Empowering Investigative Journalism with Graph-based Heterogeneous Data Management

Angelos-Christos Anadiotis[⋆], Oana Balalau[◇], Théo Bouganim[◇], Francesco Chimienti[◇], Helena Galhardas[‡], Mhd Yamen Haddad[◇], Stéphane Horel[†], Ioana Manolescu[◇], Youssr Youssef[◇],
[⋆]École Polytechnique, EPFL &IPP, [◇]Inria & IPP, [‡]INESC-ID & IST, Univ. Lisboa, [†]Le Monde

Abstract

Investigative Journalism (IJ, in short) is staple of modern, democratic societies. IJ often necessitates working with large, dynamic sets of heterogeneous, schema-less data sources*, which can be* structured, semi-structured, or textual*, limiting the applicability of classical data integration approaches. In prior work, we have developed ConnectionLens, a system capable of integrating such sources into a single heterogeneous graph, leveraging Information Extraction (IE) techniques; users can then query the graph by means of keywords, and explore query results and their neighborhood using an interactive GUI. Our keyword search problem is complicated by the graph heterogeneity, and by the lack of a result score function that would enable pruning of the search space.*

In this work, we describe an actual IJ application *studying conflicts of interest in the biomedical domain, and we show how ConnectionLens supports it. Then, we present* novel techniques addressing the scalability challenges *raised by this application: one allows us to reduce the significant IE costs while building the graph, while the other is a novel, parallel, in-memory keyword search engine, which achieves orders of magnitude speed-up over our previous engine. Our experimental study on the real-world IJ application data confirms the benefits of our contributions.*

1 Introduction

Journalism and the press are a critical ingredient of any modern society. Like many other industries, such as trade, or entertainment, journalism has benefitted from the explosion of Web technologies, which enabled instant sharing of their content with the audience. However, unlike trade, where databases and data warehouses had taken over daily operations decades before the Web age, *many newsrooms discovered the Web and social media, long before building strong information systems where journalists could store their information and/or ingest data of interest for them.* As a matter of fact, journalists' desire to protect their confidential information may also have played a role in delaying the adoption of data management infrastructures in newsrooms.

At the same time, highly appreciated journalism work often requires *acquiring, curating, and exploiting large amounts of digital data*. Among the authors, S. Horel co-authored the "Monsanto Papers" series which obtained the European Press Prize Investigative Reporting Award in 2018 [1]; a similar project is the "Panama Papers" (later known as "Offshore Leaks") series of the International Consortium of Investigative Journalists [2]. In such

Copyright 2021 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

works, journalists are forced to work with *heterogeneous* data, potentially in *different data models* (*structured* such as relations, *semistructured* such as JSON or XML documents, or graphs, including but not limited to RDF, as well as *unstructured text*). We, the authors, are currently collaborating on such an **Investigative Journalism** (IJ, in short) application, focused on the study of **situations potentially leading to conflicts of interest**[1] (CoIs, in short) between biomedical experts and various organizations: corporations, industry associations, lobbying organizations or front groups. Information of interest in this setting comes from: scientific publications (in PDF) where authors declare e.g., "Dr. X. Y. has received consulting fees from ABC"; semi-structured metadata (typically XML, used for instance in PubMed), where authors may also specify such connections; a medical association, say, French Cardiology, may build its own disclosure database which may be relational, while a company may disclose its ties to specialists in a spreadsheet.

This paper builds upon our recent work [3], where we have identified a set of requirements (**R**) and constraints (**C**) that need to be addressed to efficiently support IJ applications. We recall them here for clarity and completeness:

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

R2. Little to no effort required from users: 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 challenging. Thus, the data analysis pipeline needs to be as automatic as possible.

C1. Little-known entities: interesting journalistic datasets feature some extremely well-known entities (e.g., world leaders in the pharmaceutical industry) next to others of much smaller notoriety (e.g., an expert consulted by EU institutions, or a little-known trade association). From a journalistic perspective, such lesser-known entities may play a crucial role in making interesting connections among data sources, e.g., the association may be created by the industry leader, and it may pay the expert honoraries.

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

R3. Performance on "off-the-shelf" hardware: The efficiency of our data processing pipeline is important; also, the tool should run on general-purpose hardware, available to users like the ones we consider, without expertise or access to special hardware.

Further, IJ applications' data analysis needs entail:

R4. Finding connections across heterogeneous datasets is a core need. In particular, it is important for our approach to be tolerant of inevitable differences in the organization of data across sources. Heterogeneous data integration works, such as [4–6], and recent heterogeneous polystores, e.g., [7–9] assume that sources have well-understood schemas; other recent works, e.g., [10–12] focus on analyzing large sets of Open Data sources, all of which are tabular. IJ data sources do not fit these hypotheses: data can be semi-structured, structured, or simply text. Therefore, we opt for **integrating all data sources in a heterogeneous graph** (with no integrated schema) and for **keyword-based querying** (where users specify some terms); the system returns subtrees of the graph that connect nodes matching these terms.

C3. Lack of single, well-behaved answer score: After discussing several journalistic scenarios, we have not been able to identify a unique method (score) for deciding which are the best answers to a query. 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 a French scientist is connected to a French company through their nationality, are not of interest. Therefore, unlike prior keyword search algorithms, which fix a score function and exploit it to prune the search, our algorithm must be orthogonal and work it with any score function.

[1] According to the 2011 French transparency law, "A conflict of interest is any situation where a public interest may interfere with a public or private interest, in such a way that the public interest may be, or appear to be, unduly influenced."

Building upon our previous work, and years-long discussions of IJ scenarios, this paper makes the following contributions:

- We describe the CoI IJ application proposed by S. Horel (Section 2); we extract its technical requirements and we devise an end-to-end data analysis pipeline addressing these requirements (Section 3).
- We provide application-driven optimizations, inspired from the CoI scenario but reusable to other contexts, which speed up the graph construction process (Section 4).
- We introduce a parallel, in-memory version of the keyword search algorithm described in [3, 13], and we explain our design in both the physical database layout and the parallel query execution (Section 5).
- We evaluate the performance of our system on synthetic and real-world data. We demonstrate its scalability, and demonstrate performance improvements of several orders of magnitude over our prior work, thereby enabling the journalists to perform interactive exploration of their data (Section 6).

2 Use case: conflicts of interest in the biomedical domain

The topic. Biomedical experts such as health scientists and researchers in life sciences play an important role in society, advising governments and the public on health issues. They also routinely interact with industry (pharmaceutical, agrifood etc.), consulting, collaborating on research, or otherwise sharing work and interests. To trust advice coming from these experts, it is important to ensure the advice is not unduly influenced by vested interests. Yet, IJ work, e.g. [14–16], has shown that disclosure information is often scattered across multiple data sources, hindering access to this information. We now illustrate the data processing required to gather and collectively exploit such information.

Sample data. Figure 1 shows a tiny fragment of data that can be used to find connections between scientists and companies. *For now, consider only the nodes shown as a black dot or as a text label, and the solid, black edges connecting them; these model directly the data. The others are added by ConnectionLens as we discuss in Section 3.1.* (i) Hundreds of millions of bibliographic notices (in **XML**) are published on the PubMed web site; the site also links to research (in PDF). In recent years, PubMed has included an optional CoIStatement element where authors can declare (in free text) their possible links with industrial players; less than 20% of recent papers have this element, and some of those present are empty ("The authors declare no conflict of interest"). (ii) Within the **PDF** papers themselves, paragraphs titled, e.g., "Acknowledgments", "Disclosure statement" etc., may contain

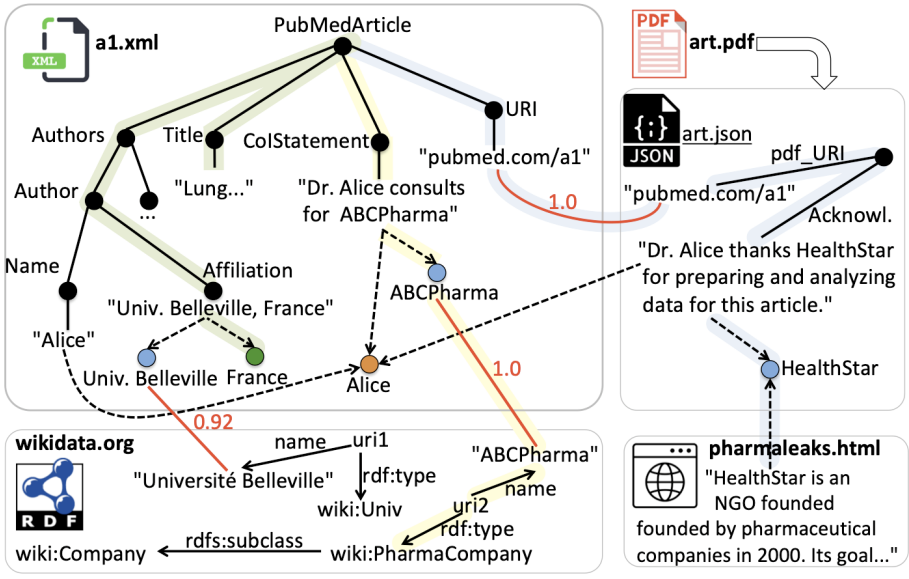


Figure 1: Graph data integration in ConnectionLens.

such information, even if the CoIStatement is absent or empty. This information is accessible if one converts the PDF in a format such as **JSON**. In Figure 1, Alice declares her consulting for ABCPharma in XML, yet the "Acknowledgments" paragraph in her PDF paper mentions HealthStar.[2] (iii) A (subset of a) knowledge base (in **RDF**) such as WikiData describes well-known entities, e.g., ABCPharma; however, less-known entities of interest in an IJ scenario are often missing from such KGs, e.g., HealthStar in our example. (iv) Specialized data sources, such as a trade catalog or a Wiki Web site built by other investigative journalists, may provide information on some such actors: in our example, the PharmaLeaks Web site shows that HealthStar is also funded by the industry. Such a site, established by a trusted source (or colleague), even if it has little or no structure, is a gold mine to be reused, since it saves days or weeks of tedious IJ work. *In this and many IJ scenarios, sources are highly heterogeneous, while time, skills, and resources to curate, clean, or structure the data are not available*.

Sample query. Our application requires *the connections of specialists in lung diseases, working in France, with pharmaceutical companies*. In Figure 1, the edges with *green* highlight and those with *yellow* highlight, together, form an answer connecting Alice to ABCPharma (spanning over the XML and RDF sources); similarly, the edges highlighted in *green* together with those in *blue*, spanning over XML, JSON and HTML, connect her to HealthStar.

The potential impact of a CoI database. A database of known relationships between experts and companies, built by integrating heterogeneous data sources, would be a valuable asset. In Europe, such a database could be used, e.g., to select, for a committee advising EU officials on industrial pollutants, experts with few or no such relationships. In the US, the Sunshine Act [17], just like the French 2011 transparency law, requires manufacturers of drugs and medical devices to declare such information. However, this does not extend to companies from other sectors.

3 Investigative journalism pipeline

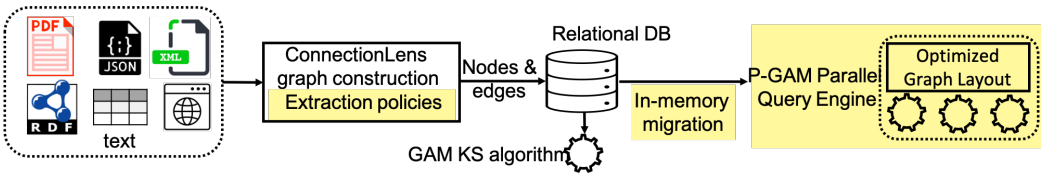


Figure 2: Investigative Journalism data analysis pipeline.

The pipeline we have built for IJ is outlined in Figure 2. First, we recall ConnectionLens graph construction (Section 3.1), which integrates heterogeneous data into a graph, stored and indexed in PostgreSQL. On this graph, the GAM keyword search algorithm (recalled in Section 3.2) answers queries such as our motivating example; these are both detailed in [3]. The modules with yellow background in Figure 2 are the novelties of this work, and will be introduced below: scenario-driven performance optimizations to the graph construction (Section 4), and an in-memory, parallel keyword search algorithm, called P-GAM (Section 5).

3.1 ConnectionLens graph construction

ConnectionLens integrates JSON, XML, RDF, HTML, relational or text data into a graph, as illustrated in Figure 1. Each source is mapped to the graph as close to its data model as possible, e.g., XML edges have no labels while internal nodes all have names, while in JSON conventions are different. Next, ConnectionLens **extracts named entities from all text nodes**, regardless the data source they come from, using trained language models. In the

[2]This example is inspired from prior work of S. Horel where she identified (manually inspecting thousands of documents) an expert supposedly with no industrial ties, yet who authored papers for which companies had supplied and prepared data.

figure, blue, green, and orange nodes denote Organization, Location, and Person entities, respectively. Each such entity node is connected to the text node it has been extracted from, by an *extraction edge* recording also the confidence of the extraction (dashed in the figure). **Entity nodes are shared across the graph**, e.g., Person:Alice has been found in three data sources, Org:BestPharma in two sources etc. ConnectionLens includes a *disambiguation* module which avoids mistakenly unifying entities with the same labels but different meanings. Finally, nodes with similar labels are *compared*, and if their similarity is above a threshold, a **sameAs** (red) edge connecting them is introduced, labeled with the similarity value.

A sameAs edge with similarity 1.0 is called an *equivalence edge*. Then, p equivalent nodes, e.g., the entity ABCPharma and the identical-label RDF literal, would lead to $p(p-1)/2$ equivalence edges. To keep the graph compact, one of the p nodes is declared the *representative* of all p nodes, and instead, we only store the $p-1$ equivalence edges adjacent to the representative. Details on the graph construction steps can be found in [3].

Formally, a ConnectionLens graph is denoted $G = (N, E)$, where nodes can be of different types (URIs, XML elements, JSON nodes, etc., including extracted entities) and edges encode data source structure, the connection between extracted entities and the text in which they were found, as well as node label similarity.

3.2 The GAM keyword search algorithm

We view our motivating query, on highly heterogeneous content with no a-priori known structure, as a **keyword search query over a graph**. Formally, a query $Q = \{w_1, w_2, \ldots, w_m\}$ is a set of m keywords, and an *answer tree* (AT, in short) is a set t of G edges which (i) together, form a tree, and (ii) for each w_i, contain at least one node whose label matches w_i. We are interested in *minimal* answer trees, i.e., answer trees that satisfy the following properties: (i) removing an edge from the tree will make it lack at least one keyword match, and (ii) if mulitple nodes match a query keyword, then all matching nodes are related through sameAs links with similarity 1.0. In the literature (see Section 7), a *score function* is used to compute the quality of an answer, and only the best k ATs are returned, for a small integer k. Our problem is harder since: (i) our ATs may span across different data sources, even of different data models; (ii) they may traverse an edge **in its original or in the opposite direction**, e.g., to go from JSON to XML through Alice; this doubles the search space, compared to a directed graph where a single direction is considered; and (iii) **no single score function serves all IJ needs** since, depending on the scenario, journalists may favor different (incompatible) properties of an AT, such as "being characteristic of the dataset" or, on the contrary, "being surprising". Thus, **we cannot rely on special properties of the score function** to help us prune unpromising parts of the search space, as done in prior work (see Section 7). Intuitively, tree size could be used to limit the search: very large answer trees (say, of more than 100 edges) generally do not represent meaningful connections. However, in heterogeneous, complex graphs, users find it hard to set a size limit for the exploration. Nor is a smaller solution always better than a larger one. For instance, an expert and a company may both have "nationality" edges leading to "French" (a solution of 2 edges), but that may be less interesting than finding that the expert has written an article specifying in its CoIStatement funding from the company (which could span 5 edges or more).

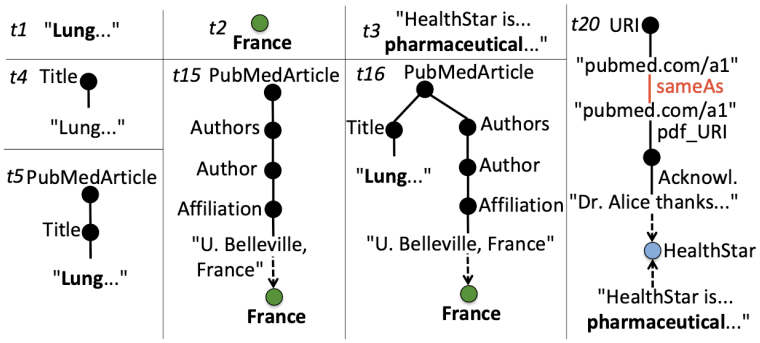


Figure 3: Trees built by GAM for our sample query.

Our **Grow-and-Aggressive-Merge (GAM)** algorithm [3, 13] enumerates trees exhaustively, until a number of answers are found, or a time-out. First, it builds 1-node trees from the nodes of G which match 1 or more keywords, e.g., t_1, t_2, t_3 in Figure 3, showing some partial trees built when answering our sample query. The

keyword match in each node label appears in bold. Then, GAM relies on two steps. **Grow** adds to the root of a tree one of its adjacent edges in the graph, leading to a new tree: thus t_4 is obtained by Grow on t_1, t_5 by Grow on t_4, and successive Grow steps lead from t_2 to t_{15}. Similarly, from t_3, successive Grow's go from the HTML to the JSON data source (the HealthStar entity occurs in both), and then to the XML one, building t_{20}. Second, as soon as a tree is built by Grow, it is **Merge**d with all the trees already found, rooted in the same node, matching different keywords and having disjoint edges wrt the given tree. For instance, assuming t_{15} is built after t_5, they are immediately merged into the tree t_{16}, having the union of their edges. Each Merge result is then merged again with all qualifying trees (thus the "agressive" in the algorithm name). For instance, when Grow on t_{20} builds a tree rooted in the PubMedArticle node (not shown; call it t_A), Merge(t_{16}, t_A) is immediately built, and is exactly the answer highlighted with green and blue in Figure 1. Section 5.2 explains the steps of GAM in the context of its parallel version introduced in this paper.

Together, Grow and Merge are guaranteed to generate all solutions. If $m = 2$, Grow alone is sufficient, while $m \geq 3$ also requires Merge. *GAM may build a tree in several ways*, e.g., the answer above could also be obtained as Merge(Merge(t_{15}, Grow(t_{20})), t_5); GAM keeps a history of already explored trees, to avoid repeating work on them. Importantly, GAM can be used with any score function. Its details are described in [3, 13].

4 Use case-driven optimization

In this section, we present an optimization we brought to the graph construction process, guided by our target application.

In the experiments we ran, Named Entity Recognition (NER) took up to 90% of the time ConnectionLens needs to integrate data sources into a graph. The more textual the sources are, the more time is spent on NER. Our application data lead us to observe that:

- Some text nodes ()e.g., those found on the path PubMedArticle.Authors.Author.Name) *always correspond to entities of a certain type* (in our example, Person). If this information is given to ConnectionLens, it can create a Person entity node, like the Alice node in Figure 1, *without calling the expensive NER procedure*.
- Other text nodes may be deemed *uninteresting for extraction* because journalists think no interesting entities appear there. If ConnectionLens is aware of this, it can *skip the NER call on such text nodes*. Observe that the input data, including all its text nodes, is always preserved; we only avoid extraction effort deemed useless (but it can still be applied later if application requirements evolve).

To exploit this insight, we introduced a notion of **context**, and allow users to specify **(optional) extraction policies**. A context is an expression designating a set of text nodes in one or several data sources. For instance, a context specified by the rooted path PubMedArticle.Authors.Author.Name designates all the text values of nodes found on that path in an XML data source; the same mechanism applies to an HTML or JSON data source. In a relational data source containing table R with attribute a, a context of the form $R.a$ designates all text nodes in the ConnectionLens graph obtained from a value of the attribute a in relation R. Finally, an RDF property p used as context designates all the values o such that a triple (s, p, o) is ingested in a ConnectionLens graph.

Based on contexts, an extraction policy takes one of the following form: (i) c *force* T_e, where c is a context and T_e is an entity type, e.g., Person, states that each node designated by the context is exactly one instance of T_e ; (ii) c *skip*, to indicate that NER should not be performed on the text nodes designated by c; (iii) as syntactic sugar, for hierarchical data models (e.g., XML, JSON etc.), c *skipAll* states that NER should not be performed on the text nodes designated by c. This allows larger-granularity control of NER on different portions of the data.

Observe that our contexts (thus, our policies) are specified *within a data model*; this is because the *regularity that allows defining them* can only be hoped for within data sources with identical structure. Policies allow journalists to state what is obvious to them, and/or what is not interesting, in the interest of graph construction

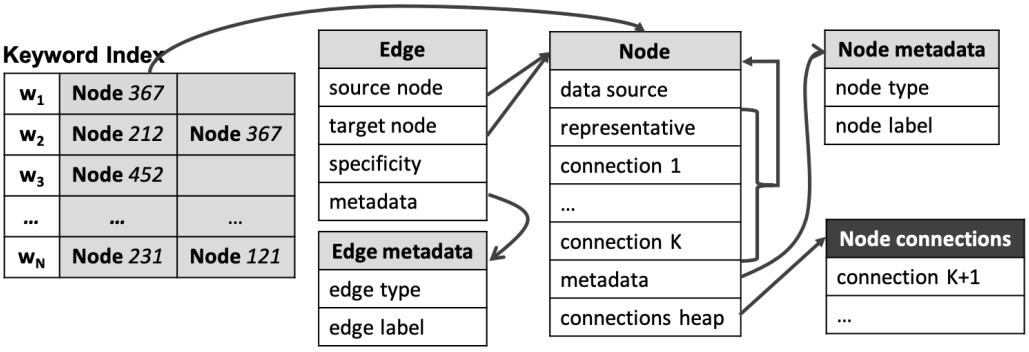


Figure 4: Physical graph layout in memory.

speed. *Force* policies may also improve graph quality, by making sure NER does not miss any entity designated by the context.

5 In-memory parallel keyword search

We now describe the novel keyword search module that is the main technical contribution of this work. A *in-memory graph storage model* specifically designed for our graphs and with keyword search in mind (Section 5.1) is leveraged by a a *multi-threaded, paralell* algorithm, called P-GAM (Section 5.2), which is a parallel extension of our original GAM algorithm outlined in Section 3.2.

5.1 Physical in-memory database design

The size of the main memory in modern servers has grown significantly over the past decade. Data management research has by now led to several DB engines running entirely in main memory, such as Oracle Database In-Memory, SAP HANA, and Microsoft SQL Server with Hekaton. Moving data from hard disks to main memory significantly boosts performance by avoiding disk I/O costs. However, it introduces new challenges on the optimization of the data structures and the execution model for a different bottleneck: memory access [18].

We have integrated P-GAM inside a novel in-memory graph database, which we have built and optimized for P-GAM operations. The physical layout of a graph database is important, given that graph processing is known to suffer from random memory accesses [19–22]. Our design (i) includes all the data needed by applications as described in Section 2, while also (ii) aiming at high-performance, parallel query execution in modern scale-up servers, in order to tackle huge search spaces (Section 3.2).

We start with the scalability requirements. Like GAM, P-GAM also performs Grow and Merge operations (recall Figure 3). To enumerate possible *Grow* steps, P-GAM needs to access all edges adjacent to the root of a tree, as well as the representative (Section 3.1) of the root, to enable growing with an equivalence edge. Further, as we will see, P-GAM (as well as GAM) relies on a simple edge metric, called **specificity**, derived from the number of edges with the same label adjacent to a given node, to decide the best neighbor to Grow to. For instance, if a node has 1 spouse and 10 friend edges, the edge going to the spouse is more specific than one going to a friend. A *Merge* does not need more information than available in its input trees; instead, it requires specific run-time data structures, as we describe below.

In our memory layout, we **split the data required for search, from the rest**, as the former are critical for performance; we refer to the latter as metadata. Figure 4 depicts the memory tables that we use. The *Node* table includes the ID of the data source where the node comes from, and references to each node's: (i) representative, (ii) K neighbors, if they exist (for a fixed K, which is pre-allocated), (iii) metadata, and (iv) other neighbors, if they exist (dynamically allocated beyond K). We separate the allocation of neighbors into static and dynamic,

Algorithm 1: P-GAM

Input: $G = (N, E)$, query Q={w$_1$, . . ., w$_m$}, maximum number of solutions M, maximum time limit
Output: Answer trees for Q on G

```
1 pQueue_i ← new priority queue of (tree, edge) pairs, 1 ≤ i ≤ nt;
2 N_Q ← ∪_{w_i∈Q} keywordIndex.lookup(w_i);
3 for n ∈ N_Q, e edge adjacent to n do
4 |   push (n, e) on some pQueue_j (distribute equally)
5 end
6 launch nt P-GAM Worker (Algorithm 2) threads;
7 return solutions
```

to keep K neighbors in the main Node structure, while the rest are placed in a separate heap area, stored in the *Node connections* table. This way, we can allocate a fixed size to each Node, efficiently supporting the memory accesses of P-GAM. In our implementation, we set $K = 5$; in general, it can be set based on the median degree of the graph vertices. The *Node metadata* table includes information about the type of each node (e.g., JSON, HTML, etc.) and its label, comprising the keywords that we use for searching the graph. The *Edge* table includes a reference to the source and the target node of every edge, the edge specificity, and a reference to the edge metadata. The *Edge metadata* table includes the type and the label of each edge. Finally, we use a *keywordIndex*, which is a hash-based map associating every node with its labels. P-GAM probes the keywordIndex when a query arrives to find the references to the Node table that match the query keywords and start the search from there. The labels are encoded in order to achieve a more compact representation, while also indexed to allow prefix matching, following the work in [23]. Among all the structures, only *Node connections* (singled out by a dark background in Figure 4) is in a dynamically allocated area; all the others are statically allocated.

The above storage is *row* (node) oriented, even though column storage often speeds up greatly analytical processing; this is due to the nature of the keyword search problem, which requires traversing the graph from the nodes matching the keywords, in BFS style. Since we consider ad-hoc queries (any keyword combinations), there are no guarantees about the order of the nodes P-GAM visits. Therefore, in our setting, the vertically selective access patterns, which are exploited by column-stores, do not apply. Instead, the crucial optimization here is to *find the neighbors of every node fast.* This is leveraged by our algorithm, as we explain below.

5.2 P-GAM: parallel keyword query execution

Our P-GAM (Parallel GAM) query algorithm builds a set of data structures, which are exploited by concurrent workers (threads) to produce query answers. We split these data structures to shared and private to the workers. We start with the shared ones. The **history** data structure holds all trees built during the exploration, while **treesByRoot** gives access to all trees rooted in a certain node. As the search space is huge, history and treesByRoot grow very much. Specfically, for history, P-GAM first has to make sure that an intermediate AT has not been considered before (i.e., browse the history) before writing a new entry. Similarly, treesByRoot is updated only when a tree changes its root or if there is a Merge of two trees; however, it is probed several times for Merge candidates. Therefore, we have implemented these data structures as lock-free hash-based maps to ensure high concurrency and prioritize read accesses. Observe that, given the high degree of data sharing, keeping these data structures thread-private would not yield any benefit.

Moving to the thread-private data structures, *each thread*, say number i, has a priority queue **pQueue**$_i$, in which we push (tree, edge) pairs, such that the edge is adjacent to the root of the tree. Priority in this queue is determined as follows: we prefer the pairs *whose nodes match most query keywords*; to break a tie, we prefer *smaller trees*; and to break a possible tie among these, we prefer the pair where the edge has the *highest-specificity*. This is a simple priority order we chose empirically; any other priority could be used, with no change to the

Algorithm 2: P-GAM Worker (thread number i out of nt)

```
1  repeat
2      pop (t, e), the highest-priority pair in pQueue_i (or, if empty, from the pQueue_j having the most
         entries);
3      t_G ← Grow(t, e);
4      if t_G ∉ history then
5          for all edges e' adjacent to the root of t_G, push (t_G, e') in pQueue_i;
6          build all t_M ← Merge(t_G, t') where t' ∈ treesByRoot.get(t_G.root) and t' matches Q keywords
             disjoint from those of t_G;
7          if t_M ∉ history then
8              recursively merge t_M with all suitable partners;
9              add all the (new) Merge trees to history;
10             for each new Merge tree t'', and edge e'' adjacent to the root of t'', push (t'', e'') in pQueue_i;
11         end
12     end
13 until time-out or M solutions are found or all pQueue_j empty, for 1 ≤ j ≤ nt;
```

algorithm.

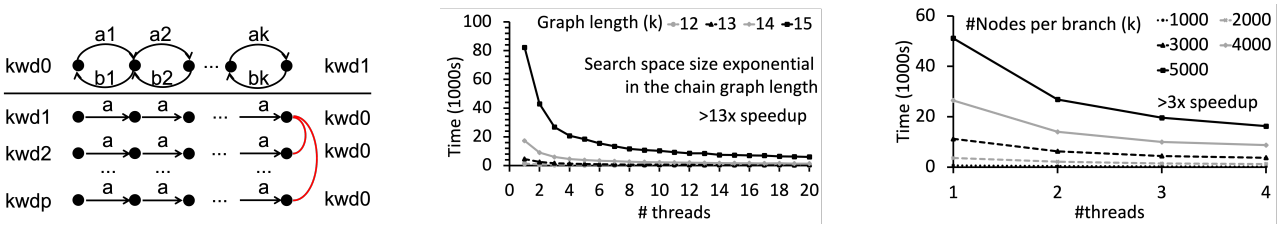
P-GAM keyword search is outlined in Algorithm 1. It creates the shared structures, and nt threads (as many as available based on the availability of computing hardware resources). The search starts by looking up the nodes N_Q matching at least one query keywords (line 2); we create a 1-node tree from each such node, and push it together with an adjacent edge (line 4), in one of the pQueue's (distributing them in round-robin).

Next, nt worker threads run in parallel Algorithm 2, until a global stop condition: time-out, or until the maximum number of solutions has been reached, or all the queues are empty. Each worker repeatedly picks the highest-priority (tree, edge) pair on its queue (line 2), and applies Grow on it (line 3), leading to a 1-edge larger tree (e.g., t_5 obtained from t_4 in Figure 3). Thus, the stack priority *orders* the possible Grow steps at a certain point during the search; it tends to lead to small solutions being found first, so that users are not surprised by the lack of a connection they expected (and which usually involves few connections). If the Grow result tree had not been found before (this is determined from the history), the worker tries to Merge it with all compatible trees, found within treesByRoot (line 6). The Merge partners (e.g., t_5 and t_{15} in Figure 3) should match different (disjoint) keywords; this condition ensures minimality of the solution. Merge results are repeatedly Merge'd again; the thread switches back to Grow only when no new Merge on the same root is possible. Any newly created tree is checked and, if it matches all query keywords, added to the solution set (and not pushed in any queue). Finally, to balance the load among the workers, if one has exhausted his queue, it retrieves the highest-priority (tree, edge) pair from the queue with most entries, pushing the possible results in its own queue.

As seen above, the threads intensely compete for access to history and treesByRoot. As we demonstrate in Section 6.3, our design allows excellent scalability as the number of threads increases.

6 Experimental evaluation

We now present the results of our experimental evaluation. Section 6.1 presents the hardware and data we used. Section 6.2 studies the impact of extraction policies (Section 4). Section 6.3 analyzes the scalability of P-GAM, focusing on its interaction with the hardware, and demonstrates its significant gains with respect to GAM. Section 6.4 demonstrates P-GAM scalability on a large, real-world graph built for our CoI IJ application.



(a) Synthetic graphs: $chain_k$, $star_{p,k}$

(b) Chain graph scaling

(c) Star graph scaling

Figure 5: Synthetic graphs performance

6.1 Hardware and software setup

We used a server with 2x10-core Intel Xeon E5-2640 v4 CPUs clocked at 2.4GHz, and 192GB of DRAM. We do not use Hyper-Threads, and we bind every CPU core to a single worker thread. As shown in Figure 2, we use ConnectionLens (90% Java, 10% Python) to *construct* a graph out of our data sources, and *store* it in PostgreSQL. Following the processing pipeline, we *migrate* the graph to our novel *in-memory graph engine*, which implements P-GAM. The query engine is a NUMA-aware, multi-threaded C++ application.

6.2 Impact of extraction policies

In this experiment, we loaded a set of 20.000 Pubmed XML bibliographic notices (38.4 MB on disk). This dataset inspired the following extraction policy: the text content of any PubMedArticle.Authors.Author.Name is a Person entity, and that extraction is skipped from the article and journal title, as well as from the article keywords. NER is still applied on author affiliations (rich with Organization and Location entities), as well as on the CoIStatement elements of crucial interest in our context.

By introducing the policy, the extraction time went down from 1199s (no policy) to 716s, yielding a speed-up of about 1.67x. The total loading time was reduced from 1461s to 929s, translating to 1.57x speed-up. As a point of reference, we also noted the time to load (and index) the graph nodes and edges in PostgreSQL; extraction strongly dominates the total time, confirming the practical interest of application-driven policies.

6.3 Scalability analysis

The scalability analysis is performed on synthetic graphs, whose size and topology we can fully control. We focus on two aspects that impact scalability: (i) contention in concurrent access to data structures, and (ii) size of the graph (which impacts the search space). To analyze the behavior of concurrent data structures, we use chain_k graphs, because they yield a big number of intermediate results, shared across threads, even for a small graph. This way, we can isolate the size of the graph from the size of the intermediate results. We repeat every experiment five times, and we report the average query execution time.

We use **two shapes of graphs (each with 1 associated query)**, leading to very different search space sizes (Figure 5a). In both graphs, all the kwd_i for $0 \leq i$ are distinct keywords, as well as the labels of the node(s) where the keyword is shown; no other node label matches these keywords. $\textbf{Chain}_k$ has $2k$ edges; on it, $\{\text{kwd}_0, \text{kwd}_1\}$ has 2^k solutions, since any two neighbor nodes can be connected by an a_i or by a b_i edge; further, $2^{k+1} - 2$ partial (non-solution) trees are built, each containing one keyword plus a path growing toward (but not reaching) the other. $\textbf{Star}_{p,k}$ has p branches, each of which is a line of length k; at one extremity each line has a keyword kwd_i, $1 \leq i \leq p$, while at the other extremity, all lines have kwd_0. As explained in Section 3.1, these nodes are equivalent, one is designated their representative (in the Figure, the topmost one), and the others are connected to it through equivalence edges, shown in red. On this graph, the query $\{\text{kwd}_0, \text{kwd}_1, \ldots, \text{kwd}_p\}$ has exactly 1 solution which is the complete graph; there are $O(k+1)2^p$ partial trees.

Graph	chain_{12}	chain_{13}	chain_{14}	chain_{15}	$\text{star}_{4,1000}$	$\text{star}_{4,2000}$	$\text{star}_{4,3000}$	$\text{star}_{4,4000}$	$\text{star}_{4,5000}$
S	4096	8192	16382	32768	1	1	1	1	1
$T_{PGAM}^{1-clean}$	40	92	215	551	34	78	133	196	242
$T_{PGAM}^{1-query}$ (ms)	3	8	17	46	151	693	1957	4711	8592
T_{PGAM} (s)	1	5	17	83	1	4	11	27	51
T^1 (ms)	160	203	234	315	4063	12580	36261	67984	108960
T (s)	674	900	900	900	60	244	900	900	900

Table 1: Single-thread P-GAM vs. GAM performance.

Single-thread P-GAM vs. GAM. We start by comparing P-GAM, *using only 1 thread*, with the (single-threaded) Java-based GAM, accessing graph edges from a PostgreSQL database. We ran the two algorithms on the synthetic graphs and queries, with a *time-out of 15 minutes*; both could stop earlier if they exhausted the search space. Table 1 shows: the number of solutions S, the time $T_{PGAM}^{1-clean}$ (ms) until the internal data structures have been cleaned and properly prepared for queried, the time $T_{PGAM}^{1-query}$ (ms) until the first solution is found by P-GAM and its total running time T_{PGAM} (s) that includes both cleaning and querying for all solutions, as well as the corresponding times T^1 and T for GAM (Java on Postgres). On these tiny graphs, both algorithms found all the expected solutions, however, even without parallelism, P-GAM is $10\times$ to more than $100\times$ faster. Further, on all but the 3 smallest graphs, GAM did not exhaust its search space in 15 minutes. This experiment demonstrates and validates the expected speed-up of a carefully designed in-memory implementation, even without parallelism (since we restricted P-GAM to 1 thread).

Parallel P-GAM. In the following, we omit the time required for cleaning up the data structures after every iteration, as we want to focus on the scalability of the algorithm. Nevertheless, the time for the maintenance of internal data structures takes less than 5% of the query time for large graphs. On the graphs chain_k for $12 \leq k \leq 15$, we report the exhaustive search time (Figure 5b) for query $\{\text{kwd}_0, \text{kwd}_1\}$ as we increase the number of worker threads from 1 to 20. We see a clear speedup as the number of threads increases, which is around 13x for the graph sizes that we report. The speedup is not linear, because as the size of the intermediate results grows, it exceeds the size of the CPU caches, while threads need to access them at every iteration. Our profiling revealed that, as several threads access the shared data structures, they evict content from the CPU cache that would be useful to other threads. Instead, we did not notice overheads from our synchronization mechanisms. Therefore, *we observe that our parallelization approach using concurrent data structures is beneficial for parallel processing, while partitioning-oblivious.*

To study the scalability of the algorithm with the graph size, we use $\text{star}_{4,k}$ for $k \in \{1K, 2K, 3K, 4K, 5K\}$ and the query $\{\text{kwd}_1, \text{kwd}_2, \text{kwd}_3, \text{kwd}_4\}$. Figure 5c shows the exhaustive search time of P-GAM on these graphs of up to 20.000 nodes, using 1 to 4 threads. We obtain an average speed-up of $3.2\times$ with 4 threads, regardless the size of the graph, which shows that P-GAM scales well for different graph models and graph sizes. After profiling, we observed that the size of the intermediate results impacts the performance, similar to the previous case of the chain graph.

In the above $\text{star}_{4,k}$ experiments, we used up to 4 threads since the graph has a symmetry of 4 (however, threads share the work with no knowledge of the graph structure). When keyword matches are poorly connected, e.g., at the end of simple paths, as in our star graphs, P-GAM search starts by exploring these paths, moving farther away from each keyword; if N nodes match query keywords, up to N threads can share this work. In contrast, as soon as these explored paths intersect, Grow and Merge create many opportunities that can be exploited by different threads. On chain_k, the presence of 2 edges between any adjacent nodes multiplies the Grow and Merge opportunities, work which can be shared by many threads. This is why on chain_k, we see scalability up to 20 worker threads, which is the maximum that our server supports.

Data model	$\|E\|$	$\|N\|$	$\|N_P\|$	$\|N_O\|$	$\|N_L\|$
XML	35,318,110	22,204,487	1,561,352	718,434	147,256
JSON	2,800,959	998,013	133,794	147,431	9,822
HTML	232,675	174,849	5,144	4,479	581
Total	38,351,744	23,377,349	1,700,290	870,344	157,659

Table 2: Statistics on Conflict of Interest application graph.

#	Keywords	T^1	T^{last}	T	S	# DS
1	A1, A2	200	4840	4840	1000	1-6, **5**
2	A1, H1	130	615	615	1000	1-7, **7**
3	A3, I1	1263	20547	60000	13	2-4, **2,3**
4	A4, I2	2860	2866	60000	3	2-3, **3**
5	A5, A6, I3	2602	4203	60000	15	6,8, **8**
6	A7, H2, I2	2385	59131	60000	22	5-9, **6**
7	A8, I2, I4	667	51186	60000	63	4-7, **6**
8	A9, H3, I2	264	59831	60000	516	3-8, **5**
9	H2, I1, P1	1267	60212	60000	148	6-8, **6**
10	A5, A10, I2	19077	23160	60000	9	8, **8**
11	A11, I1, I2, P2	4791	54477	60000	9	5,7-8, **8**
12	A9, I1, I4, I5	6327	55762	60000	38	8-9, 11, **8**
13	A7, I1, I6, P1	1857	3057	60000	8	7, 8, **7,8**
14	A12, I1, P2, H3	21031	55221	60000	24	7, **7**
15	A7, A8, I1, I2, I4	3389	28237	60000	4	7-8,11, **11**

Table 3: P-GAM performance on CoI real-world graph.

6.4 P-GAM in Conflict of Interest application

We now describe experiments on actual application data.

The graph. We selected sources based on S. Horel's expertise and suggestions, as follows. (i) We loaded more than **450.000** PubMed bibliographic notices (**XML**), corresponding to articles from 2019 and 2020; they occupy **934 MB** on disk. We used the same extraction policy as in Section 6.2 to perform only the necessary extraction. (ii) We downloaded almost **42,000** PDF articles corresponding to these notices (those that were available in Open Access), transformed them into **JSON** using an extraction script we developed, and preserved only those paragraphs starting with a set of keywords ("Disclosure", "Competing Interest", "Acknowlegments" etc.) which have been shown [1] to encode potentially interesting participation of people (other than authors) and organizations in an article. Together, these JSON fragments occupy **340 MB** on disk. *The JSON and the XML content from the same paper are connected (at least) through the URI of that paper, as shown in Figure 1.* (iii) We crawled 781 **HTML** Web pages from a set of Web sites describing people and organizations previously involved in scientific expertise on sensitive topics (such as tobacco or endocrine disruptors), including: www.desmogblog.com, tobaccotactics.org, www.wikicorporates.org and www.sourcewatch.org. These pages total **24 MB**. Table 2 shows the numbers of edges ($|E|$), of nodes ($|N|$), and of Person, Organization and Location entities ($|N_P|$, $|N_O|$, $|N_L|$), split by the data model, and overall.

Querying the graph. Table 3 shows the results of executing 15 queries, until **1000 solutions** or for at most **1 minute**, using P-GAM. From left to right, the columns show: the query number, the query keywords, the time T^1 until the first solution is found, the time T^{last} until the last solution is found, the total running time T, the number of solutions found, and some statistics on the number of data sources participating in the solutions found ($\#DS$, see below). All times are in milliseconds. We have anonymized the keywords that we use, not to single

out individuals or corporations, and since the queries are selected aiming not at them, but at a large variety of P-GAM behavior. We use the following codes: **A** for author, **H** for hospital, **P** for country, and **I** for industry (company). A $\#DS$ value of the form "2-10, **6**" means that P-GAM found solutions spanning at least 2 and at most 10 data sources, while most solutions spanned over 6 sources.

We make several observations based on the results. The stop conditions were set here based on what we consider as an interactive query response time, and a number of solutions which allow further exploration by the users (e.g., through an interactive GUI we developed). Further, solutions span over several datasets, demonstrating the interest of multi-dataset search enabled, and that P-GAM exploits this possibility. Finally, we report results after performing queries including different numbers of keywords and the system remains responsive within the same time bounds, despite the increasing query complexity.

7 Related Work and Conclusion

In this paper, we presented a complete pipeline for managing heterogeneous data for IJ applications. This innovates upon recent work [3] where we have addressed the problems of integrating such data in a graph and querying it, as follows: (i) we present a complete data science application with clear societal impact, (ii) we show how extraction policies improve the graph construction performance, and (iii) we introduce a parallel search algorithm which scales across different graph models and sizes. Below, we discuss prior work most relevant with respect to the contributions we made here; more elements of comparison can be found in [3].

Our work falls into the *data integration* area [4]; our IJ pipeline starts by ingesting data into an integrated data repository, deployed in PostgreSQL. The first platform we proposed to Le Monde journalists was a mediator [24], resembling polystores, e.g., [7, 25]. However, we found that: (i) their datasets are changing, text-rich and schema-less, (ii) running a set of data stores (plus a mediator) was not feasible for them, (iii) knowledge of a schema or the capacity to devise integration plan was lacking. ConnectionLens' first iteration [26] lifted (iii) by introducing keyword search, but it still kept part of the graph *virtual*, and split keyword queries into subqueries sent to sources. Consolidating the graph in a single store, and the centralized GAM algorithm [3] greatly sped up and simplified the tool, whose performance we again improve here. We share the goal of exploring and connecting data, with *data discovery* methods [10, 27–29], which have mostly focused on tabular data. While our data is heterogeneous, focusing on an IJ application partially eliminates risks of ambiguity, since in our context, one person or organization name typically denote a single concept.

Keyword search has been studied in XML [30, 31], graphs (from where we borrowed Grow and Merge operations for GAM) [32, 33], and in particular RDF graphs [34, 35]. However, our keyword search problem is harder in several aspects: (i) we make no assumption on the shape and regularity of the graph; (ii) we allow answer trees to explore edges in both directions; (ii) we make no assumption on the score function, invalidating Dynamic Programming (DP) methods such as [31] and other similar prunings. In particular, we show in [13] that *edges with a confidence lower than 1*, such as similarity and extraction edges in our graphs, compromise, for any "reasonable" score function which reflects these confidences, the *optimal substructure* property at the core of DP. Works on *parallel keyword search in graphs* either consider a different setting, returning a certain class of subgraphs instead of trees [36] or standard graph traversal algorithms like BFS [37–39]. To the best of our knowledge, GAM is the first keyword search algorithm for the specific problem that we consider in this paper. Accordingly, in this paper we have parallelized GAM, into P-GAM, by drawing inspiration and addressing common challenges raised in graph processing systems in the literature, in particular concerning the CPU efficiency while interacting with the main memory [19–22, 40].

Acknowledgments. The authors thank M. Ferrer and the Décodeurs team (Le Monde) for introducing us, and for many insightful discussions.

References

[1] "European Press Prize: the Monsanto Papers," 2018.

[2] "Offshore Leaks," 2013.

[3] A. G. Anadiotis, O. Balalau, C. Conceição, H. Galhardas, M. Y. Haddad, I. Manolescu, T. Merabti, and J. You, "Graph integration of structured, semistructured and unstructured data for data journalism," *Information Systems*, In Press, 2021.

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

[5] D. Calvanese, G. D. Giacomo, M. Lenzerini, D. Lembo, A. Poggi, and R. Rosati, "MASTRO-I: efficient integration of relational data through DL ontologies," in *DL Workshop*, 2007.

[6] M. Buron, F. Goasdoué, I. Manolescu, and M. Mugnier, "Obi-wan: Ontology-based RDF integration of heterogeneous data," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2933–2936, 2020.

[7] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik, "The BigDAWG polystore system," *SIGMOD*, 2015.

[8] R. Alotaibi, D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis, "Towards scalable hybrid stores: Constraint-based rewriting to the rescue," in *SIGMOD*, 2019.

[9] A. Quamar, J. Straube, and Y. Tian, "Enabling rich queries over heterogeneous data from diverse sources in healthcare," in *CIDR*, 2020.

[10] M. Ota, H. Mueller, J. Freire, and D. Srivastava, "Data-driven domain discovery for structured datasets," *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 953–965, 2020.

[11] C. Christodoulakis, E. Munson, M. Gabel, A. D. Brown, and R. J. Miller, "Pytheas: Pattern-based table discovery in CSV files," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2075–2089, 2020.

[12] F. Nargesian, K. Q. Pu, E. Zhu, B. G. Bashardoost, and R. J. Miller, "Organizing data lakes for navigation," in *SIGMOD*, 2020.

[13] A. G. Anadiotis, M. Y. Haddad, and I. Manolescu, "Graph-based keyword search in heterogeneous data sources," in *Bases de Données Avancés (informal publication)*, 2020.

[14] N. Oreskes and E. Conway, *Merchants of Doubt*. Bloomsbury Publishing, 2012.

[15] S. Horel, *Lobbytomie*. La Découverte, 2018. In French.

[16] S. Horel, "Petites ficelles et grandes manoeuvres de l'industrie du tabac pour réhabiliter la nicotine," 2020. In French.

[17] "Physician Payments Sunshine Act," 2010.

[18] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999.

[19] N. Elyasi, C. Choi, and A. Sivasubramaniam, "Large-scale graph processing on emerging storage devices," in *USENIX FAST*, 2019.

[20] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.

[21] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *SOSP*, 2013.

[22] S. Hong, S. Depner, T. Manhardt, J. V. D. Lugt, M. Verstraaten, and H. Chafi, "PGX.D: a fast distributed graph processing engine," in *SC*, 2015.

[23] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *SIGMOD*, 2009.

[24] 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," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1513–1516, 2016.

[25] 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 Parallel Databases*, vol. 34, no. 4, pp. 463–503, 2016.

[26] C. Chanial, R. Dziri, H. Galhardas, J. Leblay, M. L. Nguyen, and I. Manolescu, "Connectionlens: Finding connections across heterogeneous data sources," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 2030–2033, 2018.

[27] A. D. Sarma, L. Fang, N. Gupta, A. Y. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu, "Finding related tables," in *SIGMOD*, 2012.

[28] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, "Aurum: A data discovery system," in *ICDE*, 2018.

[29] R. C. Fernandez, E. Mansour, A. A. Qahtan, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang, "Seeping semantics: Linking datasets using word embeddings for data discovery," in *ICDE*, 2018.

[30] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: ranked keyword search over XML documents," in *SIGMOD*, 2003.

[31] Z. Liu and Y. Chen, "Identifying meaningful return information for XML keyword search," in *SIGMOD*, 2007.

[32] 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.

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

[34] S. Elbassuoni and R. Blanco, "Keyword search over RDF graphs," in *CIKM*, 2011.

[35] W. Le, F. Li, A. Kementsietsidis, and S. Duan, "Scalable keyword search on large RDF data," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2774–2788, 2014.

[36] Y. Yang, D. Agrawal, H. V. Jagadish, A. K. H. Tung, and S. Wu, "An efficient parallel keyword search engine on knowledge graphs," in *ICDE*, 2019.

[37] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *PACT*, 2011.

[38] L. Dhulipala, G. E. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *SPAA*, 2017.

[39] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers," in *SPAA*, 2010.

[40] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *USENIX ATC*, 2017.