Graph integration of structured, semistructured and unstructured data for data journalism

Angelos Christos Anadiotis, Oana Balalau, Catarina Conceicao, Helena Galhardas, Mhd Yamen Haddad, Ioana Manolescu, Tayeb Merabti, Jingmao You

Graph integration of structured, semistructured and unstructured data for data journalism

Angelos Christos Anadiotis[a], Oana Balalau[b], Catarina Conceição[c], Helena Galhardas[c], Mhd Yamen Haddad[b], Ioana Manolescu[b], Tayeb Merabti[b], Jingmao You[b]

[a]*École Polytechnique, Institut Polytechnique de Paris and EPFL*
[b]*Inria, Institut Polytechnique de Paris*
[c]*INESC-ID and IST, Univ. Lisboa, Portugal*

Abstract

Digital data is a gold mine for modern journalism. However, datasets which interest journalists are extremely heterogeneous, ranging from highly structured (relational databases), semi-structured (JSON, XML, HTML), graphs (e.g., RDF), and text. Journalists (and other classes of users lacking advanced IT expertise, such as most non-governmental-organizations, or small public administrations) need to be able to make sense of such heterogeneous corpora, even if they lack the ability to define and deploy custom extract-transform-load workflows, especially for dynamically varying sets of data sources.

We describe a complete approach for integrating dynamic sets of heterogeneous datasets along the lines described above: the challenges we faced to make such graphs useful, allow their integration to scale, and the solutions we proposed for these problems. Our approach is implemented within the ConnectionLens system; we validate it through a set of experiments.

Keywords: Data journalism, heterogeneous data integration, information extraction

1. Introduction

Data journalism [22], according to Wikipedia[1], is "a brand of journalism reflecting the increased role that numerical data is used in the production and

[1] `https://en.wikipedia.org/wiki/Data_journalism`

distribution of information in the digital era". Data journalists often have to analyze and exploit datasets that they obtain from official organizations or their sources, extract from social media, or create themselves (typically Excel or Word-style). For instance, journalists from the French Le Monde newspaper want to retrieve *connections between elected people at Assemblée Nationale and companies that have subsidiaries 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 considerable effort may still miss connections that could be found if one added information about politicians' and business people's spouses, information sometimes available in public knowledge bases such as DBPedia, or journalists' notes.

Since 2013 [21], and later in the ContentCheck (2015-2020) and then in the SourcesSay (2020-2024) projects, we have been investigating ways in which Computer Science research could benefit data journalism, as well as journalistic fact-checking, that is: the task of verifying claims contained in media articles, based on some trusted evidence. The platforms we initially developed and proposed to journalists [21, 6] followed a data integration [14] approach, specifically, a mediator (or polystore [29, 3]) architecture. These systems allowed users to express queries using a mix of languages fitting every individual data source. Demonstrating these tools to journalists from Le Monde, a leading French newspaper and partner in the above two projects, as well as others working at Ouest France, the Financial Times etc. highlighted that installing and maintaining a mediator over several data sources, as well as querying them through a mixed language, appeared way too complex and required too many IT resources. Clearly, journalists need **simpler systems** and **simpler, more intuitive methods** for querying their varied data sources. From these exchanges, and learning how they work, we have drawn the following set of requirements and constraints:

R1. Integral source preservation and provenance: in journalistic work, it is crucial to be able to trace each node within the integrated graph back to the dataset from which it came. This enables *adequately sourcing* information, an important tenet of quality journalism.

R2. Little to no user effort : journalists often lack time and resources to set up IT tools or data processing pipelines. Even when they are able to use a tool supporting one or two data models (e.g., most relational databases provide some support for JSON data), handling other data models remains

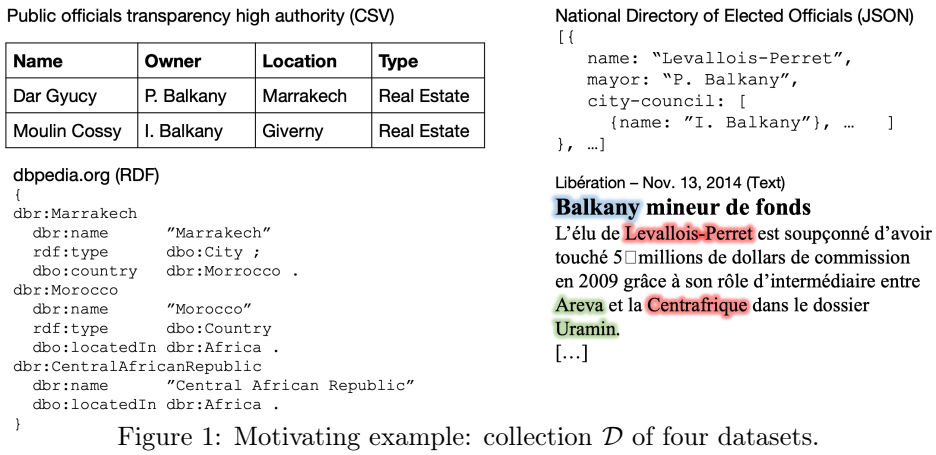


Figure 1: Motivating example: collection $\mathcal{D}$ of four datasets.

challenging. Thus, the construction of the integrated graph needs to be as automatic as possible.

C1. Little-known entities: interesting journalistic datasets feature some extremely well-known entities (e.g., highly visible National Assembly members) next to others of much smaller notoriety (e.g., the collaborators employed by the National Assembly to help organize each deputy's work; or a company in which a deputy had worked). From a journalistic perspective, such lesser-known entities may play a crucial role in making interesting connections among nodes in the graph.

C2. Controlled dataset ingestion: the confidence level in the data required for journalistic use excludes massive ingestion from uncontrolled data sources, e.g., through large-scale Web crawls.

C3. Language support: journalists are first and foremost concerned with the affairs surrounding them (at the local or national scale). This requires supporting datasets in the language(s) relevant for them - in our case, French.

R3. Performance on "off-the-shelf" hardware: Our algorithms' complexity in the data size should be low, and overall performance is a concern; the tool should run on general-purpose hardware, available to non-expert users like the ones we consider.

For what concerns querying the integrated graph, we note:

R4. Finding connections across heterogeneous datasets is a core need. In particular, our approach needs to be tolerant of inevitable differences in the organization of data across sources.

C4. Query algorithm orthogonal to answer quality scores After discussing several journalistic scenarios, no unique method (score) for de-

ciding which are the best answers to a query has been identified. Instead: (*i*) it appears that "very large" answers (say, of more than 20 edges) are of limited interest; (*ii*) connections that "state the obvious", e.g., that any two actors from a French political scenario are connected through "France", are not of interest. Therefore, our algorithm must be orthogonal, and it should be possible to use it with any score function.

To address these requirements and constraints, we have started in 2018 developing a tool called ConnectionLens, described in a short demonstration paper [10]. ConnectionLens' approach since its inception has been to (*i*) **integrate arbitrary heterogeneous datasets into a unique graph** and (*ii*) **query them with keywords**, such that an answer can span over arbitrary combinations of datasets and heterogeneous data models. However, this version of the tool [10] did not meet (R2) (little to no effort required from users), in particular, because it still followed a *mediator* approach: the integrated graph was *virtual*, thus queries were answered by evaluating subqueries over individual data sources and stitching them into a complete answer. This method did not scale on large graphs, therefore it did not meet (R3). It also suffered from other shortcomings, particularly a relatively low quality of integrated graphs from French-language data.

This paper reflects the result of the years of research and development that invested in the system.

To reach our goals under these requirements and constraints, in the following:

1. We formalize *integrated graphs* we target, and formalize the problem of constructing them from arbitrary sets of datasets. This follows the line of [10] and extends it to novel data models, such as XML and PDF.

2. We introduce an *approach*, and an *architecture* for building the graphs, leveraging data integration, information extraction, knowledge bases, and data management techniques. Steering away from [10], we adopt a centralized graph warehouse architecture, easy to install, meeting simultaneously (R2) and (R3). Within this architecture, a significant part of our effort was invested in developing resources and tools for datasets in French. English is also supported, thanks to a (much) wider availability of linguistics resources.

3. Fully novel with respect to [10], we analyze the *properties of the search space* for our keyword search problem, and propose *the first algorithm*

capable of finding matches across heterogeneous data sources while preserving each node within its original source. This algorithm vastly outperformed the previous one, which we discarded as a consequence of the change of architecture.

4. We have fully implemented our approach in an end-to-end tool; it currently supports text, CSV, JSON, XML, RDF, PDF datasets, and existing relational databases; the system has more than doubled in size and has little in common with its 2018 incarnation. Also, for the first time, we present: (*i*) a set of *use cases with real journalistic datasets*; (*ii*) an *experimental evaluation* of its scalability and the quality of the extracted graph; (*iii*) query experiments confirming the practical interest of query algorithm.

Motivating example. To illustrate our approach, we rely on a set of four datasets, shown in Figure 1. Starting from the top left, in clockwise order, we have: 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. Our goal is to *interconnect* these datasets into a graph and to be able to answer, for example, the question: "What are the connections between Levallois-Perret and Africa?" One possible answer comes by noticing that P. Balkany was the mayor of Levallois-Perret (as stated in the JSON document), and he owned the "Dar Gyucy" villa in Marrakesh (as shown in the relational table), which is in Morocco, which is in Africa (stated by the DBPedia subset). Another interesting connection in this graph is that Levallois-Perret appears in the same sentence as the Centrafrican Republic in the Libération snippet at the bottom right, which (as stated in DBPedia) is in Africa.

2. Approach and outline

We describe here the main principles of our approach, guided by the requirements and constraints stated above.

From requirement R1 (integral source preservation), it follows that *all the structure and content of each dataset is preserved* in the integrated graph, thus every detail of any dataset is mapped to some of its nodes and edges. This requirement also leads us to *preserve the provenance* of each dataset, as well as *the links that may exist within and across datasets* before loading them (e.g. interconnected HTML pages, JSON tweets replying to one another, or

RDF graphs referring to a shared resource). We term *primary nodes* the nodes created in our graph strictly based on the input dataset and their provenance; we detail their creation in Section 3.

From requirement R2 (ideally no user input), it follows that *we must identify the opportunities to automatically link (interconnect) nodes*, even when they were not interconnected in their original dataset, and even when they come from different datasets. We achieve this at several levels:

First, we leverage and extend information extraction techniques to *extract (identify) entities occurring in every node's labels in every input dataset.* For instance, "Levallois-Perret" is identified as a Location in the two datasets at right in Figure 1 (JSON and Text). Similarly, "P. Balkany", "Balkany", "I. Balkany" occurring in the relational, JSON, and Text datasets are extracted as Person entities. Our method of entity extraction, in particular for the French language, is described in Section 4.

Second, we *compare (match)* occurrences of entities extracted from the datasets to determine when they refer to the same entity and thus should be interconnected. (*i*) Some entity occurrences we encounter refer to entities such as Marrakech, Giverny etc. known in a *trusted Knowledge Base* (or KB, in short), such as DBPedia. Journalists may trust a KB for such general, non-disputed entities. We *disambiguate* each entity occurrence, i.e., try to find the URI (identifier) assigned in the KB to the entity referred to in this occurrence, and we *connect* the occurrence to the entity. Disambiguation enables, for instance, to connect an occurrence of "Hollande" to the country, and another to the former French president. A common entity found in two datasets interconnects them. We describe the module we built for entity disambiguation for the French language (language constraint C3), based on AIDA [26], in Section 5. It is of independent interest, as it can be used outside of our context. (*ii*) On little-known entities (constraint C1), disambiguation fails (no URI is found); this is the case, e.g., of "Moulin Cossy". Combined with constraint C2 (strict control on ingested sources) it leads to the lack of reliable IDs for many entities mentioned in the datasets. We strive to connect them as soon as several identical or at least *strongly similar* occurrences are found in the same or different datasets. We describe our approach for *comparing (matching)* occurrences to identify identical or similar pairs in Section 6. Section 7 describes our persistent graph storage.

For what concerns the keyword search problem, we formalize it in Section 8, showing in particular that requirement R4 leads to an explosion in the size of the search space compared to those studied in the literature. Section 9

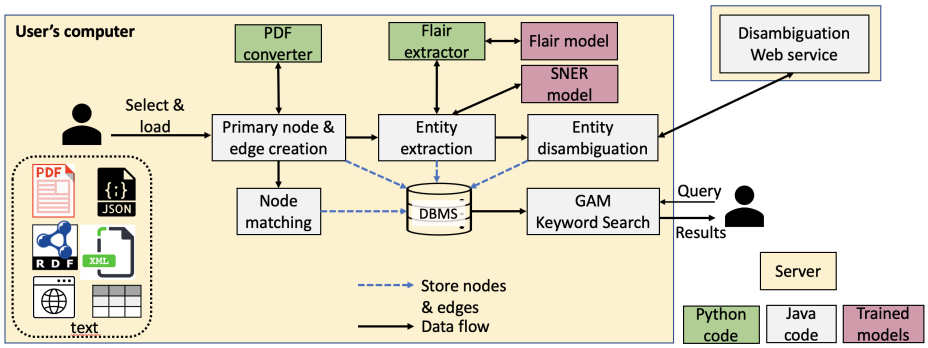


Figure 2: Outline of the ConnectionLens system architecture..

discusses some favorable cost function properties exploited in the literature and why they do not apply to our case (constraint C5). Based on this, Section 10 introduces our query algorithm.

All the above modules are implemented within the ConnectionLens systems; the connections between its modules are outlined in Figure 2 and will become clear as we present each module below. We present experiments evaluating the quality and performance of our algorithms in Section 11. Our work pertains to several areas, most notably data integration, knowledge base construction, and keyword search; we detail our positioning in Section 12.

3. Primary graph construction from heterogeneous datasets

We consider the following *data models*: relational (including SQL databases, CSV files etc.), RDF, JSON, XML, or HTML, and text. A dataset $DS = (db, prov)$ in our context is a pair whose first component is a concrete data object: a relational database, or an RDF graph, or a JSON, HTML, XML document, or a CSV, text, or PDF file. The second (optional) component *prov* is the dataset provenance; we consider here that it is the URI from which the dataset was obtained, but this could easily be generalized.

Let A be an alphabet of words. We define an **integrated graph** $G = (N, E)$ where N is the 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 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 label. We assign to each node and edge a **unique ID**, as well as a

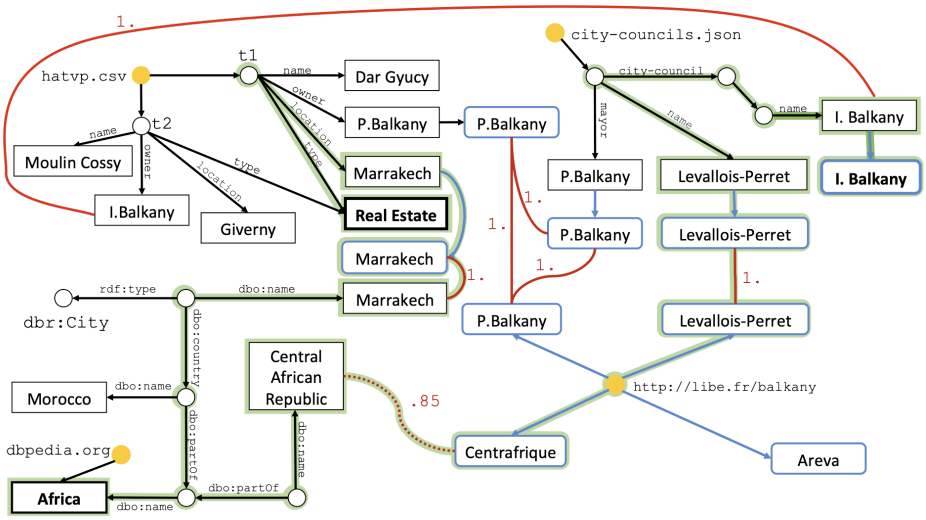


Figure 3: Integrated graph corresponding to the datasets of Figure 1. An answer to the keyword query {"I. Balkany", Africa, Estate} is highlighted in light green; the labels three keyword matches in this answer are in bold font.

type. We introduce the supported node types as needed, and write them in bold font (e.g., **dataset node**, **URI node**) when they are first mentioned; node types are important as they determine the quality and performance of matching (see Section 6). Finally, we create unique dataset IDs and associate to each node its dataset's ID.

Let $DS_i = (db_i, prov_i)$ be a dataset of any of the above models. The following two steps are taken regardless of db_i's data model: First, we introduce a **dataset node** $n_{DS_i} \in N$, which models the dataset itself (not its content). Second, if $prov_i$ is not null, we create an **URI node** $n_{prov_i} \in N$, whose value is the provenance URI $prov_i$, and an edge $n_{DS_i} \xrightarrow{\text{cl:prov}} n_{prov_i}$, where cl:prov is a special edge label denoting provenance (we do not show these edges in the Figure to avoid clutter).

Next, Section 3.1 explains how each type of dataset yields nodes and edges in G. For illustration, Figure 3 shows the integrated graph resulting from the datasets in Figure 1. In Section 3.2, we describe a set of techniques that improve the informativeness and the connectedness and decrease the size of G. Section 3.3 discusses the complexity of the graph construction.

3.1. Mapping each dataset to the graph

All the edges whose creation is described in this section reflect the *structure* of the input datasets. Thus, their confidence is always 1.0.

Relational. Let $db = R(a_1, \ldots, a_m)$ be a relation (table) (residing within an RDBMS, or ingested from a CSV file etc.) A **table node** n_R is created to represent R (yellow node with label hatvp.csv on top left in Figure 3). Let $t \in R$ be a tuple of the form $(v_1, \ldots, v_m)$ in R. A **tuple node** n_t is created for t, with an empty label, e.g., t_1 and t_2 in Figure 3. For each non-null attribute v_i in t, a **value node** n_{v_i} is created, together with an edge from n_t to n_{v_i}, labeled a_i (for example, the edge labeled owner at the top left in Figure 3). To keep the graph readable, 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. This graph modeling of relational data has been used for keyword search in relational databases [27].

RDF. The mapping from an RDF graph to our graph is the most natural. Each node in the RDF graph becomes, respectively, a URI node or a value node in G, and each RDF triple becomes an edge in E. At the bottom left in Figure 3 appear some edges resulting from our DBPedia snippet.

Text. We model a text document very simply, as a node having a sequence of children, where each child is a segment of the text (e.g., the four nodes connected by the libe.fr node at the bottom right of Figure 3). Segmentation is optional, and the default segmentation unit is a phrase, which appeared convenient for users inspecting the graph. Other segmentations (e.g., by paragraph for longer texts) could also be used, without changing the set of answers to a given query.

JSON. As customary, we view a JSON document as a tree, with nodes that are either **map nodes**, **array nodes** or value nodes. We map each node into a node of our graph and create an edge for each parent-child relation. Map and array nodes have the empty label ϵ. Attribute names within a map become edge labels in our graph. Figure 3 at the top right shows how a JSON document's nodes and edges are ingested in our graph.

XML. The ingestion of an XML document is very similar to that of JSON ones. XML nodes are either **element**, or **attribute**, or values nodes. As customary when modeling XML, value nodes are either text children of elements or values of their attributes.

HTML. An HTML document is treated very similarly to an XML one. In particular, when an HTML document contains a hyperlink of the form ..., we create a node labeled "a" and another labeled "http://a.org", and connect them through an edge labeled "href"; this is the

common treatment of element and attribute nodes. However, we detect that a child node satisfies a URI syntax, and *recognize (convert) it into a URI node*. This enables us to preserve links across HTML documents ingested together in the same graph, with the help of node comparisons (see Section 6).

PDF. A trove of useful data is found in the PDF format. To take advantage of their content, we have developed a PDF scrapper which transforms a PDF file into: (*i*) a JSON file, typically containing all the text found in the PDF document; (*ii*) if the PDF file contains bidimensional (2d) tables, each table is extracted in a separate RDF file, following an approach introduced in [9], which preserves the logical connections of each data cell with its closest. The JSON and possibly the RDFs thus obtained are ingested as explained above; moreover, from the dataset node of each such dataset d_i, we add an edge labeled cl:extractedFromPDF, whose value is the URI of the PDF file. Thus, the PDF-derived datasets are all interconnected through that URI.

3.2. Refinements and optimizations

Value node typing. The need to recognize particular types of values goes beyond identifying URIs in HTML. URIs also frequently occur in JSON (e.g., tweets), in CSV datasets etc. Thus, we examine each value to see if it follows the syntax of a URI and if so, convert the value node into a URI one, regardless of the nature of the dataset from which it comes. More generally, other categories of values can be recognized in order to make our graphs more meaningful. Currently, we similarly recognize **numeric nodes**, **date nodes**, **email address nodes** and **hashtag nodes**.

Node factorization. The graph resulting from the ingestion of a JSON, XML, or HTML document, or one relational table, is a tree; any value (leaf) node is reachable by a finite set of label paths from the dataset node. For instance, in an XML document, two value nodes labeled "Paris" may be reachable on the paths employee.address.city, while another is on the path headquartersCity. Graph creation as described in Section 3.1 creates three value nodes labeled "Paris"; we call this **per-occurrence value node creation**. Instead, **per-path** creation leads to a single node for all occurrences of "Paris" on the paths employee.address.city and employee.headquartersCity. We have also experimented with **per-dataset** value node creation, which in the above example creates a single "Paris" node, and with **per-graph**, where a single "Paris" value node is created in a graph, regardless of how many times "Paris" appears across all the datasets. Per-graph is consistent with the RDF data model, where each literal denotes one node.

Factorization turns a tree-structured dataset into a directed acyclic graph (DAG); it reduces the number of nodes and increases the graph connectivity. Factorization may introduce *erroneous connections*. For instance, constants such as *true* and *false* appear in many contexts, yet this should not lead to connecting all nodes having an attribute whose value is *true*. Named entities are another example, and they should be firstly disambiguated.

To prevent such erroneous connections, we have heuristically identified a set of **values which should not be factorized** even with *per-path*, *per-dataset* or *per-graph* value node creation. Beyond *true* and *false* and *named entities*, this currently includes *integer numeric node labels written on less than* 4 *digits*, the rationale being that small integers tend to be used for ordinals, while numbers on many digits could denote years, product codes, or other forms of identifiers. This simple heuristic could be refined. **Null codes**, or strings used to signal missing values, e.g., "N/A", "Unknown", should not be factorized, either. As is well-known from database theory, nulls should not lead to joins (or connections, in our case). Nodes with such labels will never lead to connections in the graph: they are not factorized, and they are not compared for similarity (see Section 6). This is why we currently require user input on the null codes, and assist them by showing the most frequent constants, as null codes, when they occur, tend to be more frequent than real values (as our experiments illustrate in Section 11.2.1). Devising an automated approach toward detecting null codes in the data is an interesting avenue for future work.

3.3. Complexity of the graph construction

Summing up the above processing stages, the worst case complexity of ingesting a set of datasets in a ConnectionLens graph $G = (N, E)$ is of the form:

$$c_1 \cdot |E| + c_2 \cdot |N| + c_3 \cdot |N_e| + c_4 \cdot |N|^2$$

In the above, the constant factors are explicitly present (i.e., not wrapped in an $O(\ldots)$ notation) as the differences between them are high enough to significantly impact the overall construction time (see Section 11.2.2 and 11.2.3). Specifically: c_1 reflects the (negligible) cost of creating each edge using the respective data parser, and the (much higher) cost of storing it; c_2 reflects the cost to store a node in the database and to invoke the entity extractor on its label, if it is not ϵ; N_e is the number of entity nodes found in the graph, and c_3 is the cost to disambiguate each entity; finally, the last

component reflects the worst-case complexity of node matching, which may be quadratic in the number of nodes compared. The constant c_4 reflects the cost of recording on disk that the two nodes are equivalent (one query and one update) or similar (one update).

Observe that the number of value nodes (thus N) is impacted by the node creation policy; N_e (and, as we show below, c_3) depend on the entity extractor module used.

4. Named-Entity Recognition

We enrich our graphs by leveraging Machine Learning (ML) tools for Information Extraction.

Named entities (NEs) [37] are words or phrases which, together, designate certain real-world entities. Named entities include common concepts such as people, organizations, and locations. The *Named-Entity Recognition* (NER) task consists of (*i*) identifying NEs in a natural language text, and (*ii*) classifying them according to a pre-defined set of NE types. Let n_t be a text node. We feed n_t as input to a NER module and create, for each entity occurrence E in n_t, an **entity occurrence node** (or entity node, in short) n_E; as explained below, we extract **Person, Organization** and **Location** entity nodes. Further, we add an edge from n_t to n_E whose label is cl:extract$T$, where T is the type of E, and whose confidence is c, the *confidence of the extraction*. In Figure 3, the blue, round-corner rectangles Centrafrique, Areva, P. Balkany, Levallois-Perret correspond to the entities recognized from the text document, while the Marrakech entity is extracted from the identical-label value node originating from the CSV file.

Named-Entity Recognition We describe here the NER approach we devised for our framework, for English and French. While we have used Stanford NER [17] in [10], we have subsequently developed a more performant module based on the Deep Learning Flair NLP framework [1]. Flair and similar frameworks rely on *embedding* words into vectors in a multi-dimensional space. Traditional word embeddings, e.g., Word2Vec [36], Glove [40] and fastText [5], are *static*, meaning that a word's representation does not depend on the context where it occurs. New embedding techniques are *dynamic*, in the sense that the word's representation also depends on its context. In particular, the Flair dynamic embeddings [2] achieve state-of-the-art NER performance. The latest Flair architecture [1] facilitates *combining* different

types of word embeddings, as a better performance might be achieved by combining dynamic with static word embeddings.

For English, we rely on a model[2] pre-trained using the English CoNLL-2003[3] news articles dataset. The model combines Glove embeddings [40] and so-called *forward and backward pooled* Flair embeddings that evolve across subsequent extractions. As such a model was missing for French, we trained a Flair one on WikiNER [38], a multilingual NER dataset automatically created using the text and structure of Wikipedia. The dataset contains 132K sentences, 3.4M tokens and 216K named-entities, including 74K Person, 116K Location and 25K Organization entities. The model uses stacked forward and backward French Flair embeddings with French fastText [5] embeddings.

Entity node creation. Similarly to the discussion about value node factorization (Section 3.2), we have the choice of creating an entity node n_E of type t once per occurrence, or (in hierarchical datasets) *per-path*, *per-dataset* or *per-graph*. We adopt the per graph method, with the mention that we will create one entity node for each disambiguated entity and one entity node for each non-disambiguated entity.

5. Entity disambiguation

Some (but not all) entity nodes extracted from a dataset as an entity of type T may correspond to an entity (resource) described in a trusted knowledge base (KB) such as DBPedia or Yago. When this is the case, this allows: (i) resolving *ambiguity* to make a more confident decision about the entity, e.g., whether the entity node "Hollande" refers to the former president or the country; (ii) tackling *name variations*, e.g., two Organization entities labeled "Paris Saint-Germain Football Club" and "PSG" are linked to the same KB identifier, and (iii) if this is desired, *enriching* the dataset with a certain number of facts the KB provides about the entity.

Named entity disambiguation (NED, in short, also known as entity linking) is the process of assigning a unique identifier (typically, a URI from a KB) to each named-entity present in a text. We built our NED module based on AIDA [26], part of the Ambiverse[4] framework; AIDA maps entities to resources in YAGO 3 [35] and Wikidata [47]. Our work consisted of (i) adding

[2]https://github.com/flairNLP/flair

[3]https://www.clips.uantwerpen.be/conll2003/ner/

[4]`https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/ambiverse-nlu/`

support for French (not present in Ambiverse), and (*ii*) integrating our own NER module (Section 4) within the Ambiverse framework.

For the first task, in collaboration with the maintainers of Ambiverse[5], we built a new dataset for French, containing the information required for AIDA. The dataset consists of entity URIs, information about entity popularity (derived from the frequency of entity names in link anchor texts within Wikipedia), and entity context (a set of weighted words or phrases that co-occur with the entity), among others. This information is language-dependent and was computed from the French Wikipedia.

For what concerns the second task, Ambiverse takes an input text and passes it through a text processing pipeline consisting of *tokenization* (separating words), *part-of-speech (POS) tagging*, which identifies nouns, verbs, etc., NER, and finally NED. Text and annotations are stored and processed in Ambiverse using the UIMA standard[6]. A central UIMA concept is the *Common Annotation Scheme* (or CAS); in short, it encapsulates the document analyzed, together with all the annotations concerning it, e.g., token offsets, tokens types, etc. In each Ambiverse module, the CAS object containing the document receives new annotations, which are used by the next module. For example, in the tokenization module, the CAS initially contains only the original text; after tokenization, the CAS also contains token offsets.

To integrate our Flair-based extractor (Section 4), we deployed a **new Ambiverse processing pipeline**, to which we pass as input *both the input text and the extracted entities*.

6. Node matching

This section presents our fourth and last method for **identifying and materializing connections** among nodes from the same or different datasets of the graph. Recall that (*i*) *value nodes with identical labels can be fused (factorized)* (Section 3.2); (*ii*) nodes become connected as *parents of a common extracted entity* (Section 4); (*iii*) entity nodes with different labels can be interconnected *through a common reference entity* when NED returns the same KB entity (Section 5). Other pairs of nodes with similar labels that may still need to be connected, are: *entity pairs* where disambiguation, which is

[5] `https://github.com/ambiverse-nlu/ambiverse-nlu#maintainers-and-contributors`

[6] `https://uima.apache.org/doc-uima-why.html`

context-dependent, returned no result for one or for both *(entity, value)* pairs where the value corresponds to a node from which no entity was extracted (the extraction is context-dependent and may also have some misses); and *value pairs*, such as numbers, dates, and (if desired) identical texts. Comparing texts is useful, e.g., to compare social media posts when their topics (short strings) and/or body (long string) are very similar.

When a comparison finds two nodes with *very similar labels*, we create an edge labeled **cl:sameAs**, whose confidence is the similarity between the two labels. In Figure 3, a dotted red edge (part of the subtree highlighted in green) with confidence .85 connects the "Central African Republic" RDF literal node with the "Centrafrique" Location entity extracted from the text.

When a comparison finds two nodes with *identical labels*, one could unify them, but this raises some modeling issues, e.g., when a value node from a dataset d_1 is unified with an entity encountered in another dataset d_2. Instead, we *conceptually* connect the nodes with sameAs edges whose confidence is 1.0. These edges are drawn in solid red lines in Figure 3. Nodes connected by a 1.0 sameAs edge are also termed *equivalent*. Note that conceptual equivalence edges are not stored; instead, the information about k equivalent nodes is stored using $O(k)$ space, as we explain in Section 7.

Inspired by the data cleaning literature, we start by *normalizing* node labels, e.g., person names are analyzed to identify first names, last names, civility prefixes such as "Mr.", "Dr.", and a normalized label is computed as "Firstname Lastname". Subsequently, our approach for matching is *set-at-a-time*. More precisely, we form node group pairs (Gr_1, Gr_2), and we compare pairs of labels using the *similarity function* known to give the best results (in terms of matching quality) for those groups. The Jaro measure [28] gives good results for *short strings* [14] and is applied to compute the similarity between pairs of entities of the same type recognized by the entity extractor (i.e., person names, locations, and organization names which are typically described by short strings).

7. Graph storage

The DBMS used to store our graphs (Figure 2) is PostgreSQL, accessed through JDBC. This is a mature, efficient and free tool, running on a variety

of platforms, thus meeting our requirement R3 from the Introduction[7].

The table **Nodes(id, label, type, datasource, label, normaLabel, representative)** stores the basic attributes of a node, the ID of its data source, its normalized label, and its representative's ID. For nodes not equivalent to any other, the representative is the ID of the node itself. As explained previously, the *representative* attribute allows encoding information about equivalent nodes. Table **Edges(id, source, target, label, datasource, confidence)** stores the graph edges derived from the data sources, as well as the extraction edges connecting entity nodes with the parent(s) from which they have been extracted. Finally, the **Similar(source, target, similarity)** table stores a tuple for each similarity comparison whose result is above the threshold but less than 1.0. The pairs of nodes to be compared for similarity are retrieved by means of SQL queries.

This relational store is encapsulated behind a Graph interface, which can be easily implemented differently to take advantage of other storage engines. When creating a graph, storage is far from being the bottleneck (as our experiments in Section 11.2.1 show); on the contrary, when querying the graph (see below), the relational store incurs a relatively high access cost for each edge. We currently mitigate this problem using a memory cache for nodes and edges within the ConnectionLens application.

8. Querying the graph

We formalize the keyword search problem over a graph built out of heterogeneous datasets as previously described.

8.1. Search problem

We consider a graph $G = (N, E)$ and we denote by $\mathcal{L}$ the set of all the labels of G nodes, plus the empty label ϵ (see Figure 3). Let W be the set of *keywords*, obtained by stemming the label set $\mathcal{L}$; 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 set t of G edges which (i) together, form a tree (each node is reachable from any other through exactly one path), (ii) for each w_i, contain at least one node whose label matches w_i. Here, the edges are **considered**

[7]We had also experimented with Neo4J [42] as a DBMS, specifically with shortest-path search, but found no performance advantage for the operations ConnectionLens requires.

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 a node $n_i \in t$ such that $w_i \in \lambda(n_i)$.

We treat the edges of G as undirected when defining the AT to allow more query results on a graph built out of heterogeneous content whose structure is not well-known to users. Further, we are interested in **minimal** answer trees, that is: (i) removing an edge from the tree should make it lack one or more of the query keywords w_i; (ii) if a query keyword w_i matches the label of more than one node in the answer tree, then all these matching nodes must be equivalent.

Condition (ii) is specific to the graph we consider, built from *several data sources connected by equivalence or similarity edges*. In classical graph keyword search problems, each query keyword is matched *exactly once* in an answer (otherwise, the tree is considered non-minimal). In contrast, our answer trees *may need to traverse equivalence edges*, and if w_i is matched by one node connected by such an edge, it is also matched by the other. For instance, consider the three-keyword query "Gyucy Balkany Levallois" in Figure 3: the keyword Balkany is matched by the two nodes labeled "P. Balkany" which are part of the answer.

As a counter-example, consider the query "Balkany Centrafrique" in Figure 3, assuming the keyword Centrafrique is also matched in the label "Central African Republic"[8]. Consider the tree that connects a "P. Balkany" node with "Centrafrique", and also traverses the edge between "Centrafrique" and "Central African Republic": this tree is not minimal, thus it is not an answer. The intuition for rejecting it is that "Centrafrique" and "Central African Republic" are not necessarily equivalent (we have a similarity, not an equivalence edge). Therefore the query keyword "Centrafrique" is matched by two potentially different things in this answer, making it hard to interpret.

A direct consequence of minimality is that *in an answer, each and every leaf matches a query keyword*. A graph may hold several minimal answer trees for a given query. We consider available a *scoring function* which assigns a higher value to more interesting answer trees (see Section 9).

Problem Statement. Given the graph G built out of the datasets $\mathcal{D}$ 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

[8]This may be the case using a more advanced indexing system that includes some natural language understanding, term dictionaries, etc.

traverse G edges in any direction; this makes the problem challenging.

8.2. Search space and complexity

The problem that we study is related to the (Group) Steiner Tree Problem, which we recall below.

Given a graph G with weights (costs) on edges, and a set of m nodes $n_1, \ldots, n_m$, the *Steiner Tree Problem (STP)* [19] 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, or that considered in similar works**, discussed in Section 12. This 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) [19] is: given m *sets of nodes* from G, find the minimum-cost subtree connecting one node from each of these sets. GSTP does not raise the problem ($\circ$), but still has all the others.

In conclusion, the complexity of the problem we consider is extremely high. Therefore, fully solving it is unfeasible for large and/or high-connectivity graphs. Instead, our approach is: (*i*) *Attempt to find all answers from the smallest* (fewest edges) *to the largest.* Enumerating small trees first is both a practical decision (we use them to build larger ones) and fits the intuition that we should not miss small answers that a human could have found manually. However, as we will explain, we still "opportunistically" build some trees before exhausting the enumeration of smaller ones, whenever this is likely to lead faster to answers. The strategy for choosing to move towards bigger instead of smaller tress leaves room for optimizations on the search order. (*ii*) *Stop at a given time-out or when m answers have been found*, for some $m \geq k$; (*iii*) *Return the k top-scoring answers* found.

9. Scoring answer trees

We now discuss how to evaluate the quality of an answer. Section 9.1 introduces the general notion of score on which we base our approach. Sec-

tion 9.2 describes the metric that we attach to edges to instantiate this score, and Section 9.3 details the actual score function we used.

9.1. Generic score function

We have configured our problem setting to allow *any scoring function*, which enables the use of different scoring schemes fitting different users' requirements. As a consequence, this approach allows us to study the interaction of the scoring function with different properties of the graph.

Given an answer tree t to a query Q, we consider a score function consisting of (at least) the following two components. First, the *matching score* $ms(t)$, which reflects the quality of the answer tree, that is, how well its leaves match the query terms. Second, the *connection score* $cs(t)$, which reflects the quality of the tree connecting the edges. Any formula can be used here, considering the number of edges, the confidence or any other property attached to edges, or a query-independent property of the nodes, such as their PageRank or betweenness centrality score, etc.

The score of t for Q, denoted $s(t)$, is computed as a combination of the two independent components $ms(t)$ and $cs(t)$. Popular combination functions (a weighted sum, or product, etc.) are monotonous in both components, however, our framework does not require monotonicity. Finally, both $ms(t)$ and $cs(t)$ can be tuned based on a given user's preferences, to personalize the score or make them evolve in time through user feedback etc.

9.2. Edge specificity

We now describe a metric on edges, which we used (through the connection score $cs(t)$) to favor edges that are "rare" for both nodes they connect. This metric was inspired by our experiments with real-world data sources and it helped return interesting answer trees in our experience.

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 exiting n.
The **specificity** of an edge $e = n_1 \xrightarrow{l} n_2$ is defined as:

$$s(e) = 2/(N^l_{n_1\rightarrow} + N^l_{\rightarrow n_2}).$$

Specificity 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 Levallois-Perret comprises only one mayor (and one individual cannot be mayor of two cities in

France). 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 $2/(1 + 54) \simeq .036$.

9.3. Concrete score function

We have implemented the following prototype scoring function in our system. For an answer t to the query Q, we compute the matching score $ms(t)$ as the *average*, over all query keywords w_i, of the similarity between the t node matching w_i and the keyword w_i itself; we used the edit distance.

We compute the connection score $cs(t)$ based on edge confidence, on the one hand, and edge specificity on the other. We *multiply* the confidence values, since we consider that uncertainty (confidence < 1) multiplies; and we also *multiply* the specificities of all edges in t, to discourage many low-specificity edges. Specifically, our score is computed as:

$$score(t, Q) = \alpha \cdot ms(t, Q) + \beta \cdot \prod_{e \in E} c(e) + (1 - \alpha - \beta) \cdot \prod_{e \in E} s(e)$$

where α, β are parameters of the system such that $0 \leq \alpha, \beta < 1$ and $\alpha + \beta \leq 1$.

10. Answering keyword queries

We now present our approach for computing query answers on the graph, which integrates the heterogeneous datasets.

10.1. Grow *and* Merge

Our algorithm relies on concepts from prior literature [13, 25] while exploring many 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 one-node partial tree. For instance, in Figure 3, one-node trees are built from the nodes with boldface text, labeled "Africa", "Real Estate" and "I. Balkany". We identify two transformations that can be applied to form increasingly larger trees, working toward query answers:

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 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.

In a *single-dataset* context, GROW and MERGE have the following properties. (gm_1) GROW alone is **complete** (guaranteed to find all answers) for $k = 1, 2$ only; for higher k, GROW and MERGE together are complete. (gm_2) Using MERGE steps helps to find answers faster than using just GROW [25]: partial trees, each starting from a leaf that matches a keyword, are merged into an answer as soon as they have reached the same root. (gm_3) An answer can be found through **multiple combinations of GROW and MERGE**. For instance, consider a linear graph $n_1 \rightarrow n_2 \rightarrow \ldots n_p$ and the two-keyword query $\{a_1, a_p\}$ where a_i matches the label of n_i. The answer is obviously the full graph. It can be found: starting from n_1 and applying $p - 1$ GROW steps; starting from n_p and applying $p - 1$ GROW steps; and in $p - 2$ ways of the form MERGE(GROW(GROW...), GROW(GROW...)), each merging in an intermediary node $n_2, \ldots, n_{p-1}$. These are all the same according to our definition of an answer (Section 8.1), which does not distinguish a root in an answer tree; this follows users' need to know how things are connected, and for which the tree root is irrelevant.

10.2. Adapting to multi-datasets graphs

The changes we brought for our harder problem (bidirectional edges and multiple interconnected datasets) are as follows.

1. Bidirectional growth. We allow GROW to traverse an edge both going from the source to the target, and going from the target to the source.

2. Many-dataset answers. As defined in a single-dataset scenario, GROW and MERGE do not allow to connect multiple datasets. To make that possible, we need to enable one, another, or both to also traverse similarity and equivalence edges (shown in solid or dotted red lines in Figure 3). We decide to simply extend GROW to allow it to traverse not just data edges, but also *similarity* edges between nodes of the same or different datasets. We handle *equivalence* edges as follows:

GROW-to-representative Let t be a partial tree developed during the search, rooted in a node n, such that the representative of n is a node

$n_{rep} \neq n$. GROW2REP creates a new tree by adding to t the edge $n \overset{\equiv}{\rightarrow} n_{rep}$; this new tree is rooted in n_{rep}. If n is part of a group of p equivalent nodes, only *one* GROW2REP step is possible from t, to the unique representative of n; GROW2REP does not apply again on GROW2REP(t), because the root of this tree is n_{rep}, which is its own representative.

Together, GROW, GROW2REP and MERGE enable finding answers that span multiple data sources, as follows: GROW allows exploring data edges within a dataset and similarity edges within or across datasets. GROW2REP goes from a node to its representative when they differ; the representative may be in a different dataset. MERGE merges trees with the same root: when that root represents p equivalent nodes, this allows connecting partial trees, including GROW2REP results, containing nodes from different datasets. Thus, MERGE can build trees spanning multiple datasets.

One potential performance problem remains. Consider again p equivalent nodes $n_1, \ldots, n_p$; assume without loss of generality that their representative is n_1. Assume that during the search, a tree t_i is created rooted in each of these p nodes. GROW2REP applies to all but the first of these trees, creating the trees $t'_2, t'_3, \ldots, t'_p$, all rooted in n_1. Now, MERGE can merge any pair of them, and can then repeatedly apply to merge three, then four such trees, etc., as they all have the same root n_1. The exponential explosion of GROW trees, avoided by introducing GROW2REP, is still present due to MERGE.

We solve this problem as follows. Observe that in an answer, a *path of two or more equivalence edges* of the form $n_1 \overset{\equiv}{\rightarrow} n_2 \overset{\equiv}{\rightarrow} n_3$ such that *a node internal to the path*, e.g. n_2, *has no other adjacent edge*, even if allowed by our definition, is *redundant*. Intuitively, such a node brings nothing to the answer since its neighbors, e.g., n_1 and n_3, could have been connected directly by a single equivalence edge, thanks to the transitivity of equivalence. We call *non-redundant* an answer that does not feature any such path, and decide to **search for non-redundant answers** only.

The following properties hold on non-redundant answers:

Property 1. *There exists a graph G and a k-keyword query Q such that a non-redundant answer contains $k-1$ adjacent equivalence edges (edges that, together, form a single connected subtree).*

We prove this by exhibiting such an instance. Let G be a graph of $2k$ nodes shown in Figure 4 (a), such that all the x_i are equivalent, and consider the k-keyword query $Q = \{a_1, \ldots, a_k\}$ (each keyword matches exactly the

respective a_i node). An answer needs to traverse all the k edges from a_i to x_i, and then connect the nodes $x_i, \ldots, x_k$; we need $k-1$ equivalence edges for this.

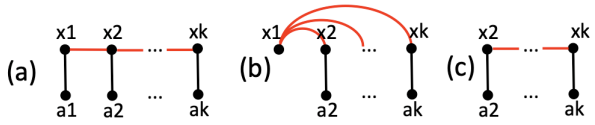


Figure 4: Sample graph and answer trees.

Next, we show:

Property 2. *Let t be a non-redundant answer to a query Q of k keywords. A group of adjacent equivalence edges contained in t has at most $k-1$ edges.*

We prove this by induction over k. For $k = 1$, each answer has 1 node and 0 edge (trivial case).

Now, consider this true for k and let us prove it for $k+1$. Assume by contradiction that a non-redundant answer t_Q to a query Q of $k+1$ keywords comprises $k+1$ adjacent equivalence edges. Let Q' be the query having only the first k keywords of Q, and t' be a subtree of t that is a non-redundant answer to Q':

- t' exists, because t connects all Q keywords, thus also the Q' keywords;
- t' is non-redundant, because all its edges are in the (non-redundant) t.

By the induction hypothesis, t' has at most $k-1$ adjacent equivalence edges. This means that there are *two adjacent equivalent edges* in $t \setminus t'$.

1. If these edges, together, lead to two distinct leaves of t, then t has *two* leaves not in t'. This is not possible, because by definition of an answer, t has $k+1$ leaves (each matching a keyword) and similarly t' has k leaves.
2. It follows, then, that the two edges lead to a single leaf of t, therefore the edges form a redundant path. This contradicts the non-redundancy of t, and concludes our proof.

Property 2 gives us an important way to control the exponential development of trees due to p equivalent nodes. GROW, GROW2REP and MERGE, together, can generate trees with up to k (instead of $k-1$) adjacent equivalence edges. This happens because GROW2REP may "force" the search to visit the representative of a set of k equivalent nodes (see Figure 4(b), assuming x_1

is the representative of all the equivalent x_is, and the query $\{a_2, \ldots, a_k\}$). The resulting answer may be redundant if the representative has no other adjacent edges in the answer other than equivalence edges. In such cases, in a **post-processing step**, we remove from the answer the representative and its equivalence edges, then reconnect the respective equivalent nodes using $k-1$ equivalence edges. This guarantees to obtain a non-redundant tree, such as the one in Figure 4(c).

10.3. The GAM algorithm

We now have the basic exploration steps we need: GROW, GROW2REP and MERGE. In this section, we explain how we use them in our integrated keyword search algorithm.

We decide to apply in sequence: one GROW or GROW2REP (see below), leading to a new tree t, immediately followed by all the MERGE operations possible on t. Thus, we call our algorithm **Grow and Aggressive Merge** (GAM, in short). We merge aggressively to detect as quickly as possible when some of our trees, merged at the root, form an answer.

Given that every node of a currently explored answer tree can be connected with several edges, we need to decide which GROW (or GROW2REP) to apply at a certain point. For that, we use a **priority queue** U in which we add (tree, edge) entries: for GROW, with the notation above, we add the (t, e) pair, while for GROW2REP, we add t together with the equivalence edge leading to the representative of t's root. In both cases, when a (t, e) pair is extracted from U, 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 read from the graph the (data, similarity, or equivalence) edges adjacent to t_G's root and add to U more (tree, edge) pairs to be considered further during the search. The algorithm then picks the highest-priority pair in U and reiterates; it stops when U is empty, at a timeout, or when a maximum number of answers are found (whichever comes first).

The last parameter impacting the exploration order is the priority used in U: at any point, U gives the highest-priority (t, e) pair, which determines the operations performed next.

1. Trees matching *many query keywords* are preferable, to go toward complete query answers;

Procedure **process**(tree t)

- if t is not already in E
- then
  - add t to E
  - if t has matches for all the query keywords
  - then post-process t if needed; output the result as an answer
  - else insert t into K

Algorithm **GAMSearch**(query $Q = \{w_1, w_2, \ldots, w_k\}$)

1. For each w_i, $1 \leq i \leq k$
   - For each node n_i^j matching w_i, let t_i^j be the 1-node tree consisting of n_i^j; process(t_i^j)
2. Initial MERGE*: try to merge every pair of trees from E, and process any resulting answer tree.
3. Initialize U (empty so far):
   (a) Create GROW opportunities: Insert into U the pair (t, e), for each $t \in E$ and e a data or similarity edge adjacent to t's root.
   (b) Create GROW2REP opportunities: Insert into U the pair $(t, n \rightarrow n_{rep})$ for each $t \in E$ whose root is n, such that the representative of n is $n_{rep} \neq n$.
4. While (U is not empty)
   (a) Pop out of U the highest-priority pair (t, e).
   (b) Apply the corresponding GROW or GROW2REP, resulting in a new tree t''; process(t'').
   (c) If t'' was not already in E, agressively MERGE:
      i. Let NT be a set of new trees obtained from the MERGE (initially $\emptyset$).
      ii. Let $\mathbf{p_1}$ be the keyword set of t''
      iii. For each keyword subset $\mathbf{p_2}$ that is a key within K, and such that $\mathbf{p_1} \cap \mathbf{p_2} = \emptyset$
         A. For each tree t^i that corresponds to $\mathbf{p_2}$, try to merge t'' with t^i. Process any possible result; if it is new (not in E previously), add it to NT.
   (d) Re-plenish U (add more entries in it) as in step 3, based on the trees $\{t''\} \cup NT$.

Figure 5: Outline of GAM algorithm

2. At the same number of matched keywords, *smaller trees* are preferable in order not to miss small answers;

3. 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*.

Algorithm details Beyond the priority queue U described above, the algorithm also uses a *memory of all the trees explored*, called E. It also organizes all the (non-answer) trees into a map K in which they can be accessed by the subset of query keywords that they match. The algorithm is shown in pseudocode in Figure 5, following the notations introduced in the above discussion.

While not shown in Figure 5 to avoid clutter, the algorithm *only develops minimal trees* (thus, it only finds minimal answers). This is guaranteed:

- When creating Grow and Grow2Rep opportunities (steps 3 and 4d): we check not only that the newly added does not close a cycle, but also that the matches present in the new tree satisfy our minimality condition (Section 8.1).

- Similarly, when selecting potential Merge candidates (step 4(c)iiiA).

11. Experimental evaluation

The algorithms described above are implemented in the **ConnectionLens** prototype, available online. Below, we report the results of an experimental evaluation we carried out to study the performance of its algorithms, as well as quality aspects of the constructed graphs. Section 11.1 describes the software and hardware setup, and our datasets. Section 11.2 focuses on graph construction, while Section 11.3 targets keyword query answering.

11.1. Software, hardware, and datasets

ConnectionLens is a **Java application** (44.700 lines) that relies on a relational database to store the constructed graphs and as a back-end used by the query algorithm. It features controllable-size caches to keep in memory as many nodes and edges as possible; this allows adapting to different memory sizes. It also comprises **Python** code (6.300 lines) which implements entity extraction (Section 4) and content extraction from PDF documents to JSON (see [8]), tasks for which the most suitable libraries are in Python.

The Flair extractor (Section 4) and the disambiguator (Section 5) are Web services which ConnectionLens calls. The former is deployed on the machine where ConnectionLens runs. We deployed the latter on a dedicated Inria server, adapting the original Ambiverse code to our new pipeline introduced in Section 5; the disambiguator consists of 842 Java classes.

For our experiments, we used a **regular server** from 2016, equipped with 2x10-core Intel Xeon E5-2640 (Broadwell) CPUs clocked at 2.40GHz, and 128GB DRAM, which uses PostgreSQL 12.4 to store the graph content in a set of tables. This is a medium-capacity machine without special capabilities; recall our requirement **R3** that our algorithms be feasible on off-the-shelf hardware. We also used a **GPU server** from 2020, with a 2x16-core Intel Xeon Gold 5218 (Skylake) CPUs clocked at 2.30GHz, an NVIDIA Tesla V100 GPU and 128GB DRAM. To show our software's applicability to standard hardware configurations, we focus on the results that we obtained with our regular server. However, we also include some results on the more advanced server to show that our platform adapts seamlessly to modern as well as heterogeneous hardware, which includes both CPUs and GPUs. When needed to separate them, we will refer to each server with its CPU generation name.
Data sources Most of our evaluation is on *real-world* datasets, described below from the smallest to the largest (measuring their size on disk before being input to ConnectionLens). **1.** We crawled the French online newspaper Mediapart and obtained 256 articles for the search keywords "*corona, chloroquine, covid*" (256 documents), and 886 for "economie, chomage, crise, budget" (1142 documents and **18.4 MB** overall). **2.** An **XML document**[9] comprising business interest statements of French public figures, provided by HATVP (*Haute Autorité pour la Transparence de la Vie Publique*); the file occupies **35 MB**. **3.** A subset of the **YAGO 4** [46] RDF knowledge base, comprising entities present in the French Wikipedia and their properties; this takes **2.49 GB** on disk (17.36 M triples). For a fine-granularity, controlled study of our query algorithm, we also devised a set of *small synthetic graphs*, described in Section 11.3.

11.2. Construction evaluation

We present the results of our experiments measuring the performance and quality of the graph construction modules. We study graph construction

[9] https://www.hatvp.fr/livraison/merge/declarations.xml

performance in Section 11.2.1, the quality of our information extraction in Section 11.2.2, and that of disambiguation in Section 11.2.3.

11.2.1. Graph construction

We start by studying the impact of **node factorization** (Section 3.2) on the number of graph nodes and the graph storage time. For that, we rely on the XML dataset, and *disable entity extraction, entity disambiguation, and node matching*. Its (unchanged) number of edges $|E|$ is 1.588.839. For each type of loading, we report the number of nodes $|N|$, the time spent storing nodes and edges to disk T_{DB}, and the total running time T in Table 6.

Value node creation policy	$\|N\|$	T_{DB} (s)	T (s)
Per-instance	1.588.840	318	319
Per-path	1.093.060	271	318
Per-path w/ null code detection	1.207.951	276	323
Per-dataset	1.084.918	265	307
Per-graph	1.084.918	179	228
Per-graph w/ null code detection	1.199.966	179	229

Figure 6: Impact of node factorization.

Moving from per-instance to per-path node creation reduces the number of nodes by a third. However, this introduces some errors, as the dataset features many **null codes** (Section 3.2); for instance, with per-instance value creation, there are 1.113 nodes labeled *"néant"* (meaning "non-existent"), 32.607 nodes labeled *"Données non publiées"* (unavailable information) etc. Using per-path, the latter are reduced to just 1.154, which means that in the dataset, *"Données non publiées"* appears 32.607 times on 1.154 different paths. However, this factorization, which introduces connections between the XML nodes which are parents of this "null" value, may be seen as wrong. When the null codes were input to ConnectionLens, such nodes are no longer unified; the number of nodes increases, and so does the storage time. In this graph, consisting of one data source, per-dataset and per-graph produce the same number of nodes, overall the smallest; it also increases when null codes are not factorized. We conclude that *per-graph value creation combined with null code detection* is a practical alternative.

Next, we study the impact of **named entity extraction** (Section 4) and **disambiguation** (Section 5) on the graph construction time.

For this, we load 100.000 triples from our Yago subset, with per-graph factorization, natural for the RDF data model where each literal or URI denotes one node in the RDF graph. In Figure 7, we load the triples using several configurations: without any entity extraction (NONE); with SNER

entity extraction, without and then with disambiguation; with FLAIR entity extraction, without and then with disambiguation. While Flair is slower, its results are qualitatively much better than those obtained with SNER (see Section 11.2.2 below). To make it faster, we also implement a **batch extraction** mechanism whereas l_B labels are input a time to each extraction service, to take advantage of the parallel processing capacity available in current CPUs. In Figure 7, in the "FLAIR, BATCH" column, we used $l_B = 128$ which maximized performance in our experiments. A second optimization leverages the fact that so-called *sequence to sequence (seq2seq)* models such as that used in our Flair extractor, when given a batch of inputs, pad the shortest ones to align them to the longest input, and some computation effort is lost on useless padding tokens. Instead, if *several batches*, say n_B, are received by the extraction service, it can *re-group* the inputs so that one call to the seq2seq model is made over inputs of very similar length, thus no effort is wasted. In our experiments, we used $n_B = 10$.

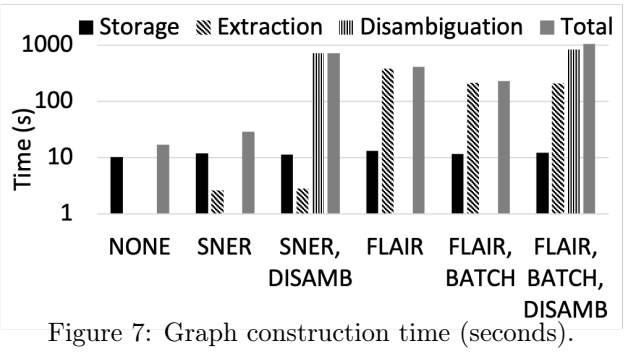


Figure 7: Graph construction time (seconds).

Figure 7 shows the time taken by storage, extraction and disambiguation (when applied) and the total graph creation time; note the logarithmic y axis. Storing the graph PostgreSQL dominates the loading time in the absence of extraction, or when using SNER. In contrast, Flair extraction takes more than one order of magnitude longer; batching reduces it by a factor of two. Finally, disambiguation, relying on computationally complex operations, takes longest; it also incurs a modest invocation overhead as it resides on a different server (with the regular server hardware described in Section 11.1), in the same fast network.

Next, we study **node matching** (Section 6). For this, we loaded the XML dataset, which comes from individual declaration of interest, filled in by hundreds of users, with numerous spelling variations, small errors and typos. Loaded per-graph mode, with batched Flair extraction, the resulting graph has 1.102.209 nodes and 1.615.291 edges. We test two configurations: comparing *all leaf node pairs* to find possible similarity edges, respectively, comparing *only entity pairs*. On the regular server, in both cases, data stor-

age took 3 minutes, and extraction 13 minutes. When all comparisons are made, they take 39 minutes, dominating the total time of 56 minutes. A total of 28.875 similar pairs are found to be above our similarity thresholds, and lead to the same number of cl:sameAs edges stored in the graph, together with the respective similarity values. Only 748 among these are entity pairs; the others are pairs of similar strings. This confirms the interest of node matching; we hope to reduce its cost further by using techniques such as Locality-Sensitive Hashing (LSH).

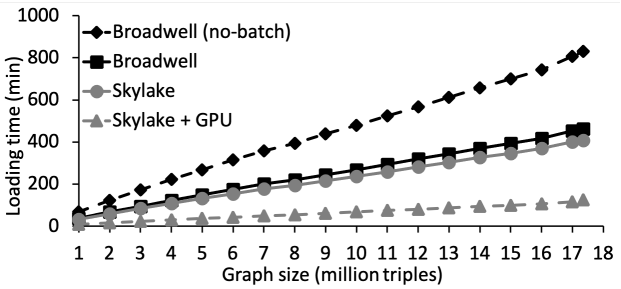


Figure 8: YAGO loading time (minutes) using Flair.

To study the **scalability** of our loading process, we loaded our YAGO subset by slices of 1M triples and measured the running time for these increasing data sizes, using the best extractor (Flair) with the same batch size(s) as above. Figure 8 shows the loading time as the data grows, in three different hardware settings: on our regular server, our more powerful GPU server *disabling GPU use*, and the same *exploiting the GPU*. Figure 8 shows that loading time scales linearly in the data volume on all configurations, and batching helps make the most out of our regular server.

Complete graph Finally, we loaded all the data sources described in Section 11.1 in a single graph, using per-graph node creation, batched Flair extraction, and disambiguation for HTML and XML (not for the RDF Yago subset, whose literals are already associated to URIs). The graph has $|N| =$ 8.019.651 nodes (including 677.459 person entities, 275.316 location entities, and 61.452 organization entities), and $|E| = 20.642.207$ edges. Many entities occur across data sources, e.g., 330 person entities occur, each, in at least 10 sources; the French president E. Macron occurs in 183 distinct sources. All these lead to interconnections across sources. On our fastest (GPU) hardware configuration, loading the RDF data took 128 minutes; the HTML articles another 260, and the XML document 96 minutes more. The last two times reflect the relatively high cost of disambiguation (recall Figure 7). ConnectionLens users can turn it off when it is not needed (e.g., if users feel they know the real-world entities behind entity labels encountered in the graph), or trigger it *selectively*, e.g., on organizations but not on people nor locations etc.

11.2.2. Named-Entity Recognition quality

Due to the unavailability of an off-the-shelf, good-quality entity extractor for French text, we decided to train a new model. To decide the best NLP framework to use, we experimented with the Flair [1] and SpaCy (`https://spacy.io/`) frameworks. Flair allows *combining* several embeddings, which can lead to significant quality gains. Following [1], after testing different word embedding configurations, we trained a Flair model using *stacked forward and backward French Flair embeddings* with *French fastText embeddings* on the WikiNER dataset. We will refer to this model as *Flair-SFTF*.

Below, we describe a *qualitative comparison* of *Flair-SFTF* with the French Flair and SpaCy *pre-trained* models. The French pre-trained Flair model is trained with the WikiNER dataset, and uses French character embeddings trained on Wikipedia, and French fastText embeddings. As for SpaCy, two pre-trained models are available for French: a medium (*SpaCy-md*) and a small one (*SpaCy-sm*). They are both trained with the WikiNER dataset and the same parameterization. The difference is that *SpaCy-sm* does not include word vectors, thus, in general, *SpaCy-md* is expected to perform better, since word vectors will most likely impact the model performance positively. Our evaluation also includes the model previously present in ConnectionLens [10], trained using *Stanford NER* [17], with the Quaero Old Press Extended Named Entity corpus [18].

We measured the precision, recall, and $F1$-score of each model using the *conlleval* evaluation script, previously used for such tasks[10]. *conlleval* evaluates *exact matches*, i.e., both the text segment of the proposed entity and its type, need to match "gold standard" annotation, to be considered correct. Precision, recall, and $F1$-score (harmonic mean of precision and recall) are computed for each named-entity type. To get an aggregated, single quality measure, *conlleval* computes the *micro-average* precision, recall, and $F1$-score over all recognized entity instances, of all named-entity types.

For evaluation, we used the entire FTBNER dataset [41]. We pre-processed it to convert its entities from the seven types they used, to the three we consider, namely, persons, locations and organizations. After pre-processing, the dataset contains 12K sentences and 11K named-entities (2K persons, 3K locations and 5K organizations).

[10]The script `https://www.clips.uantwerpen.be/conll2002/ner/` has been made available in conjunction with the CoNLL (Conference on Natural Language Learning).

Entities	Flair-SFTF	Flair-pre-trained	SpaCy-md	SpaCy-sm	Stanford NER
LOC-P	59.52%	53.26%	55.77%	54.92%	62.17%
LOC-R	79.36%	77.71%	78.00%	79.41%	69.05%
LOC-$F1$	68.02%	63.20%	65.04%	64.93%	65.43%
ORG-P	76.56%	74.57%	72.72%	71.92%	15.82%
ORG-R	74.55%	75.61%	54.85%	53.23%	5.39%
ORG-$F1$	75.54%	75.09%	62.53%	61.18%	8.04%
PER-P	72.29%	71.76%	53.09%	57.32%	55.31%
PER-R	84.94%	84.89%	74.98%	79.19%	88.26%
PER-$F1$	78.10%	77.78%	62.16%	66.50%	68.00%
Micro-P	69.20%	65.55%	61.06%	61.25%	50.12%
Micro-R	77.94%	77.92%	65.93%	66.32%	40.69%
Micro-$F1$	73.31%	71.20%	63.40%	63.68%	44.91%

Table 1: Quality of NER from French text.

The evaluation results are shown in Table 1. All models perform better overall than the *Stanford NER* model previously used in ConnectionLens [10], which has a micro $F1$-score of about 45%. The *SpaCy-sm* model has a slightly better overall performance than *SpaCy-md*, with a small micro $F1$-score difference of 0.28%. *SpaCy-md* shows higher $F1$-scores for locations and organizations, but is worse on people, driving down its overall quality. All Flair models surpass the micro scores of SpaCy models. In particular, for people and organizations, Flair models show more than 11% higher $F1$-scores than SpaCy models. Flair models score better on all named-entity types, except for locations when comparing the SpaCy models, specifically, with the *Flair-pre-trained*. *Flair-SFTF* has an overall $F1$-score of 73.31% and has better scores than the *Flair-pre-trained* for all metrics and named-entity types, with the exception of the recall of organizations, lower by 1.06%. In conclusion, *Flair-SFTF* is the best NER model we evaluated.

11.2.3. Disambiguation quality

We now evaluate the quality of the disambiguation module. As mentioned in Section 5, our module works for both English and French.

The performance for English has been measured on the CoNLL-YAGO dataset [26], by the developers of Ambiverse. They report a micro-accuracy of 84.61% and a macro-accuracy of 82.67%. To the best of our knowledge, there is no labeled corpus for entity disambiguation in French. Thus we

evaluate the module's performance on the FTBNER dataset previously introduced. FTBNER consists of sentences annotated with named entities. The disambiguation module takes a sentence, the type, and offsets of the entities extracted from it, and returns for each entity either the URI of the matched entity or an empty result if the entity was not found in the KB. In our experiment, 19% of entities have not been disambiguated, more precisely 22% of organizations, 29% of persons, and 2% of locations. For a fine-grained error analysis, we sampled 150 sentences, and we manually verified the disambiguation results (Figure 9). The module performs very well, with excellent results for locations ($F1 = 98.01\%$), followed by good results for organizations ($F1 = 82.90\%$) and for persons ($F1 = 76.62\%$). In addition to these results, we obtain a micro-accuracy of 90.62% and a macro-accuracy of 90.92%. The performance is comparable with the one reported by the Ambiverse authors for English. We should note that the improvement for French might be due to our smaller test set.

11.3. Answering keyword queries

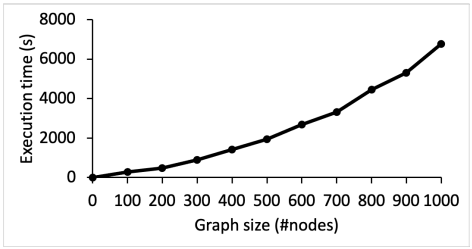
	Precision	Recall	$F1$
LOC	99.00%	97.05%	98.01%
ORG	92.38%	75.19%	82.90%
PER	75.36%	77.94%	76.62%
Micro	90.51%	82.94%	86.55%

Figure 9: Quality of disambiguation for French.

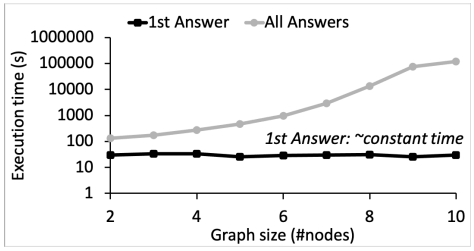
This section presents the results that we obtained by using synthetic and real-world datasets. In each case, we first describe the datasets, and then we present and explain our findings. We bound the query execution time to 120 seconds, after which the algorithm stops searching for matches.

11.3.1. Queries on synthetic datasets

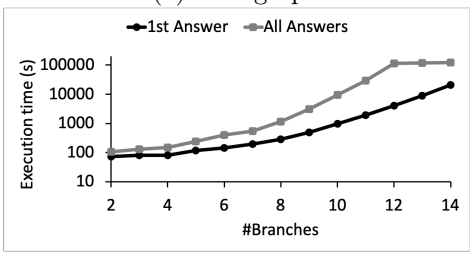
We first study the performance of our algorithm on different types of synthetic datasets. The first type is the *line graph*, where every node is connected with two others, having one edge for each node, except two nodes connected with only one. We use the line graph to clearly show the performance of GROW and MERGE operations with respect to the graph size. The second type is the *chain graph*, which is the same as the line, but it has two edges (instead of one) connecting every pair of nodes. We use the chain graph to show the algorithm's performance as we double the number of edges of the line graph, and we give more options to GROW and MERGE. The third type is the *star graph*, where we have several line graphs connected



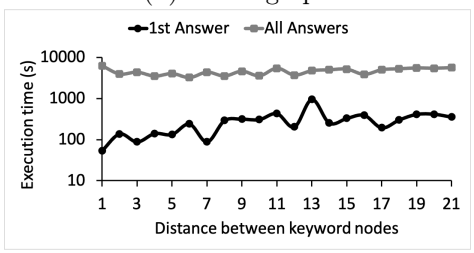
(a) Line graph



(b) Chain graph



(c) Star graph



(d) Barabasi-Albert

Figure 10: Query execution time on different synthetic graph types.

through a strongly connected cluster of nodes with a representative. We use this type to show the performance of GROW2REP, by placing the query keywords on different line graphs. The fourth type is a random graph based on the *Barabasi-Albert* (BA) model [4], which generates scale-free networks with only a few nodes (hubs) of the graph having a much higher degree than the rest. The graph in this model is created in a two-stage process. During the first stage, a network of some nodes is created. Then, during the second stage, new nodes are inserted in the graph and they are connected to nodes created during the first stage. We set every node created at the second stage to be connected with exactly one node created at the first stage. In the following, The black line shows the time elapsed until the first answer is found, whereas the grey line shows the overall execution time.

Figure 10 includes the results that we obtained by querying the synthetic datasets. The black line shows the time elapsed until the first answer is found, whereas the grey line shows the overall execution time.

Figure 10a shows the execution time of our algorithm when executing a query with two keywords on a line graph, as we vary the number of nodes of the graph. We place the keywords on the two "ends" of the graph to show the distance's impact on the execution time. Our algorithm's performance is naturally affected by the size of the graph, as it generates $2 \cdot N$ answer trees, where N is the number of nodes. Given that this is a line graph, there is

only one answer, which is the whole graph, and, therefore, the time to find the first answer is also the overall execution time.

Figure 10b shows the performance of our algorithm on a chain graph. The execution times for the first answer are almost the same, as the graph size increases slowly. Instead, the overall execution times increase at a much higher (exponential) rate; note the logarithmic scale of the y axis. The reason is that every pair of nodes is connected with two edges, which increases the number of answers exponentially with the number of nodes in the graph.

In Figure 10c, we report the execution time for the star graph. We place keywords in two different lines connected through the center of the graph, forcing the algorithm to use Grow2Rep, whereas in the previous cases it only had to use Grow and Merge. The number of branches, depicted on the x axis, corresponds to the number of line graphs connected in the star. Each line graph has 10 nodes, and we place the query keywords at the extremities of two different line graphs. The number of merges is exponential to the number of branches, that is $\mathcal{O}(2^K)$ where K is the number of branches since the algorithm will check all possible answers. This behavior is clearly shown in both lines of Figure 10c, where on the y axis (in logarithmic scale) we show the times to find the first, and, respectively, all answers. Above 12 branches, the timeout of 120 seconds that we have set is hit and, thus, the search is terminated, as shown when we search for all answers.

Figure 10d depicts the query performance with the Barabasi-Albert model. We fix the graph size to 2000 nodes and we vary the position of two keywords, by choosing nodes that have a distance, as given in the x axis; note the logarithmic y axis. As the graph is randomly generated within the BA model, we note some irregularity in the time to the first solution, which, however, grows at a moderate pace as the distance between the keyword node grows. The overall relationship between the time to the first solution and the total time confirms that the search space is very large but that most of the exploration is not needed, since the first solution is found quite fast.

11.3.2. Queries on the complete, real-world graph

Next, we describe results that we obtained querying a graph obtained by loading all the real-world data sources described in Section 11.1. We report our findings in Table 2. The queries feature terms that appear in recent French news; they are related to the economy, the Covid crisis, world events, and/or French politics. In most queries, keywords are exact entity names, with their most common spelling. In other cases, we allowed node labels to

Query keyword(s)	Answers	Answer trees	Time to 1st (ms)
aéronautique, *Macron*	2779	1152577	7225
Brigitte Macron, Clara Gaymard	4584	545020	1412
Chine, *covid*, *France*	17	25974	8380
chômage, *covid*	108	205584	4476
covid, El Khomri	16	215952	52486
confinement, Christophe Castaner	4	120367	3820
Chine, France, Didier Raoult	36	146261	14666
Ebola, Raoult	1	37751	75759
entreprise, *Raffarin*	6336	1174822	6589
Julien Denormandie, Macron	464	20181	2661
Khalid al-Falih, Kristalina Georgieva	1	1775	765
Kristalina Georgieva, Walter Butler	1	3224	353
Louis Beam, Ku Klux Klan, Trump	1	24172	15207
Macron, *Royal*	102	6413	4107
Marisol Touraine, Jean-François Delfraissy	3	1497	1183
masque, *France*	35	15082	7126
Michael Ryan, Anthony Fauci	2	6653	12215
Pascale Gruny, Jean-François Delfraissy	2	1332	91591
vaccination, *Trump*	12	1188	6518
Yazdan Yazdanpanah, Jean-François Delfraissy	29	5446	1687

Table 2: Query results on the complete graph.

approximately match a keyword (based on PostgreSQL' stemming and string pattern matching); such keywords are shown in italic in the table. Again, we gave a timeout of 2 minutes, and on this large graph, all the GAM searches stopped at a time-out. The table shows that queries return a varied number of answers, but many ATs are developed in all cases, and the first is found quite before the timeout. Finally, an inspection of the results showed that most are obtained from different datasets, confirming the interest in linking datasets in ConnectionLens. These results show the feasibility and interest of GAMSearch on large heterogeneous graphs.

12. Related work and perspectives

Our work belongs to the area of *data integration* [14]. Data integration can be achieved either in a *warehouse* fashion (consolidating all the data sources into a single repository), or in a *mediator* fashion (preserving the data in their original repository, and querying through a mediator module which distributes the work to each data source, and combines the results). In this work, in contrast with our previous platforms built for data journalism [21, 6], we (*i*) pursue a **warehouse** approach; (*ii*) base our architecture on a relational DBMS; (*iii*) simplify the query paradigm to keyword querying.

ConnectionLens integrates a wide variety of data sources: JSON, relational, RDF and text since [10], to which we have added XML, multidimensional tables, and PDF documents. We integrate such heterogeneous content in a graph, therefore, our work recalls the production of *Linked Data*. A significant difference is that *we do not impose that our graph is RDF*, and *we do not assume, require, or use a domain ontology*.

Graphs are also produced when *constructing knowledge bases*, e.g., Yago [35, 46]. Our setting is more limited in that we are only allowed to integrate a given set of datasets that journalists trust to not "pollute" the database. Therefore, we *use* a KB only for disambiguation and accept (as stated in Section 1) that the KB does not cover some entities found in our input datasets. Our simple techniques for matching (thus, connecting) nodes are reminiscent of data cleaning, entity resolution [44, 39], and key finding in knowledge bases, e.g. [45]. Much more elaborate techniques exist, notably, when the data is regular, its structure is known and fixed, an ontology is available, etc.; none of these holds in our setting.

Keyword search (KS) is widely used for searching in unstructured (typically text) data, and it is also the best search method for novice users, as witnessed by the enormous success of keyword-based search engines. As databases grow in size and complexity, KS has been proposed as a method for searching *also* in structured data [50], when users are not perfectly familiar with the data, or to get answers enabled by different tuple connections. For relational data, in [27] and subsequent works, tuples are represented as nodes, and two tuples are interconnected through primary key-foreign key pairs. The resulting graphs are thus quite uniform, e.g., they consist of "Company nodes", "Employee nodes" etc. The same model was considered in [12, 43, 48, 51]; [43] also establishes links based on similarity of constants appearing in different relational attributes. As explained in Section 8.2, our problem is (much) harder since our trees can traverse edges in both directions, and paths can be (much) longer than those based on PK-FK alone. [49] proposes to incorporate user feedback through active learning to improve the quality of answers in a relational data integration setting. We are working to devise such a learning-to-rank approach for our graphs, also.

KS has also been studied in **XML documents** [24, 33], where an answer is defined as a subtree of the original document, whose leaves match the query keywords. This problem is much easier than ours, since: (i) an XML document is a tree, guaranteeing just one connection between any two nodes; in contrast, there can be any number of such connections in our graphs;

(*ii*) the maximum size of an answer to a k-keywords query is $k \cdot h$ where h, the height of an XML tree, is almost always quite small, e.g., 20 is considered "quite high"; in contrast, with our bi-directional search, the bound is $k \cdot D$ where D is the diameter of our graph - which can be much larger.

Our GROW and MERGE steps are borrowed from [13, 25], which address KS for **graphs**, assuming optimal-substructure, which does not hold for us, and single-direction edge traversal. For **RDF graphs** [16, 30] traverse edges in their direction only; moreover, [30] also make strong assumptions on the graph, e.g., that all non-leaf nodes have types, and that there are very few types (regular graph). In [11], the authors investigate a different kind of answers, called r-clique graphs, which they find using specific indexes.

Keyword search across **heterogeneous datasets** has been previously studied in [15, 31]. However, in these works, *each answer comes from a single dataset*, that is, they never consider answers spanning over and combining multiple datasets, such as the one shown in Figure 3. In turn, multiple datasets lead to equivalence and similarity edges; we have shown how to compactly encode the latter using representatives, and efficiently enumerate solutions which may span data, equivalence, and similarity edges.

A recent work [34] leverages text data sources to find more answers to queries over knowledge graphs. An important difference is that we integrate all datasets *prior to querying* whereas in [34], text is accessed when required to complement the loaded data graph. The effort we invest in building the graphs pays off by making queries faster. Further, our work is more general in the data formats we support. Finally, [34] applies a set of heuristics, tied to the nature of the graphs they used, to find the most relevant answers; our work does not make such assumptions.

(G)STP has been addressed under **simplifications** that do not hold in our context. For instance: the quality of a solution decreases exponentially with the tree size, thus search can stop when all trees are under a certain threshold [7]; edges are considered in a single direction [51, 16, 30]; the cost function has the suboptimal-structure property [13, 32] etc. These assumptions reduce the computational cost; in contrast, to leave our options open as to the best score function, we build a feasible solution for the general problem we study. Some works have focused on finding **bounded (G)STP approximations**, i.e., (G)STP trees solutions whose cost is at most f times higher than the optimal cost, e.g., [20, 23]. Beyond the differences between our problem and (G)STP, due notably to the fact that our score is much more general (Section 9), non-expert users find it hard to set f.

Acknowledgements. We thank Julien Leblay for his contribution to earlier versions of this work [10]. This work has been partially funded by the AI Chair project SourcesSay Grant no ANR-20-CHIA-0015-01 and by Portuguese national funds through FCT with reference UIDB/50021/2020 (INESC-ID).

References

[1] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf. Flair: An easy-to-use framework for state-of-the-art NLP. In *ACL*, 2019.

[2] A. Akbik, D. Blythe, and R. Vollgraf. Contextual string embeddings for sequence labeling. In *ACL*, 2018.

[3] R. Alotaibi, D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *SIGMOD*, 2019.

[4] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439), 1999.

[5] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *TACL*, 5, 2017.

[6] R. Bonaque, T. D. Cao, B. Cautis, F. Goasdoué, J. Letelier, I. Manolescu, O. Mendoza, S. Ribeiro, X. Tannier, and M. Thomazo. Mixed-instance querying: a lightweight integration architecture for data journalism. *PVLDB*, 9(13), 2016.

[7] R. Bonaque, B. Cautis, F. Goasdoué, and I. Manolescu. Social, structured and semantic search. In *EDBT*, 2016.

[8] O. Bălălău, C. Conceiçao, H. Galhardas, I. Manolescu, T. Merabti, J. You, and Y. Youssef. Graph integration of structured, semistructured and unstructured data for data journalism. In *BDA (informal publication only)*, 2020.

[9] T. D. Cao, I. Manolescu, and X. Tannier. Extracting linked data from statistic spreadsheets. In *Semantic Big Data Workshop*, 2017.

[10] C. Chanial, R. Dziri, H. Galhardas, J. Leblay, M. H. L. Nguyen, and I. Manolescu. ConnectionLens: Finding connections across heterogeneous data sources. *VLDB*, 2018.

[11] Y.-R. Cheng, Y. Yuan, J.-Y. Li, L. Chen, and G.-R. Wang. Keyword query over error-tolerant knowledge bases. *Journal of Computer Science and Technology*, 31, 2016.

[12] P. de Oliveira, A. S. da Silva, and E. S. de Moura. Ranking candidate networks of relations to improve keyword search over relational databases. In *IEEE*, 2015.

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

[14] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[15] X. Dong and A. Halevy. Indexing dataspaces. In *SIGMOD*. ACM, 2007.

[16] S. Elbassuoni and R. Blanco. Keyword search over RDF graphs. In *CIKM*, 2011.

[17] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *ACL*, 2005.

[18] O. Galibert, S. Rosset, C. Grouin, P. Zweigenbaum, and L. Quintard. Extended named entity annotation on OCRed documents: From corpus constitution to evaluation campaign. In *LREC*, 2012.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co. New York, 1990.

[20] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group Steiner tree problem. In *SIAM*, 1998.

[21] F. Goasdoué, K. Karanasos, Y. Katsis, J. Leblay, I. Manolescu, and S. Zampetakis. Fact Checking and Analyzing the Web. In *SIGMOD*, 2013.

[22] J. Gray, L. Chambers, and L. Bounegru. *The Data Journalism Handbook: How Journalists can Use Data to Improve the News*. O'Reilly, 2012.

[23] A. Gubichev and T. Neumann. Fast approximation of Steiner trees in large graphs. In *CIKM*, 2012.

[24] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[25] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[26] J. Hoffart, M. A. Yosef, I. Bordino, H. Fürstenau, M. Pinkal, M. Spaniol, B. Taneva, S. Thater, and G. Weikum. Robust disambiguation of named entities in text. In *EMNLP*, 2011.

[27] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, 2002.

[28] M. A. Jaro. Advances in record linkage methodology as applied to the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, 84, 1989.

[29] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases*, 2016.

[30] W. Le, F. Li, A. Kementsietsidis, and S. Duan. Scalable keyword search on large RDF data. *IEEE Trans. Knowl. Data Eng.*, 26(11), 2014.

[31] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.

[32] R. Li, L. Qin, J. X. Yu, and R. Mao. Efficient and progressive group Steiner tree search. In *SIGMOD*, 2016.

[33] Z. Liu and Y. Chen. Identifying meaningful return information for XML keyword search. In *ACM SIGMOD*, pages 329–340, 2007.

[34] X. Lu, S. Pramanik, R. S. Roy, A. Abujabal, Y. Wang, and G. Weikum. Answering complex questions by joining multi-document evidence with quasi knowledge graphs. In *ACM SIGIR*, pages 105–114. ACM, 2019.

[35] F. Mahdisoltani, J. Biega, and F. M. Suchanek. YAGO3: A knowledge base from multilingual wikipedias. In *CIDR*, 2015.

[36] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[37] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30, 2007.

[38] J. Nothman, N. Ringland, W. Radford, T. Murphy, and J. R. Curran. Learning multilingual named entity recognition from wikipedia. *Artificial Intelligence*, 194, 2013.

[39] G. Papadakis, E. Ioannou, and T. Palpanas. Entity resolution: Past, present and yet-to-come. In *EDBT*. OpenProceedings.org, 2020.

[40] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[41] B. Sagot, M. Richard, and R. Stern. Referential named entity annotation of the Paris 7 French TreeBank) [in French]. In *TALN*, 2012.

[42] H. Sahovic. Graph querying in ConnectionLens (internship report), 2019. Available at `https://team.inria.fr/cedar/files/2021/04/Haris-SAHOVIC-2019-report.pdf`.

[43] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, 2007.

[44] F. M. Suchanek, S. Abiteboul, and P. Senellart. PARIS: probabilistic alignment of relations, instances, and schema. *Proc. VLDB Endow.*, 5(3), 2011.

[45] D. Symeonidou, V. Armant, and N. Pernelle. BECKEY: understanding, comparing and discovering keys of different semantics in knowledge bases. *Knowl. Based Syst.*, 195, 2020.

[46] T. P. Tanon, G. Weikum, and F. M. Suchanek. YAGO 4: A reason-able knowledge base. In *ESWC*, 2020.

[47] D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[48] Q. H. Vu, B. C. Ooi, D. Papadias, and A. K. H. Tung. A graph method for keyword-based selection of the top-k databases. In *SIGMOD*, 2008.

[49] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Active learning in keyword search-based data integration. *VLDB J.*, 24(5), 2015.

[50] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Synthesis Lectures on Data Management. 2009.

[51] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1), 2010.