are equivalent. In JSON, we consider equivalent nodes found on the same labeled path, from the JSON document root, to the node. A path is a concatenation of node and edge labels, separated by . (dots), where we assign special labels: μ to each map node, α to each array node, and ϵ to each empty node or edge label. For instance, in the JSON snippet [{"name": "Alice", "address": {"street": "Main", "city": "NY"}}] the path to "NY" is: $\alpha.\epsilon$.address.μ.city. In RDF, there are numerous ways to define node equivalence [10]. RDF collection graphs are built through the TypedStrong summarization method [12], working as follows. Whenever an RDF node has one or more types, all nodes with the same set of types are said equivalent (in RDF, a node can have several types, related or not, e.g. *Student* and *Employee*). For nodes without types, TypedStrong summarization relies on the properties (labels of incoming and outgoing edges) that the nodes have, by introducing a notion of *outgoing/incoming property cliques*: (*i*) two properties that a node have, are in the same outgoing clique, e.g., agency and pilot are in the same outgoing clique because they are both property of node 10 in Fig. 1, and also with descr because node 14 has agency and *descr*; incoming property cliques are symmetrically defined based on incoming properties; (*ii*) two nodes without types are equivalent if they have identical outgoing and incoming property cliques. For instance, nodes 10 and 14 (the two spacecrafts) are equivalent, since they have the same outgoing and incoming property cliques.

Fig. 1 (bottom) shows the collection graph corresponding to the data graph. We named the collections C_1, C_2, etc. Note that some data models have *labeled* edges, e.g., RDF, while others have (mostly) *unlabeled* edges, e.g., XML. For uniformity, we transform any labeled edge into a node and extend our summarization also to such nodes. In Fig. 1, collection C_6 contains the nodes introduced instead of the pilot edges in the RDF dataset. Collection names in our figure are only for ease of explanation (they are not required in our method). Some collections, such as C_{12}, C_{13}, have nodes with identical names, in which case we use that name. For collections such as C_1, with RDF nodes each with a different URI, we use an intuitive name, e.g., "Spacecraft".

Entity collection profiles Each leaf collection in the collection graph corresponds to a set of literals (strings), out of which various NEs may have been extracted. These collections' names end with a # to help distinguish them (e.g. C_5 agency#). For each

such collection C, we compute an *entity profile* storing how many entities of each type were extracted from its string nodes. In Fig. 1, there are four such profiles, each shown as a box filled with the color of an entity type, e.g., the child of C_5 is pink reflecting the Organization entities extracted from agency values. In practice, long text nodes often lead to multiple NEs extracted, of several types. Knowledge of which leaf collections contain which kind of entities will be crucial to help users explore the graph (Sec. 3).

2.3 From multiple datasets to a collection graph

Journalists often need to work with several datasets, e.g. a JSON collection of political tweets, the list of French mayors in XML, and an RDF graph of public investment in companies across France. Such datasets often have *common values*, e.g., the cities that mayors represent are also the places where companies are situated. Interesting data paths can be found *across* data sources.

To obtain a single collection graph from a set of datasets, we proceed as follows. First, we build a separated collection graph from each dataset (as in Sec. 2.2). Then, whenever two collections C_1, C_2 from distinct datasets share values, we replace them by a new collection $C_{1,2}$, which contains the values of C_1 and C_2, and inherits all the incoming and outgoing edges of C_1 and C_2 in the collection graph they came from. Their original collection graphs are thus connected, and the entity profile of the new collection $C_{1,2}$ is built. In Fig. 1, the collection graph reflects this unification: the pink filled node reflects organizations from both RDF and XML.

3 Paths between entities

In this section, we discuss categories of paths that users might be interested in, and how to ask for their input.

An important first observation is: by the way we built the collection graph, *to each dataset path corresponds a path in the collection graph*. For instance, consider the data path $13 \leftarrow 12 \xleftarrow{\text{name}} 11 \xleftarrow{\text{pilot}} 10 \xrightarrow{\text{agency}} 15 \rightarrow 16$ in Fig. 1. The collection graph features the corresponding path ■ $\leftarrow C_9 \leftarrow C_8 \leftarrow C_7 \leftarrow C_6 \leftarrow C_1 \rightarrow C_4 \rightarrow C_5 \rightarrow$ ■.

Further, *some paths in the collection graph correspond to no paths in the data graph*. For instance, the path $C_6 \leftarrow C_1 \rightarrow C_2$ does not correspond to any path in the data, because no spacecraft (part of the collection C_1) has *both* the agency and descr properties. Such collection paths, with no support in the data, are introduced by summarization, which compresses the graph structure with some information loss. In our example, the fact that a spacecraft has agency and pilot, another has pilot and descr, and none has agency and descr is "simplified" into a collection C_1 that may have any combination of the three properties (represented by collections C_2, C_4, C_6). We consider this loss of information acceptable in exchange for the ability to work on a much smaller object (collection graph) instead of a potentially very large data graph.

Based on the above, our approach will be to (*i*) enumerate paths on the collection graph, then (*ii*) turn each path into a query, and (*iii*) evaluate this query on the data graph, and show users the resulting actual data connections (if any).

3.1 Characterizing entity paths in the collection graph

Each path between two entities is first, characterized by a **pair of entity types** of the form (τ_1, τ_2), where $\tau_1, \tau_2 \in \mathcal{E}$, with $\mathcal{E}$ being the set of supported entity types. $\mathcal{E}$ contains entity types such as Person, Location, Organization, Email, URI, Hashtag, Date, etc.

An entity path is also characterized by its **length**, i.e., the number of edges it contains. Depending on the application, interesting connections can be made by paths of different lengths; however, it appears likely that beyond a length such as 10 or 15, connections may become meaningless. Therefore, and also to control how many collection paths they want to inspect, users may specify a **maximum path length** L_{max}, whose default value we set to 10.

Path directionality By definition, each entity-to-entity collection path cp is of the form: $\square \leftarrow C_1 \leftrightsquigarrow C_2 \rightarrow \blacksquare$, where $\square, \blacksquare$ are two entity profiles, such that the first, respectively, the second, contains some entities of types τ_1, respectively, τ_2, and C_1, C_2 are value collections such as C_5 and C_9 in the example at the beginning of Sec. 3. The directions of the leftmost and rightmost edges are by convention always towards $\square, \blacksquare$, which represent entities. Let cp_0 denote the path $C_1 \leftrightsquigarrow C_2$. This path may be:

- *Unidirectional*, i.e., all cp_0 edges go from C_1 towards C_2, or the opposite;
- *Shared-sink*, i.e., cp_0 may contain a (collection) node C such that all edges between C_1 and C (if any) go from C_1 towards C, and all edges between C_2 and C (if any) go from C_2 towards C. A shared-sink path is $C_1 \rightarrow C_6 \rightarrow C_7 \leftarrow C_{10} \leftarrow C_{11}$.
- *Shared-root*, i.e., cp_0 may contain a (collection) node C such that all edges between C and C_1 (if any) go from C towards C_1, and all edges between C and C_2 (if any) go from C towards C_2. A shared-root path is $C_9 \leftarrow C_8 \leftarrow C_7 \leftarrow C_6 \leftarrow C_1 \rightarrow C_4 \rightarrow C_5$.
- *General*, i.e., the edges may be in any direction.

Unidirectional paths are quite rare. This is because entity-connecting paths must have at each end a node from which an entity is extracted. Most of the time, these are *two literal (string) nodes* (as opposed to internal nodes structuring the dataset). Literals have incoming edges, but not outgoing ones (other than those towards extracted entities); thus, *there is no unidirectional path from a literal to another*. However, in some RDF datasets, *NEs are extracted from URIs*, e.g., the triple https://dbpedia.org/Facebook $\xrightarrow{\text{locatedIn}}$ http://dbpedia.org/California is a unidirectional data path from an Organization to a Location. Similarly, shared-sink paths only occur when nodes in C_1 and C_2 have outgoing edges, and NEs appear in their labels; this only happens in RDF URIs.

Low-specificity connections Some edges in the data graph reflect connections that can be seen as weak, or *non-specific*. In details, let's first consider data edges with non-empty labels, e.g., RDF triples. Let e be the edge $n_1 \xrightarrow{a} n_2$ for some URIs n_1, a, n_2. The *specificity* of e, denoted e_s, is computed as $2/(N_{1,a} + N_{2,a})$, where $N_{1,a}$, $N_{2,a}$ are the numbers edges labeled a outgoing, resp. incoming n_1, resp. n_2 [5]. The highest $N_{1,a}$ and/or $N_{2,a}$, the lowest e_s. For instance, the specificity of each agency edge in Fig. 1 is $2/(1 + 2) = 2/3$. For our purposes, we extend specificity to unlabeled edges as follows: the specificity of an edge $n_1 \xrightarrow{\epsilon} n_2$ is $2/(1 + n_{1,2})$ where $n_{1,2}$ is the number of ϵ (empty-label) edges outgoing n_1, towards nodes having the same label as n_2. For instance, the specificity of the edge between nodes 6 and 7 is also $2/3$.

In the collection graph, the edges with a non-empty label, connecting nodes from two equivalence classes lead to a collection, e.g., agency triples lead to C_4. We attach to this *collection*, *the average specificity of all the data edges it comes from*, e.g., to C_4 we attach 2/3. Empty-label edges connecting graph nodes from two equivalence classes lead to an edge in the collection graph, e.g., $C_{11} \rightarrow C_{10}$. We attach the average specificity of the original edges to this *edge* between two collections.

3.2 Eliciting user input and collection path enumeration

First, we ask users which two entity types they want to connect (thus selecting τ_1, τ_2). Next, we ask the maximum path length L_{max} (with a default value, currently 10). Then, we ask *how many types of data edges they are willing to review*, in order to possibly decide to disallow paths to traverse such connections; the default value is 5. By *type of data edge*, we mean either an edge label, such as agency, or a pair of (parent name, child name), in case the edge is unlabeled, such as launch-pilot in Fig. 1. Once the user chooses this value, say n_{spec}, we show her successively the n_{spec} collections (or collection graph edges) with the lowest average specificity, and ask if such a collection (or edge) should be excluded from the paths we search for, or not.

Finally, based on the collection graph, we *enumerate all the paths connecting entities* of types τ_1, τ_2, of maximum length L_{max}, and respecting the exclusion constraints which users may have specified. We use a simple dynamic programming algorithm, running in memory, as the collection graph is typically much smaller than the data. We then inform the user: "There are N_{uni} unidirectional paths, N_{sink} shared-sink paths, N_{root} shared-root paths, and N_{gen} general paths between entities of type τ_1 and τ_2." The user can then either (*i*) chose to materialize one of these path sets, or (*ii*) change settings, e.g., L_{max}, n_{spec}, exclude more or less edge types, etc., until the user is satisfied with the predicted number of paths and decides to trigger materialization of a path set (Sec 4).

4 Materializing data paths

At this point, we have a set of *collection paths*, which must be transformed into queries and evaluated *on the data graph*. Each such query matches similar-structure data paths, thus its results are shown to users as a table: the first and last attribute of such a table comprise entities of type τ_1, τ_2, while the intermediary attributes are the nodes and edges connecting these entities in the data graph. For instance, let τ_1 be Person, τ_2 be Organization: the light-yellow, respectively, light-pink background shapes in Fig. 1 materialize the two paths which, in this graph, connect the pink child of C_5 (■) with the yellow children (■) of C_9, respectively, C_{15}.

4.1 From a collection path to a query over the data graph

Each collection path translates into a chain-shaped conjunctive query. For instance, the path on yellow background in Fig. 1, going through C_5 and C_9, becomes:

$$q_1(\bar{x}) \text{ :- } n(x_1, \tau_{\text{Org}}, \blacksquare), e(x_2, x_1, \_), n(x_2, \_, C_5), e(x_3, x_2, \text{agency}), n(x_3, \_, C_1),$$
$$e(x_3, x_4, \text{pilot}), n(x_4, \_, C_7), e(x_4, x_5, \text{name}), n(x_5, \_, C_9), e(x_5, x_6, \_),$$
$$n(x_6, \tau_{\text{Person}}, \blacksquare)$$

This query refers to two relations: $n(ID, type, equiv)$, describing nodes, with the last attribute denoting their equivalence class, and $e(s, d, label)$, describing edges between nodes s and d and carrying a certain label. Each x_i is a variable; $\bar{x}$ in the query

head denotes all the x_i variables, $1 \leq i \leq 6$. We use _ to denote a variable which only appears once, in a single query body atom. Finally, τ_{Org} and τ_{Person} denote the node types of extracted Organization, respectively, extracted Person entity. Similarly, the pink-background collection path translates into:

$$q_2(\bar{x}) \text{ :- } n(x_1, \tau_{\text{Org}}, \blacksquare), e(x_2, x_1, \_), n(x_2, \_, C_3), e(x_3, x_2, \mathsf{descr}), n(x_3, \_, C_1),$$
$$e(x_3, x_4, \mathsf{pilot}), n(x_4, \_, C_7), e(x_5, x_4, \_), n(x_5, \_, C_{11}), e(x_6, x_5, \_), n(x_6, \_, C_{12}),$$
$$e(x_6, x_7, \_), n(x_7, \_, C_{14}), e(x_7, x_8, \_), n(x_8, \_, C_{15}), e(x_8, x_9, \_), n(x_9, \tau_{\text{Person}}, \blacksquare)$$

Each of these queries can be evaluated through any standard graph database. However, evaluating dozens or hundreds of path queries on large graphs can get very costly. Further, since we do not know which paths may result from the user choices, we cannot establish path indexes beforehand.

View-based optimization To address this problem, we propose an optimization, based on the observation that *queries resulting from collection paths may share some subpaths*. For instance, the subquery $s(x_3, x_4)$:- $n(x_3, \_, C_1), e(x_3, x_4, \mathsf{pilot}), n(x_4, \_, C_7)$ is shared by $q_1(\bar{x})$ and $q_2(\bar{x})$. Therefore, we decide to (*i*) evaluate s and store its results; (*ii*) rewrite $q_1(\bar{x})$ and $q_2(\bar{x})$ by replacing these atoms in each query, by a single occurrence of the atom $s(x_3, x_4)$. The next sections formalize this for larger query sets, also showing how to handle different *alternatives* that may arise as to which shared subpaths to materialize.

4.2 Enumerating candidate views

A first question we need to solve is enumerating, based on a set $\mathcal{Q}$ of path queries, the possible subqueries that we could materialize, and based on which we could rewrite some workload queries.

Let $q \in \mathcal{Q}$ be a path query: it is an alternating sequence of node (n) and edge (e) atoms. We denote by n_q the number of edge atoms, then the number of node atoms is $n_q + 1$. We denote by $n_{\mathcal{Q}}$ the highest n_q over all $q \in \mathcal{Q}$.

Without loss of generality, our first heuristic **(H1)** is: we only consider **connected subpaths** of q as candidate subqueries. If q is of the form $q(\bar{x})$:- $n_1, e_1, \ldots, e_{n_q}, n_{n_q+1}$, each connected subpath of q, denoted sq, is determined by two integers $1 \leq i \leq n_q$, $i < j \leq n_q + 1$, such that $sq(x_i, x_j)$:- $n_i, e_i, \ldots, n_j, e_j$, and x_i, x_j are the IDs of the nodes in the atoms n_i, n_j, respectively. We denote by $q|^{i,j}$ the subquery of q determined by the positions i, j. For instance, when q_1 is the sample query in Sec. 4.1, $q_1|^{3,4}$ is the subquery $s(x_3, x_4)$ introduced there. Considering connected (cartesian-product free) candidate views is common in the literature, too (see Sec. 6).

Each query $q \in \mathcal{Q}$ has $O(n_q^2)$ connected subpaths, that can be easily enumerated from q's syntax. A second heuristic **(H2)** we adopt is: we only consider **shared subpaths**, that is, those subpaths s for which there exist $q', q'' \in \mathcal{Q}$, $q' \neq q''$, and integers i', j', i'', j'' such that $s = q'|^{i',j'} = q''|^{i'',j''}$, possibly after some variable renaming. For the queries q_1, q_2 in Sec. 4.1, the subquery $s_{3,4}$ is $q_1|^{3,4}$ and also $q_2|^{3,4}$. (H2) restricts the number of candidate views from $|\mathcal{Q}| \times n_{\mathcal{Q}}^2$ to a number that depends on the actual workload $\mathcal{Q}$, and which decreases when $\mathcal{Q}$ paths look more like each other. Another interest of (H2) is: the benefit of using a view v to rewrite one query q is likely offset by the cost of materializing v; actual performance improvements start when v is *used twice (or more)*, which is exactly the case for subqueries shared by several $\mathcal{Q}$ queries.

Algorithm 1: Selecting views to materialize and the respective view-based rewritings

Input : Queries $\mathcal{Q}$, candidate materialized views $\mathcal{V}$
Output: Materialized views $\mathcal{M}$ and rewritings $\mathcal{R}$ for some $\mathcal{Q}$ queries

$\mathcal{M} \leftarrow \emptyset$; $\mathcal{R} \leftarrow \emptyset$
while $\mathcal{V} \neq \emptyset$ **do**
  for $v \in \mathcal{V}$ **do**
    $ben(v) \leftarrow 0$; $cost(v) \leftarrow$ cost to compute and store the view v
    for $q \in \mathcal{Q}$, q can be rewritten using v **do**
      $(ben(v, q), r_{q,v}) \leftarrow$ the cost of evaluating q based on v, through the rewriting $r_{q,v}$, minus the cost of evaluating q directly on the graph
      $ben(v) \leftarrow ben(v) + ben(v, q)$
  $(v_{max}, b_{max}) \leftarrow$ a view v_{max} maximizing $ben(v) - cost(v)$, and its benefit
  if $b_{max} - cost(v_{max}) < 0$ **then**
    exit
  Add v_{max} to $\mathcal{M}$
  for $q \in \mathcal{Q}$, *q can be rewritten using* v_{max} **do**
    if $ben(v_{max}, q) > 0$ **then**
      Add $r_{q,v_{max}}$ to $\mathcal{R}$
      Remove q from $\mathcal{Q}$
  Remove v_{max} from $\mathcal{V}$

Our third heuristic **(H3)** is: among the possible subqueries shared by two queries q', q'', **consider only the longest ones**. That is, if s_1, s_2 are two shared subqueries of q' and q'' such that $n_{s_1} > n_{s_2}$, do not consider the subquery s_2.

Our heuristics (H1), (H2), (H3) lead to **building the candidate view set** $\mathcal{V}$ as follows. For each pair of distinct queries (q', q'') where $q', q'' \in \mathcal{Q}$, add to $\mathcal{V}$ the longest, shared, connected subqueries of q' and q''. The complexity of this algorithm is $O(|\mathcal{Q}|^2 \times n_{\mathcal{Q}}^2)$, while $|\mathcal{V}|$ is in $O(|\mathcal{Q}|^2)$.

4.3 Selecting materialized views and rewriting path queries

Knowing the path queries $\mathcal{Q}$ and the candidate view set $\mathcal{V}$, we need to determine: a set $\mathcal{M} \subseteq \mathcal{V}$ of views which we actually materialize, in order to rewrite some $\mathcal{Q}$ queries. We collect the rewriting of each such queries in $\mathcal{R}$. The decision to materialize a view incurs a *cost*, since the view data must be computed and stored. We denote $cost(\cdot)$ the cost of evaluating a view (or query), and assume it can be computed without actually computing it. Materializing a view is more attractive if (*i*) rewritings using it reduce significantly query evaluation costs, and (*ii*) its own materialization cost is small.

In the most general case, a query could be rewritten based on any number of views, and also involving the base graph. For instance, query q_1 from Sec. 4.1 could be rewritten as: $q_1|^{1,3} \bowtie q_1|^{3,4} \bowtie q_1|^{4,6}$, where each $\bowtie$ denotes a natural join, on the variables x_3, respectively, x_4. However, enumerating all such alternatives makes the query rewriting problem NP-hard [14]. Instead, we adopt another pragmatic heuristic **(H4)**: **rewrite**

each query using not more than one view. This simple choice keeps the view selection complexity under control, all the while providing good performance.

Algorithm 1 depicts our **greedy** method for finding $\mathcal{M}$ and $\mathcal{V}$. It computes the *benefit* of each view v for each query that may be rewritten using v, as well as the cost of v. In a greedy fashion, it decides to materialize the view v_{max} maximizing the *overall* benefit (for all Q queries), and uses it to rewrite all queries whose evaluation cost can be reduced thanks to v_{max}, via the rewriting $r_{q,v_{max}}$. These queries are then removed from Q, the benefits of the remaining views are recomputed over the diminished Q, and the process repeats until no profitable view to materialize can be found.

Estimating costs Algorithm 1 needs to compute: (*i*) $cost(\cdot)$, the cost to evaluate a query q or materialize a view v; (*ii*) $r_{q,v}$, the rewriting of q using a view v; and (*iii*) $cost(r_{q,v})$, the cost of such a rewriting. All these costs must be estimated *before* any query or view results are computed. We do this as follows. For (*i*), we *use the cost estimation of the graph data management system* (GDBM, in short) *storing the graph*. Our implementation relies on PostgreSQL, whose `explain` command returns both the estimated number of results of certain query (or view), denoted $size(q)$, and the cost of computing those results. For (*ii*), recall (Sec. 4.2) that when v is used to rewrite q, v is a subpath of q, thus there exist i, j such that $v = q^{|i,j}$. The rewriting $r_{q,v}$ is easily obtained by replacing, in the body of q, the atoms from the ith to the jth, with the head of v. Handling (*iii*) is more complex than (*i*). This is because the cost of a query (or view) is estimated based on *statistics* the GDBM has about the stored graph. In contrast, the GDBM cannot estimate cost of $r_{q,v}$, because *v has not been materialized* yet, thus the GDBM cannot reason about v like it does about the graph. To compensate, we proceed as follows: we compute the cost of reading the hypothetical view v_{max} from the database, by multiplying $size(v_{max})$, the estimation of the view size, with a constant (we used Postgres' own CPU_TUPLE_COST); then, we estimate the cost of $r_{q,v_{max}}$ as this reading cost plus the cost of estimating the parts of q not in v_{max} plus the cost of joining v_{max} with these (one or two) remaining query parts. We estimate the cost of each such join by adding their input sizes, which we then multiply with another (GDBM) constant. This reflects the fact that modern databases feature efficient join algorithms, such as memory-based hash joins, whose complexity is linear in the size of their inputs.

Complexity Algorithm 1 makes at most $O(|\mathcal{V}| \times |Q|)$ iterations, which can be simplified into $O(|Q|^3)$. Forming a rewriting takes $O(n_Q)$, bringing the total to $O(|Q|^3 \times n_Q)$.

Impact of heuristics As previously discussed, (H1) is universally adopted in the literature: no candidate view features cartesian products. (H2), imposing that views benefit at least two queries, preserves result quality, i.e., cost savings, under every *monotone cost model*, ensuring that the cost of evaluating a query q is at least that of evaluating s, when s is a subquery of q. In contrast, (H3) and (H4) may each divert from the globally optimal solution. However, as our experiments show, our chosen rewritings perform well in practice, and the algorithm itself is very efficient.

5 Experimental evaluation

Our approach is fully implemented in Java 11, on top of ConnectionLens [4,5] which builds the data graph (Sec. 2.1) and Abstra [8,9] which builds the collection graph (Sec. 2.2); these are stored in PostgreSQL. We experimented on a Linux server with an

Dataset name	$\|N\|$	$\|E\|$	$\|\tau_P\|$	$\|\tau_L\|$	$\|\tau_O\|$	$min(e_s)$
PubMed	63,052	89,710	5,993	2,151	5,096	0.001
Nasa	59,408	128,068	634	690	4,530	0.0002
YelpBusiness	57,963	61,627	322	427	1,437	0.001
YelpBusiness4	229,949	247,074	1,099	1,230	4,199	0.0002

Table 1. Dataset overview.

Intel Xeon Gold 5218 CPU @ 2.30 GHz and 196GB of RAM. We used PostgreSQL v9.6. Our system is available at: `https://team.inria.fr/cedar/projects/pathways/`. Our evaluation seeks to answer the two following questions: (*i*) *how are entities connected in each dataset?* (Section 5.2) and (*ii*) *does Algorithm 1 help to evaluate paths queries over the data graph?* (Section 5.3).

5.1 Datasets and settings

We present experiments on an XML, an RDF and a JSON dataset. They all come from real-life applications (as opposed to synthetic) to stay close to application needs, and to ensure realistic Named Entities (NEs). Indeed, synthetic datasets are often generated with an interest on structure, while the leaf (text) values lack interesting information.

We used an **XML PubMed** subset describing scientific articles from PubMed, a database of biomedical publications. Each article is described by its title, journal, link, year, DOI, keywords list and authors list. Authors are identified by their name and their affiliation; they may declare their conflicts of interest in the `<COIStatement>` tag. We used the **RDF Nasa** dataset, describing NASA flights, spacecrafts involved in launches, related space missions and the participating agencies. Finally, we used the **JSON YelpBusiness** dataset where each business has an id, a name, an address, a city, a state, a postal code and coordinates (latitude and longitude). It also has a set of categories (e.g. bakery, shoe store, etc.) and a set of attributes (e.g. `acceptCreditCards`), the latter may be deeply nested. They also received reviews from customers modeled as a number of stars (from 0 to 5) and the number of reviews. **YelpBusiness4**, 4 times larger than YelpBusiness, allows studying the scalability of our algorithm. Tab. 1 shows for each dataset: its number of nodes $|N|$, edges $|E|$, numbers of extracted NEs $|\tau_P|$, $|\tau_L|$, $|\tau_O|$ and the minimum edge specificity $min(e_s)$. Without loss of generality, we experiment with the NE types Person, Location, Organization, whose types are denoted τ_P, τ_L, τ_O, respectively. We set L_{max} to 10, and avoided connections whose assigned specificity (Sec. 3) was less than 5% of the average specificity over all node/edge collections.

5.2 Path enumeration

For each dataset and pair of entity types, Fig. 2 and 3 report the number of paths of each directionality (Sec. 3.1), the minimum and maximum length L_p of each path, and the minimum and maximum data path support (number of results when evaluated on the data), this is denoted S_p. For the PubMed (XML) and YelpBusiness (JSON) datasets, we obtained only shared-root paths: this is because of the tree structure of these datasets, where text values (leaves) are only connected by going through a common ancestor node. In the RDF Nasa dataset, we also found general-directionality paths. The JSON datasets are more irregular, leading to more paths. In almost every case, a few collection paths had 0 support, due to dataset summarization (Sec. 3). The maximum support may be high, e.g. 15,181 in the PubMed dataset.

(τ_1, τ_2)	N_{root}	min(S_p)	max(S_p)
(τ_P, τ_O)	21	0	13,988
(τ_P, τ_L)	21	0	15,181
(τ_L, τ_O)	21	0	5,054
(τ_P, τ_P)	21	0	389
(τ_L, τ_L)	21	0	1,214
(τ_O, τ_O)	21	3	3,090

(τ_1, τ_2)	N_{root}	N_{gen}	min(S_p)	max(S_p)
(τ_P, τ_O)	99	1	0	629
(τ_P, τ_L)	95	5	0	137
(τ_L, τ_O)	97	3	0	603
(τ_P, τ_P)	97	3	0	89
(τ_L, τ_L)	97	3	0	3,050
(τ_O, τ_O)	97	3	0	8,960

Fig. 2. Entity paths in PubMed (left) and Nasa (right). PubMed: $min(L_p) = 5$; $max(L_p) = 8$. Nasa: $min(L_p) = 5$; $max(L_p) = 9$.

(τ_1, τ_2)	N_{root}	max(L_p)	min(S_p)	max(S_p)
(τ_P, τ_O)	41	7	0	651
(τ_P, τ_L)	33	7	0	193
(τ_L, τ_O)	21	5	0	1,412
(τ_P, τ_P)	28	7	0	35
(τ_L, τ_L)	15	5	2	158
(τ_O, τ_O)	21	5	0	1,232

(τ_1, τ_2)	N_{root}	max(L_p)	min(S_p)	max(S_p)
(τ_P, τ_O)	48	7	0	2,593
(τ_P, τ_L)	39	7	0	760
(τ_L, τ_O)	39	5	0	258
(τ_P, τ_P)	36	7	0	207
(τ_L, τ_L)	15	5	0	674
(τ_O, τ_O)	21	5	0	4,889

Fig. 3. Entity paths in the YelpBusiness (left) and YelpBusiness4 (right) datasets. YelpBusiness and YelpBusiness4: $min(L_p) = 5$.

(τ_1, τ_2)	T_0	$\lvert Q_{TO}\rvert$	$\lvert Q_{NV}\rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{M}\rvert$	T_R	$T_{Q_{NV}}$	$T = T_R + T_{Q_{NV}}$	$s = T_0/T$
PubMed									
(τ_P, τ_O)	250.36	5	1	16	5	3.78	0.32	4.10	61×
(τ_P, τ_L)	37.29	0	1	16	5	19.06	0.32	19.38	2×
(τ_L, τ_O)	151.29	2	2	16	5	11.88	8.59	20.47	7×
(τ_P, τ_P)	152.59	3	1	16	5	44.19	0.08	44.27	3×
(τ_L, τ_L)	169.64	2	1	16	5	71.32	0.31	71.63	2×
(τ_O, τ_O)	317.92	5	1	16	5	22.99	0.25	23.24	13×
Nasa									
(τ_P, τ_O)	195.47	1	0	80	10	54.14	N/A	54.14	3×
(τ_P, τ_L)	254.26	3	0	68	10	44.57	N/A	44.57	5×
(τ_L, τ_O)	1073.55	32	0	77	9	131.58	N/A	131.58	8×
(τ_P, τ_P)	278.95	4	0	76	10	92.01	N/A	92.01	3×
(τ_L, τ_L)	1103.48	30	0	77	9	101.35	N/A	101.35	10×
(τ_O, τ_O)	1318.78	37	0	77	9	247.43	N/A	247.43	5×

Table 2. Data paths evaluation on the PubMed and Nasa datasets.

These results show that numerous interesting entity paths exist in our datasets, of significant length (up to 9), and some with high support, bringing the need for an efficient evaluation method.

5.3 Efficiency of path evaluation

We now study the efficiency of data path computations over the graph. Tab. 2 and Tab. 3 show, for each dataset and entity type pair, T_0 is the time to evaluate the corresponding path queries without the view-based optimization of Sec. 4.2 and 4.3, referred to as **VBO** from now on. $|Q_{TO}|$ is the number of queries whose execution we stopped (timeout of 30s) without VBO. $|Q_{NV}|$ is the number of queries for which Algo. 1 did not recommend a view. T_R is the time to evaluate the rewritten queries on the data graph,

(τ_1, τ_2)	T_0	$\lvert Q_{TO}\rvert$	$\lvert Q_{NV}\rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{M}\rvert$	T_R	$T_{Q_{NV}}$	$T = T_R + T_{Q_{NV}}$	$s = T_0/T$
YelpBusiness									
(τ_P, τ_O)	205.95	2	0	22	6	4.20	N/A	4.20	49×
(τ_P, τ_L)	410.87	7	1	19	5	40.87	1.27	42.12	9×
(τ_L, τ_O)	239.90	0	1	20	10	1.15	0.6	1.75	137×
(τ_P, τ_P)	466.58	9	2	23	5	15.33	12.02	27.35	17×
(τ_L, τ_L)	450.00	15	1	8	4	9.89	< 0.01	9.89	45×
(τ_O, τ_O)	334.22	4	1	10	5	2.83	< 0.01	2.83	118×
YelpBusiness4									
(τ_P, τ_O)	804.70	26	0	23	6	62.52	N/A	62.52	12×
(τ_P, τ_L)	454.19	10	1	20	5	92.50	< 0.01	92.50	5×
(τ_L, τ_O)	242.57	5	1	10	5	62.74	6.61	69.35	3×
(τ_P, τ_P)	317.00	7	1	27	7	14.35	1.08	15.43	20×
(τ_L, τ_L)	395.49	10	1	8	4	2.62	18.15	20.77	19×
(τ_O, τ_O)	347.23	8	1	10	5	42.93	2.34	45.27	7×

Table 3. Data path evaluation on the YelpBusiness datasets.

while $T_{Q_{NV}}$ is the time to evaluate the non-rewritten queries Q_{NV}; $T = T_R + T_{Q_{NV}}$ is the (total) execution time to evaluate queries using VBO. Finally, $s = T_0/T$ is the speed-up due to VBO. We do not report times to materialize views because they were all very short (less than 0.01s). All times are in seconds.

The evaluation time T_0 without VBO ranges from 100s to 2000s; these path queries require 5 to 9 joins, on graphs of up to more than 200,000 edges (Tab. 1). $|Q_{NV}|$, the number of queries that could not make use of any views, is rather small, which is good. The number of candidate views, respectively, materialized views depend on the complexity of the dataset, and thus on the complexity of the paths. The total path evaluation time T is reasonable. Finally, the VBO speed-up is at least 2× and at most 137×, showing that our view-based algorithm allows to evaluate path queries much more efficiently.

5.4 Experiment conclusion

On graph leading to entity paths of various lengths and support, the view-based rewriting significantly reduces the path query evaluation time over the data graph.

6 Related work and conclusion

Many graph exploration methods exist, see, e.g., [18]. Modern graph query languages such as GPML [11] (no implementation so far) or the JEDI [2] SPARQL extension allow *checking* for paths between query variables, if users can specify a regular expression that the path labels match. Other systems interact with users to incrementally build SPARQL queries, In keyword-based search (KBS, in short) [3,5,20], one asks for connections between two or more nodes matching specific keywords. KBS is handy when users *know keywords (entities) to search for*. In [6], graph queries are extended with a KBS primitive. The algorithms proposed there work directly on the graph; finding such trees in general is NP-hard, since it is related to the Group Steiner Tree problem. Complementary to these, we focus on *identifying, and efficiently computing, all paths between certain extracted entities*, to give a first global look at the dataset content, for graphs obtained from multiple data models.

Our view selection problem is a restriction (to path-only queries) of those considered, e.g., in [13,14,17,19]. This allows for its low complexity, while being very effective. Our algorithms rely on collection graphs built by Abstra [8,9], the interactive path enumeration approach, including VBO, is novel; it has been demonstrated in [7].

Acknowledgments This work has been funded by the DIM RFSI PHD 2020-01 project and the AI Chair SourcesSay (ANR-20-CHIA-0015-01) chair.

References

1. Abedjan, Z., Golab, L., Naumann, F., Papenbrock, T.: Data Profiling. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2018)
2. Aebeloe, C., Setty, V., Montoya, G., Hose, K.: Top-K Diversification for Path Queries in Knowledge Graphs. In: ISWC Workshops (2018)
3. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A system for keyword-based search over relational databases. In: ICDE (2002)
4. Anadiotis, A., Balalau, O., Bouganim, T., et al.: Empowering investigative journalism with graph-based heterogeneous data management. IEEE DEBull. (2021)
5. Anadiotis, A., Balalau, O., Conceicao, C., et al.: Graph integration of structured, semistructured and unstructured data for data journalism. Inf. Systems **104** (2022)
6. Anadiotis, A.C., Manolescu, I., Mohanty, M.: Integrating Connection Search in Graph Queries. In: ICDE (Apr 2023)
7. Barret, N., Gauquier, A., Law, J.J., Manolescu, I.: Pathways: entity-focused exploration of heterogeneous data graphs (demonstration). In: ESWC (2023)
8. Barret, N., Manolescu, I., Upadhyay, P.: Abstra: toward generic abstractions for data of any model (demonstration). In: CIKM (2022)
9. Barret, N., Manolescu, I., Upadhyay, P.: Computing generic abstractions from application datasets. In: EDBT (2024)
10. Cebiric, S., Goasdoué, F., Kondylakis, H., Kotzinos, D., Manolescu, I., Troullinou, G., Zneika, M.: Summarizing Semantic Graphs: A Survey. The VLDB Journal **28**(3) (Jun 2019)
11. Deutsch, A., Francis, N., Green, A., Hare, K., Li, B., Libkin, L., et al.: Graph pattern matching in GQL and SQL/PGQ. In: SIGMOD (2022)
12. Goasdoué, F., Guzewicz, P., Manolescu, I.: RDF graph summarization for first-sight structure discovery. The VLDB Journal **29**(5) (Apr 2020)
13. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: View selection in semantic web databases. PVLDB **5**(2) (2011)
14. Halevy, A.Y.: Answering queries using views: A survey. VLDB J. **10**(4) (2001)
15. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword proximity search on XML graphs. In: ICDE (2003)
16. Jiang, L., Naumann, F.: Holistic primary key and foreign key detection. J. Intell. Inf. Syst. **54**(3) (2020)
17. Le, W., Kementsietsidis, A., Duan, S., et al.: Scalable multi-query optimization for SPARQL. In: ICDE (2012)
18. Lissandrini, M., Mottin, D., Hose, K., Pedersen, T.B.: Knowledge graph exploration systems: are we lost? In: CIDR. www.cidrdb.org (2022)
19. Mistry, H., Roy, P., Sudarshan, S., Ramamritham, K.: Materialized view selection and maintenance using multi-query optimization. SIGMOD Rec. **30**(2), 307–318 (may 2001)
20. Yang, J., Yao, W., Zhang, W.: Keyword search on large graphs: A survey. Data Sci. Eng. **6**(2) (2021)