


POLITECNICO DI TORINO – TÉLÉCOM PARIS

Master of Science in Computer Engineering
Diplôme d'Ingénieur

Double Degree Master Thesis

Multi-Threaded Keyword Search in a Memory-Resident Graph Store

Supervisors
prof. Paolo Garza
prof. Antoine Amarilli

Candidate
Chimienti Francesco [265388]

Internship Tutors
prof. Angelos-Christos Anadiotis
prof. Ioana Manolescu

Accademic year 2020-2021

Summary

ConnectionLens is a Java-based project which integrates a set of heterogeneous data sources (e.g. CSV, plain text, RDF, JSON, PDF, etc.) into a single graph [1]. Such a graph is stored into a relational database (i.e. *PostgreSQL*). Furthermore, a keyword search query algorithm has been developed in Java, consulting *ConnectionLens* nodes and edges from the *PostgreSQL* back-end [2].

At the same time, the size of the main memory in modern servers has been growing significantly in the past decade, while data management research has been leading to several database engines running entirely in main memory. Therefore, in the present work, we seek to improve the performance of the query algorithm by:

1. Migrating graphs built by *ConnectionLens* into a new graph store, developed in *C++* and memory-resident;

2. Developing a novel query algorithm that accesses the graph from such an in-memory store and takes advantage of multi-thread parallelism.

In doing so, we pay special attention to in-memory data layout, leveraging storage bandwidth. Furthermore, we design and employ concurrent and sequential native data structures, which are specialized in performing the tasks required by the scenario at hand.

This accompanies the methodical yet flexible design of a multi-threaded query algorithm, which eventually implements effective data partitioning policies in order to even the workload among the threads.

Contents

Chapter 1

Introduction to ConnectionLens

ConnectionLens (hereinafter also *CL*) [1]–[3] is a system developed by the CEDAR research team aiming to answer a class of application needs encountered in the field of data journalism (i.e. journalistic work significantly based on digital data). Specifically, *CL* identifies connections across a set of heterogeneous, independently produced data sources. In this context, journalists face several challenges that arise from the following factors:

1. The data is large (ref. *big data*) and therefore precludes any manual analysis.

2. The data comes from heterogeneous sources, more or less structured (e.g. JSON files, HTML pages, RDF graphs, plain text, etc.).

3. The information must be properly integrated and interconnected.

4. The structure, size, and shape of such data is unknown, favoring keyword-based queries.

5. The set of data sources is highly dynamic, as more information can be added at any moment and combined with the existing one.

6. Data provenance must be preserved since it is important to be able to show where each piece of information in an answer came from, and how the connections were created.

In response to these issues, *CL* is a system capable of:

1. Ingesting data from different kinds of data sources into a graph, which is then stored into a persistent store (i.e. *PostgreSQL*). The graph is then available for future processing without requiring to integrated and load the data again.

2. Answering keyword-based queries over the graph, where an answer is a tree connecting a set of nodes such that one node matches each keyword.

In particular, *CL* can ingest a relational database, files in CSV, text, RDF, JSON, HTML, or XML format, as well as a PDF files, the latter being transformed in JSON and possibly some RDF files and then actually ingested in the graph.

To make this report self-contained, the sequel of this chapter recalls the main functionalities of *ConnectionLens* prior to our work. Specifically, Section 1.1 details the construction of a graph out of multiple heterogeneous data sources. Then, Section 1.2 discusses the main features of graph construction. Finally, Section 1.3 outlines the keyword-based query answering algorithm devised and implemented in prior work on *ConnectionLens* [2].

1.1 Graph Construction

ConnectionLens integrates JSON, XML, RDF, HTML, relational or text data into a graph, as illustrated in Figure 1.1 [4].

First of all, there exists one data source node for each data source, and each source is mapped to graph entities as close to its data model as possible (e.g. XML edges have no labels while internal nodes all have names). Next, *CL* extracts named entities from all text nodes, regardless of the data source they come from, using trained language models. In Figure 1.1, blue, green, and orange nodes denote `ENTITY_ORGANIZATION`, `ENTITY_LOCATION`, and `ENTITY_PERSON` respectively. Each of those entity nodes is connected to the source node it has been extracted from by an extraction edge, which records also the confidence of the extraction (i.e. in the figure, they are depicted as dashed edges). 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 that 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 user-specified threshold, a `SAME_AS` edge is introduced connecting them, labeled with the similarity value (in Figure 1.1, they are depicted in red).

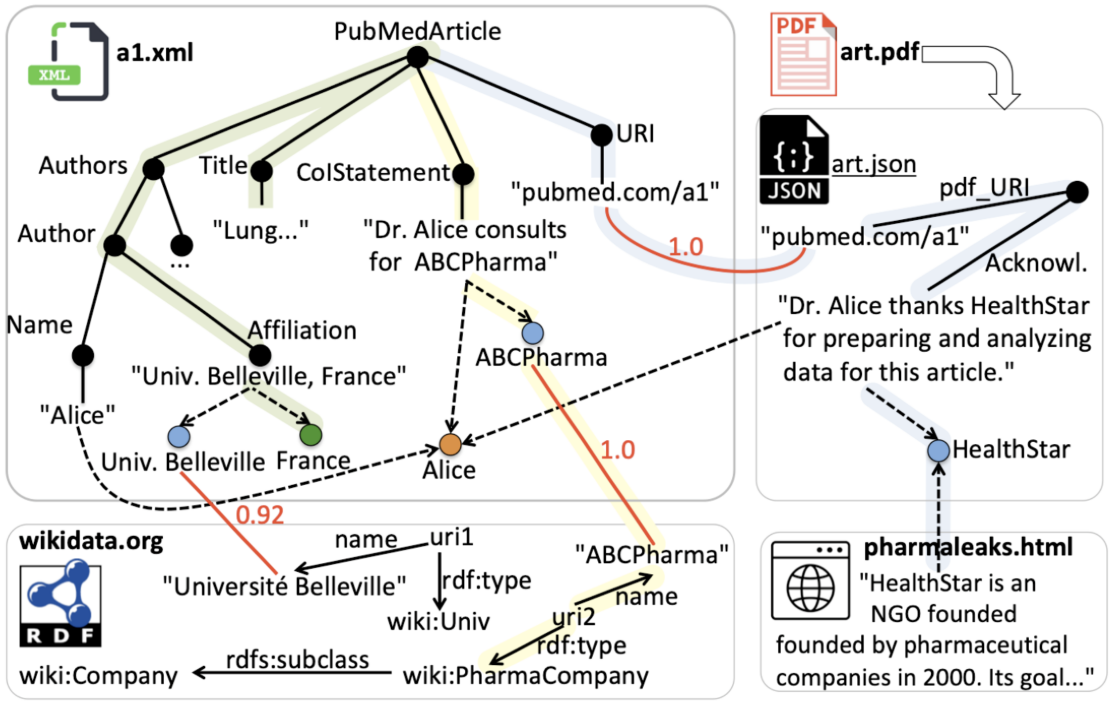


Figure 1.1: Graph data integration in *ConnectionLens*

A `SAME_AS` edge with similarity 1.0 is called an equivalence edge. Then, p equivalent nodes (e.g. the *ABCPharma* entity and the identical-label RDF literal) would lead to $\frac{p(p-1)}{2}$ equivalence edges. To keep the graph compact, one of the p nodes is declared to be the *representative* of all p nodes so that we only store the $p-1$ equivalence edges adjacent to the representative.

We also build a metric called *specificity* on edges, which is used in order to favor edges that are rare for both nodes they connect. 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 $e = n_1 \rightarrow^l n_2$ is defined as $s(e) = 2/(N^l_{n_1\rightarrow} + N^l_{\rightarrow n_2})$. $s(e)$ is 1.0 for edges that are unique for both their source and their target, and decreases when the edge does not stand out among the edges of these two nodes. Such a metric was inspired by experiments on real-world data, as it helped returning interesting answers according to empirical evidence.

Further details on graph construction can be found in [1]. Formally, a *CL* graph is denoted $G = (N, E)$, where nodes can be of different types (e.g. URIs, XML elements, JSON nodes, extracted entities, etc.) and edges encode data source structure, entities extracted from text and node label similarity.

1.2 Relational Graph Storage

The data loading process ends up with a relational database stored in *PostgreSQL*, representing graph entities and auxiliary information. Each database describes a single graph. Therefore, from this point on, the terms *graph* and *database* are employed interchangeably and equivalently.

In the following, we describe the structure of the tables and their attributes that are relevant for future purposes:

- **catalog**. It stores the information related to the data sources that are loaded in a given graph. *Attributes*:
  - `id` - an automatically assigned integer;
  - `type` - the nature of the data (RDF, JSON, XML...);
  - `path` - the path in the file system from where the data has been loaded;
  - `original_uri` - the source of the data (e.g. an external URI).
- **nodes**. It stores the nodes of the graph that have been extracted from the sources. *Attributes*:
  - `id` - the node identifier (an automatically assigned integer);
  - `ds` - the source identifier, i.e. foreign key into **catalog**;
  - `type` - the category of the node which can be: (*a*) the different nodes that we can encounter in a dataset (e.g. XML element, XML literal, RDF URI, RDF literal, JSON map, etc.), (*b*) all the types of nodes that *CL* creates (e.g. `ENTITY_PERSON`, `ENTITY_LOCATION`, `RECORD_ORGANIZATION`, `EMAIL`, etc.);
  - `label` - the node label which can be empty as a structural object like JSON map does not have its own name;
  - `normalabel` - a normalized (i.e. post-processed) label; by default, it is the exact node label; however for some nodes, it differs from the label and *CL* computes the normalized one to ease matching or indexing (e.g. if the label is *Mr. François Ruffin* or *Mr. Ruffin François* and the node is of type `ENTITY_PERSON` the normalized label ends up being "françois ruffin").
- **edges**. It stores the edges that connect the nodes. *Attributes*:

  - id - the edge identifier (an automatically assigned integer);
  - ds - if both nodes are from the same data source, the data source of the edge; otherwise, one of the two data sources (not very meaningful);
  - type - the category of the edge, edges that come from the data are just of type EDGE, but we can also have EXTRACTED_* edges (that connect an entity with the parent text it has been extracted from) and SAME_AS edges;
  - source and target - the two nodes connected by the edge; if the edge is oriented, then we keep the orientation; but we will see that the query algorithm doesn't care about the orientation of the edges;
  - label - the edge label;
  - confidence - the edge confidence, which is 1.0 for parent-child edges and in general for all edges that come from the data; <1 for SAME_AS edges and for extraction edges.

- **weak_same_as**. It stores similarity edges strictly smaller than 1 and greater than a user-specified threshold; its schema is equivalent to **edges**'.
- **specificities**. It stores the specificities of the nodes, as the number of input and output edges.
- **edge_specificity**. It actually materializes the specificity of the edges, as described in Section 1.1.

1.3 The GAM (Grow and Aggressive Merge) Keyword Search Algorithm

Given a graph G and a set of keywords $\{k_1, k_2, \ldots, k_m\}$, the algorithm aims at identifying the set of all *minimal* trees connecting nodes matching each keyword [2]. Here, by *minimal* we mean that no node or edge can be removed from the tree while still keeping both one leaf matching each keyword and a single connected tree.

In practice, since the search space is huge, the algorithm runs until a timeout or a given number of solutions have been found (whichever happens first). An important point to note is that, from a semantic point of view, the root of a tree does not bring any information. Therefore, from the user perspective, each solution is just a set of edges.

1.3.1 Algorithm Outline

The main algorithmic steps are:

1. **`Init`**. With the help of an index, the nodes matching each keyword are identified. Each of them becomes a *1-node tree*.

2. **Build the first `Grow/Grow2Rep` opportunities**. Starting from such *1-node trees*, `Grow/Grow2Rep` opportunities are defined as $(tree, edge)$ pairs, with *edge* being every edge adjacent to *tree*'s root or the edge connecting *tree*'s root to its representative node.

3. **`Grow/Grow2Rep`**. Given a `Grow/Grow2Rep` opportunity, a new tree is created by "growing" the previous tree by one edge (the tree is said to be *growing at the root*). `Grow/Grow2Rep` opportunities are built from the new tree.

4. **`Merge`**. Two trees having the same root are merged into a new tree having the same root and all the edges of the input trees. `Grow/Grow2Rep` opportunities are built from the new tree.

`Init`, `Grow/Grow2Rep` (also simply `Grow`), and `Merge` are the three steps that generate new trees, each of which may or may not be a solution. A tree that is not a solution may or may not have been encountered before. To keep track of the trees generated throughout the search, specific data structures are employed.

1.3.2 Exploration Order

`Grow/Grow2Rep` opportunities are pushed in a priority queue whose sorting criteria are defined based on what is meant by *best solution*. Therefore, given $pair_1 = (tree_1, edge_1)$ and $pair_2 = (tree_2, edge_2)$ and supposing that pulling from the priority queue gives the highest ranked pair:

1. $pair_1 > pair_2$ if $tree_1$ matches more keywords than $tree_2$ going toward complete answers;

2. $pair_1 > pair_2$ if $tree_1$ and $tree_2$ match the same number of keywords but $tree_1$ is smaller than $tree_2$; this ensures that we do not miss smaller answers, which users may find more intuitive;

3. $pair_1 > pair_2$ if $tree_1$ and $tree_2$ are equally big and match the same number of keywords but $edge_1$'s specificity is higher than that of $edge_2$.

It is worth noting that if none of the above criteria for determining a priority order between two $(tree, edge)$ pair allows discriminating two pairs, the tie will be broken in an implementation-dependent way. This includes a (small, but real) opportunity for non-determinism.

1.3.3 Preconditions for Creation of `Grow` Opportunities

Any $(tree, edge)$ pair that gets out of the priority queue will be used to `Grow` a new tree. Therefore, the decision of whether to consider a certain `Grow` move or not is taken when we feed the priority queue. Of course, a necessary condition for an opportunity to exist is *edge* being adjacent to *tree*'s root.

Moreover, a $(tree, edge)$ pair is not pushed onto the queue if:

1. *edge* is a node loop, that is $edge.source = edge.target$;
2. *edge* connects a pair of nodes that already belong to *tree*;
3. *edge* closes a dataset loop, i.e. it connects a node equal to *tree*'s root another one lying in another dataset that already belongs to datasets explored by *tree*';
4. *edge* connects *tree*'s root to its representative, but *tree* already grew by `Grow2Rep`;
5. *edge* brings redundancy, i.e. $edge.source$ and $edge.target$ match the same keyword(s) but $edge.target$ is not the representative for $edge.source$.

Furthermore, we keep track of the history of all pairs inserted in the priority queue since the beginning of the query: we don't push a `Grow` opportunity in a priority queue if it has been already seen and exploited.

1.3.4 Preconditions to `Merge`

The decision whether to consider a certain `Merge` move on $tree_1$ and $tree_2$ is made as follows:

1. $tree_1.root = tree_2.root$;
2. $tree_1.keywords \cap tree_2.keywords = \emptyset$ (possibly except for the root of the trees);
3. $tree_1.nodes \cap tree_2.nodes = \emptyset$.

To find `Merge` candidate efficiently, we index the (partial) answer trees by their root, since it appears to be the most restrictive condition. In this way, whenever we have to find candidates to `Merge` for a given *tree*, we firstly look for other trees rooted in *tree*'s root and then check for the other conditions to be met on each of these trees.

1.3.5 Solution Score

Given an answer *tree* to a query $Q = \{k_1, k_2, ..., k_m\}$, we need to evaluate the quality of the solution by means of a *score*. In this regard, we employ a weighted sum of a $score_{marching}$ and a $score_{connection}$ so that $0 < score < 1$. In details:

- $score_{matching}$ states how much *tree* is semantically relevant as an answer to Q. It is the average, over $\{k_1, k_2, ..., k_m\}$, of the similarity between the *tree*'s node matching a given keyword and the keyword itself. Here, we employed the *Levenshtein distance*.
- $score_{connection}$ aims at discouraging trees with low-confidence edges and/or low-specificity edges. It is the sum of the product of *tree.edges*' specificity and the product of *tree.edges*' confidence.

Chapter 2

In-Memory Data Layout

The size of the main memory in modern servers has grown significantly over the past decade. For instance, *AWS EC2* offers nodes providing up to 24TB of main memory and 448 hardware threads [5]. Data management research has by now led to several mature products (i.e. DB engines) running entirely in main memory, such as *Oracle Database In-Memory*, *SAP HANA*, and *Microsoft Hekaton*. Moving the data from the hard disk to the main memory significantly boosts performance, 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: the memory access [6].

In the following, we will move from *ConnectionLens* to *in-memory ConnectionLens* (hereinafter *CLMem*), a novel in-memory graph database, which we have built and optimized to execute a parallel implementation of GAM search algorithm (hereinafter P-GAM), as described in Chapter 3. To do so, data need to be migrated from one system to the other. As we use the permanent storage as a buffer between the two, we split the discussion into two parts: Section 2.1 explains how data is read from *PostgreSQL* and stored into binary files; Section 2.2 starts from binary files deserialization and brings data into memory realizing a specific layout.

2.1 Moving Data from *ConnectionLens* to Disk

Suppose that we have built a *PostgreSQL* relational database on top of several heterogeneous data sources by running *CL*'s engine, as explained in Section 1.1. Our purpose is to move information out of such database into

CLMem's data structures; for this purpose, we will use the permanent storage (i.e. disk) as a buffer. Therefore, we will store on disk several files, each of them representing a partition of the overall graph. The main steps of the migration pipeline are illustrated in Figure 2.1.

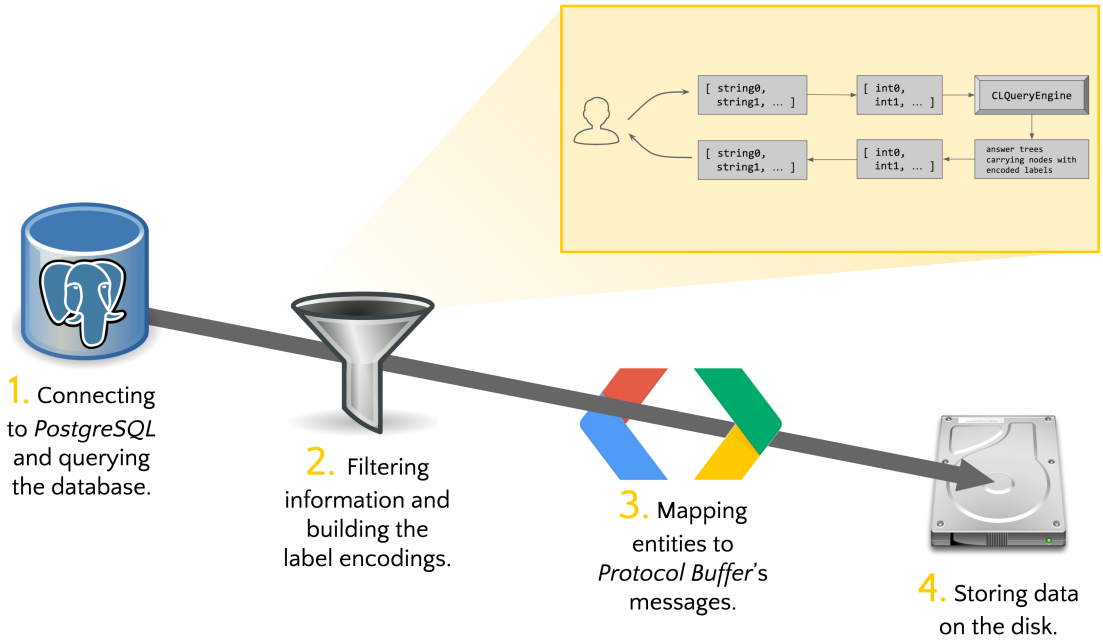


Figure 2.1: Migration pipeline

In detail, the data migration task is carried out by a *Python* script (hereinafter *CLMig*). In the very first place, it connects by means of *psycopg* [7] to a given *PostgreSQL* database, whose name is specified on the command-line interface along with several other connection parameters.

Therefore, 3 *cursors* are instantiated in order to retrieve the information we need from the database, whose details are outlined in Section 1.2. These cursors are:

1. `curCatalog`, which queries **catalog** table to retrieve information about the sources ingested by *CL* — the attributes we keep are in order `(id, type, path, original_uri)`;

2. `curEdges`, which queries a temporary view putting together information coming from **edges** table (which is joined with **edge_specificity** to retrieve edges' specificity) and **weak_same_as** table — the attributes we keep are in order `(id, type, source, target, label, confidence, specificity)`;

3. `curNodes`, which queries **nodes** table and computes the degree of its entries by means of `id = source OR id = target` join condition with the temporary view previously created — the attributes we keep are in order `(id, ds, type, normalabel, representative, num_connections)`, with `num_connections` being the degree of the node.

It's important to highlight that an implicit filtering operation is executed on the data so that we keep only the attributes that we need in *CLMem* in order to implement GAM search and build final query results. Such attributes can be also found in *Protocol Buffers*' messages definition, which is the method we employ to serialize structured data [8]. Pre-compiled headers are going to be produced both for *Python* and *C++* before starting the migration process.

It is worth mentioning the migration pipeline is also in charge of building integer encodings for the node labels. In fact, working with strings and comparing them would end up being a major bottleneck located at the core of the query algorithm. Practically, a query is asked in the form of keywords (i.e. strings), which are meant to be encoded so that the query engine can efficiently work with integers. Eventually, before returning answers to the user (e.g. on the command-line interface), node labels are decoded into strings again. Such a conversion is ruled by a dictionary that is built while migrating the data to the disk and serialized into a separate file `labels.bin`. Analogously, the information related to **catalog** table is going to be serialized in `catalog.bin`.

As far as the rest of the graph is concerned, partitioning policies represent the core of *CLMig* as they describe how nodes are edges are split across different partition files. Therefore, a specific partitioner is instantiated according to a parameter passed on the command-line interface. Currently, the only implemented partitioning policy just splits nodes on one side and edges on the other, for an equal amount of files.

2.2 Moving Data from Disk to *CLMem*

Once a graph has been serialized on disk, we can finally load it inside *CLMem*. Therefore, from now on, we will leave everything related to *CL* and *CLMig* behind and solely describe *CLMem*, entirely implemented in *C++*.

The loading routine takes care of building the graph starting from the binaries, while implementing the memory configuration shown in Figure 2.2.

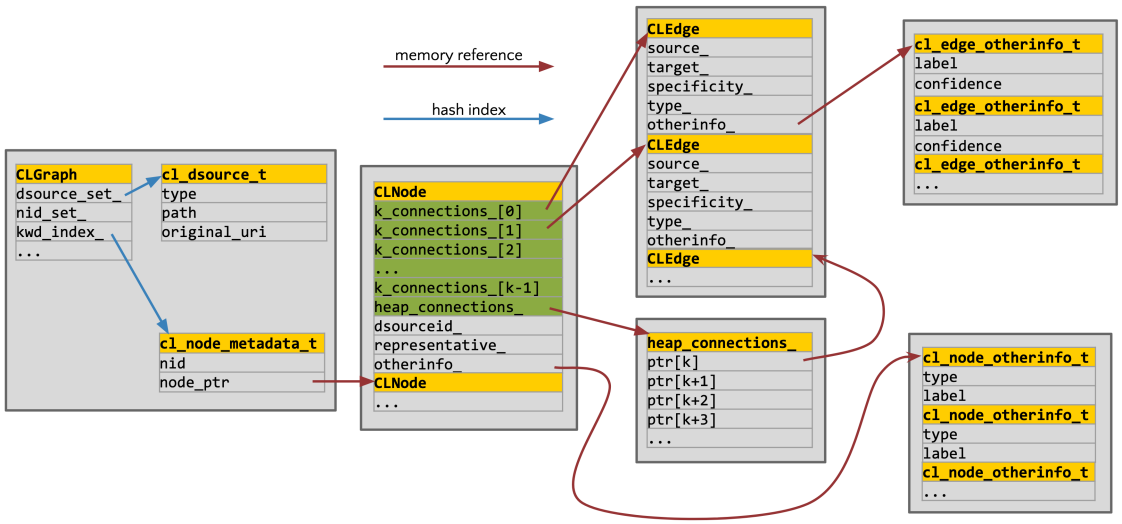


Figure 2.2: *CLMem*'s in-memory data layout

Our design includes all the data needed by applications as described in Section 2.1, while also aiming at high performance, parallel query execution in modern scale-up servers, in order to tackle huge search spaces. In fact, the physical layout of a graph database is important, given that graph processing is known to suffer from random memory accesses [9]–[12]. Therefore, spatial locality of reference plays a crucial role as we want to limit *TLB misses* as much as possible and, at the same time, exploiting caching and pre-fetching mechanisms. Exploratory graph queries are almost unpredictable so making efficient use of memory and cache is an object of research. In our memory layout, we adopt a conservative approach and we split the data required for the search, from the rest, as the former is critical for performance; in the following, we will also refer to the latter as *metadata*.

Before describing the data layout, we need to mention that memory allocation is entirely managed by a memory pool. As extensively outlined in Appendix A, it allows us to acquire *page groups*, which are arrays of memory pages for which contiguous allocation is granted and whose size matches the one of *L1 cache*. In Figure 2.2 the grey boxes are meant to be intended as page groups.

So, starting to focus on `CLNode` objects, we may notice that they are allocated one after the other inside a page group, as much as they can fit. As soon as the last page of the group is full, we just request another page group to the memory pool. As a matter of fact, we first compute the number of page groups we need for all the nodes inside a given graph partition, and

then we request them to the memory pool once for all.

A `CLNode` object includes the identifier of the data source where the node comes from, a reference to the representative, and a reference to node metadata. Such metadata 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. Furthermore, we need an array of references to `CLEdge` objects connected to each node, which should be then allocated contiguously in one or more page groups. However, following several experiments, we found that the nodes' degree can be high enough to spread such an array across one or more page groups, thus shattering the `CLNode` object itself. For this reason, we keep k references inside the object and we define k such that

$$k \cdot \texttt{sizeof(CLPtr<CLEdge>)} + \texttt{sizeof(CLNode)} < \texttt{page\_group\_sz} \quad (2.1)$$

In this way, we know that is always possible to instantiate a `CLNode` object in a page group. Then, we store the rest of the references to the edges into another memory segment (i.e. a separate heap area) so that we can still preserve locality. This logic is implemented by means of a dedicated class called `CLImmutableArrayHeap_TS<T>`, which is described in Appendix B.2.5 and represented by the fields depicted in green in Figure 2.2. In general, other criteria might play a role in choosing a value for k, e.g. the average degree of the graph nodes. Further improvements might include a smarter policy for splitting the edges according to some criteria (e.g. confidence, specificity).

Then, a `CLEdge` object includes a reference to the source and the target node of every edge, the edge specificity and type, and a reference to the edge metadata, which includes the confidence and the label of each edge.

Finally, we build a keyword index `kwd_index_` stored in `CLGraph` object, i.e. hash-based map associating every label with the list matching nodes. P-GAM probes `kwd_index_` when a query arrives to find the references to the nodes that match the query keywords and start the search from there.

We conclude the present discussion by saying that the above storage is row-oriented (i.e. *node-oriented*), even though column-oriented 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 fully ad-hoc queries (i.e. 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 optimally exploited by column-stores, do not apply. Instead, the crucial optimization here is to find the neighbors of every node fast: in fact, this is leveraged by our algorithm.

Chapter 3

P-GAM Algorithm

This chapter describes the main technical contributions of the present work. The stated goal is a parallel, in-memory variant of the GAM algorithm, leveraging the in-memory graph representation described in Section 2.2. Below, we start with a discussion that was not initially planned but which turned out to be necessary: an analysis of the role that a *pruning* strategy, not described in previous GAM write-ups [2], [3], plays with respect to the algorithm completeness (Section 3.1). We then describe the design choices made for our P-GAM algorithm (Section 3.2), detail its data structures (Section 3.3), and finally outline the algorithm itself (Section 3.4).

3.1 GAM Completeness Analysis

Prior writings on GAM algorithm did not include a formal analysis of its completeness [2], [3]. In fact, since GAM's moves (i.e. `Grow` and `Merge`) are inspired from those of prior work [13], the completeness claim made in such a context was tacitly assumed to carry to GAM and it is straightforward to prove.

However, GAM *as implemented* contained an additional pruning technique, as follows:

Definition 3.1.1 (Edge-set pruning (ESP))**.** During GAM search process, when a tree t_1 consisting of the edges $E_t = \{e_1, e_2, \ldots, e_n\}$ $(n \geq 1)$ is created such that another tree t_0 with the same edge set E_t had been created previously, the *edge set pruning* (ESP, in short) consists of discarding (ignoring) t_1 for the rest of the search.

While testing our parallel implementation of GAM (described below in

Section 3.4), its behavior on some inputs required further investigation which established that solutions were sometimes missed even if the algorithm was executed in single-threaded mode. Testing the original GAM implementation on the same inputs as in *CL* turned out to pose the same problem! This required us to study the *completeness of GAM with ESP* in further detail, whose findings are described below. The following analysis naturally also carries to our parallel algorithm since both GAM and P-GAM follow the same search steps.

We first introduce some terminology. All the definitions below apply within one solution (i.e. *answer tree*) for a query on a *ConnectionLens* graph.

Definition 3.1.2 (Leaf)**.** A leaf is a node connected to only one edge.

The three definitions below enable us to refer to trees at three different levels of precision. They are important since they enable us to make specific statements in our discussion of completeness:

Definition 3.1.3 (Edge set)**.** An edge set is a set of edges that, together, form a tree and such that at most 1 leaf does not match a query keyword.

Definition 3.1.4 (Rooted tree)**.** A rooted tree is an edge set together with one distinguished node present in these edges, called the root.

Observe that $(k + 1)$ potential rooted trees correspond to each edge set of k edges; one such tree is rooted in each node. We say *potential* because in general, there is no guarantee that all these rooted trees will actually be built.

Definition 3.1.5 (Provenance)**.** A provenance is a formula of the forms described below, together with a tree node called the *root* of the provenance:

1. `Init(`n`)` where n is a node matching a keyword; the root of such a provenance is n itself;
2. `Grow(`t, e`)` where t is a provenance, its root is n_0, e is an edge going from n_0 to n_1 and n_1 does not appear in t; in this case, n_1 is the root of the `Grow` provenance;
3. `Grow2Rep(`t, e`)`, where t is a provenance rooted in n_0, e is an equivalence edge going from n_0 to n_1, n_1 is the representative of n_0; in this case, n_1 is the root of the `Grow2Rep` provenance;
4. `Merge(`t_1, t_2`)`, where t_1 and t_2 are provenances, rooted in $n_1 = n_2$; in this case, $n_1 = n_2$ is the root of the `Merge` provenance.

Observe that there may be several provenances for the same rooted tree. In this regard, a provenance is more specific than a rooted tree (the provenance not only describes the tree but also enforces one specific way to build it); similarly, a rooted tree is a more specific notion than an edge set. Conversely: a given edge set might come from several different rooted trees, each of them being the result of several provenances.

Intuitively, the goal of ESP is to limit the construction of provenances that correspond to the same edge set, given that a solution is an edge set. Thus, ideally, if we built just one provenance for each edge set (while still guaranteeing that one such provenance is built for each solution), this could suffice to ensure search completeness.

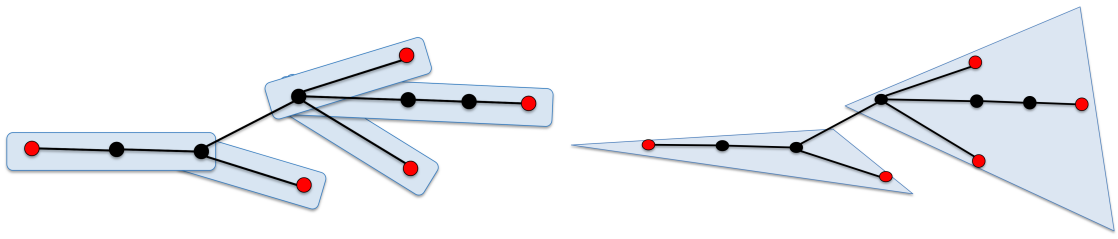
Based on these notions, below, we start by showing the completeness of GAM with (despite) ESP for some situations only, before exhibiting cases where this completeness is not guaranteed.

3.1.1 Restricted Completeness Guarantees for GAM with ESP

In this section, we establish the completeness of results for the GAM algorithm with ESP.

Property 1 (Restricted completeness). Assume we are given a graph G, a query Q, and a solution (answer tree for Q on G) denoted t, such that **no keyword is matched in an internal node of** t. Then, the solution t is guaranteed to be found by GAM with ESP.

We show this below. Recall that the solution t, as per the definition of solutions in Section 1.3, is an edge set (its root is irrelevant).



(a) Rooted paths (b) Maximal merges

Figure 3.1: Illustrations of concepts introduced for the restricted completeness discussion.

Definition 3.1.6 (Rooted path)**.** Within a solution t as above, a *rooted path* is a path that goes from a leaf (which is guaranteed to match a keyword) to its nearest node that has at least 3 edges.
Observe that our restricted assumptions imply that that the root of a rooted path does not match any keyword.

Rooted paths can be visualized in Figure 3.1a as light blue rectangles.

Lemma 3.1.1. *Any rooted path in a solution t as above is guaranteed to be built by (and not pruned) by GAM with ESP.*

Proof. We prove this in a constructive way, by exhibiting a provenance that is guaranteed to be built (and not pruned) by GAM with ESP. A guarantee on the provenance directly ensures a guarantee that the corresponding rooted tree is built by GAM and not pruned by ESP, which in turn leads to a similar guarantee for the corresponding edge set.

1. For each keyword match in t, the 1-node `Init` provenance corresponding to this node is guaranteed to be built and pushed in the priority queue. ESP pruning does not apply.

2. Any provenance applying only successive `Grow` steps on top of such an `Init` provenance, is guaranteed to be built by GAM (given how the algorithm works). Further, such a provenance is not pruned by ESP, because it is the only provenance for its edge set. Thus, the ESP condition does not apply.

□

Next, we introduce:

Definition 3.1.7 (Maximal merge)**.** Within a solution t as described above, a *maximal merge* is a rooted tree merging all the rooted paths having a common root node.

Maximal merges are illustrated in Figure 3.1b, for the same solution as the one in Figure 3.1a.

Lemma 3.1.2. *Any maximal merge of rooted paths appearing in a solution t as above is guaranteed to be built by GAM and not pruned by ESP.*

Proof. Lemma 3.1.1 entails that any rooted path is built. This, along with the aggressive application of `Merge` within the GAM algorithm, concludes the proof. □

Lemma 3.1.2 can be also stated as: *one among all the provenances corresponding to a maximal merge is guaranteed to survive ESP pruning.* It is easy to see that this will be the first maximal merge developed for the specific edge set: based on it, ESP eliminates all the other maximal merges subsequently developed for the same edge set.

Lemma 3.1.3. *Within a solution t as above, any rooted tree that can be built by a sequence of* `Grow` *on top of a maximal merge, and whose root is not a leaf of t, is guaranteed to be built by GAM and not pruned by ESP.*

Proof. Lemma 3.1.2 entails that any merge of rooted paths is built (as a rooted tree). This means one of its provenances is built. By the GAM algorithm design, all `Grow` steps on top of this provenance will be tried. Let t_g be one such provenance.

- By the assumption made on t (the solution of which t_g is a subtree) and given that the root of t_g is not a leaf, this root does not match any query keyword. Thus, t_g is the only provenance for its rooted tree, since it is the only way to build with t_g's leaves and not rooted in a keyword. Therefore, t_g is not pruned by ESP.

- Given that each such t_g is not pruned by ESP, all `Grow` steps will be attempted on it, and, by similar reasoning, not pruned by ESP.

□

Based on the above Lemmas, we can finalize the proof of Property 1, that is: establish that a restricted solution such as t is guaranteed to be built by GAM and not pruned by ESP. We rely on the following observations:

1. If Q has just 1 keyword, the property is trivially proved.

2. If Q has two keywords, any solution is a merge of two rooted paths and is guaranteed to be built by GAM and not pruned by ESP, by Lemma 3.1.1.

3. If Q has 3 or more keywords:

   (a) t has that number of leaves (at least 3) and thus t contains one or more maximal merge trees, each of which are guaranteed to be found (by Lemma 3.1.2). Figure 3.2 shows such a solution, highlighting the maximal merge trees as light blue triangles.

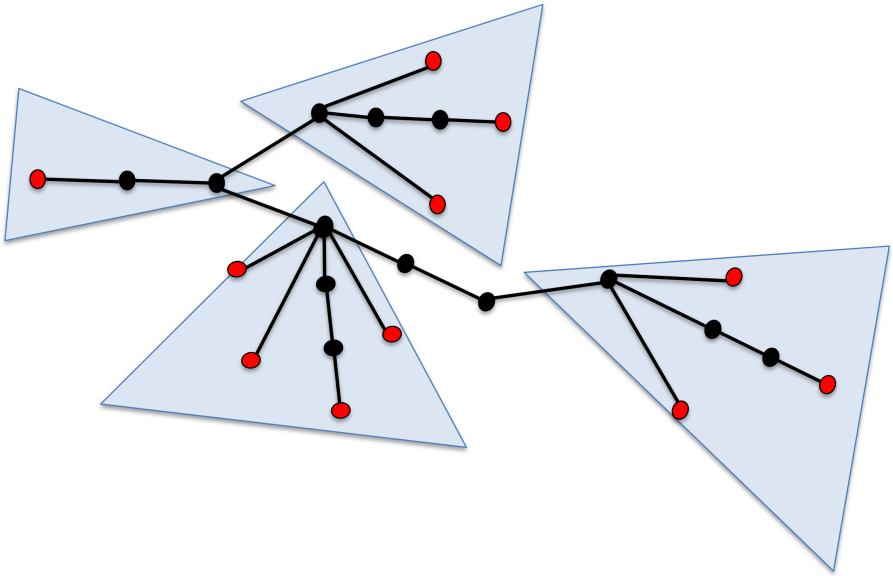


Figure 3.2: Sample solution satisfying the restriction of Property 1.

(b) Since t is a connected tree, `Grow` paths on top of the maximal merges of t are bound to intersect each other, which leads to one or more `Merge` provenances being built from them.

(c) There exists at least one such provenance that covers exactly t.

(d) If there are several such provenances, ESP will prune all but the first one encountered; this still guarantees that the solution (edge set) t is found.

Further, one can easily show:

Property 2 (GAM with ESP completeness for 3-keyword queries). Let G be a graph, Q be a query of 3 keywords. Any solution of Q on G is guaranteed to be found by GAM and not pruned by ESP.

Let t be a solution for Q on G. There are two cases:

1. If in t no keyword is matched on a path between the two other keyword matches, Property 1 ensures t is found.

2. On the contrary, assume (without loss of generality) that the query keyword kwd_2 is matched on the path between the t nodes matching the keywords kwd_1 and kwd_3. GAM builds paths starting from the three nodes, and as long as their roots do not match any keywords, they are not pruned by ESP (reasoning similar to Lemma 3.1.1). Further, these

paths have to intersect (`Merge`) into a tree matching 2 keywords, which necessarily will encounter (and `Merge` with), immediately or after some `Grow` steps, the third path. Thus, such solutions are also guaranteed to be found.

3.1.2 Incompleteness of GAM with ESP

The solutions left out by Property 1 and Property 2 may be missed by GAM with ESP. Formally:

Property 3 (ESP incompleteness for at least 4 keywords). There exists a graph G, a query Q of four keywords, and a solution such that ESP may prevent GAM from finding the solution.

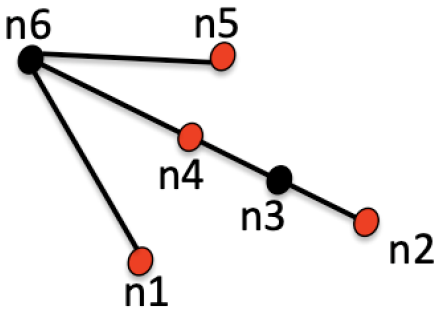


Figure 3.3: Example where GAM with ESP could be incomplete

Figure 3.3 exhibits such an example. The nodes in red (n_1, n_2, n_4 and n_5) match four query keywords. Depending on the priority used in its queue, and also on how priority ties are broken (this includes some non-determinism), GAM search with ESP *may* take the following course:

1. Search starts from all the node matching keywords.

2. The first tree tree connecting n_2, n_3 and n_4 is $t^3_{2,3,4}$, found by `Merge` on the root n_3. From now on, ESP discards any other tree with the same edges.

3. No `Grow` can apply on $t^3_{2,3,4}$ because it already contains all the edges adjacent to its root. No `Merge` can apply on it either, because `Merge` only applies on trees whose nodes are disjoint except for the root, and no such tree rooted in n_3 exists.

4. Separately, the paths grown from n_1, n_4 and n_5 intersect in n_6 and lead to a tree $t^6_{1,4,5,6}$ rooted there. We assume this is the first tree with this set of edges; any future tree with the same edges will be pruned by ESP.

5. At this point, the search is "stuck" as there is no way to connect the upper left subtree $t^6_{1,4,5,6}$ with the lower right one $t_{2,3,4}$, given that they do not have the same root. Further, none among these trees can `Grow` towards the other, because each already contains all the edges adjacent to the root. It is easy to see that building on the other subtrees, developed prior to $t^3_{2,3,4}$ and $t^6_{1,4,5,6}$, does not allow to find the solution, either.

Thus, in the above exploration sequence, the only solution will be missed.

Observe that the search space could be traversed in a different, more favorable order, for instance:

1. Build the tree $t^6_{1,5,6}$ rooted in n_6, then, based on it, $t^4_{1,4,5,6}$ rooted in n_4;

2. Build the tree $t^4_{2,3,4}$ rooted in n_4;

3. Merge these into a provenance for the solution.

The exact exploration sequence depends on the properties of the G edges involved, which are inputs to the priority function used attached to the queue used by GAM. Thus, in general, one cannot be sure which exploration sequence will be taken.

Thus, we conclude that:

- GAM with ESP is complete for at most 3 keywords (Property 2); note that these are the most frequently encountered;

- GAM with ESP is potentially incomplete for 4 or more keywords and should be used only if execution speed is favored over completeness guarantees.

3.2 Algorithm Design

In the following, we start designing a parallel implementation of GAM search algorithm by adopting a systematic approach based on a permeable partition of the design process as suggested in [14]. However, we do not rigidly follow the scheme and instead try to adapt the guidelines to the situation at hand and its peculiarities.

3.2.1 Partitioning

First of all, the computational task that has to be performed and/or the data operated on by such a task need to be decomposed into smaller chunks. At this stage, practical issues (e.g. the number of processors in the target computer) are going to be ignored, while the focus is kept on identifying opportunities for parallel execution.

We choose to adopt a *data-driven* approach, namely *domain decomposition*: we start by trying to get a data decomposition as fine-grained as possible; then, based on the result of such a process, we partition the computation that is to be performed on these data. Therefore, the most fine-grained data decomposition corresponds to bootstrapping GAM with a thread per priority queue and a priority queue per `Grow`/`Grow2Rep` opportunity generated by the *1-node trees*. At this point, let's suppose that each of these threads might execute GAM independently from each other, working on their own local structures, namely (besides the priority queue, hereinafter `pQueue`):

1. **Memory of all the explored trees** ⟦`memoryTrees`⟧. It holds the history of all trees built during the exploration of the graph. Such a data structure is crucial in that it precludes further investigation of trees that have already been observed, allowing to move computational resources to essential and meaningful tasks.

2. **Memory of all the pairs inserted in the priority queue** ⟦`memoryPairs`⟧. It holds the history of all the `Grow`/`Grow2Rep` opportunities generated during the exploration. As `memoryTrees`, it serves optimization purposes.

3. **Root → trees rooted in that node map** ⟦`treesByRoot`⟧. It gives access to all trees rooted in a certain node. Since trees are merged only at the root, such a map is essentially an index on `memoryTrees`, returning the `Merge` candidates for a given tree.

4. **List of query answers** ⟦`queryAnswers`⟧. It is a collection of final answer trees.

At its own pace, each thread will end up exhaustively exploring the search space: therefore the correctness of the results and the convergence towards the complete solution space is guaranteed, as shown in Section 3.1. On the other hand, only drawbacks come from adopting such a naive parallel design, as performances will be worse than single-threaded GAM's. We might need

proper communication channels and protocols in order to synchronize the activity and the tasks among the threads.

3.2.2 Communication Requirements

We identify the following communication requirements for the current scenario:

1. *Global*, since each task requires to communicate with all the others concurrent tasks;

2. *Structured*, as tasks' communication partners form a regular pattern (such as a tree or a grid) and do not change over time;

3. *Static*, because the identity of communication partners does not change over time;

4. *Asynchronous*, since producers are not able to determine when consumers may require data, hence, consumers must explicitly request data from producers.

Giving a closer look at possible implementations, we can deal with communication in two different ways: (*i*) *shared data structures*, with the help of locks and condition variables mechanism; (*ii*) *messages*, so that each thread can keep on working on its own structures solely, at its own pace.

Both of these solutions have their own computational costs and complexity. Furthermore, it should be noted that communication has not to be real-time, since it only serves to speed up the convergence towards the solution space. However, it is not straightforward to determine a proper rate so that overall communication costs end up being lower than the benefits.

On the other hand, as we have outlined in Section 3.2.1, GAM itself is heavily based on certain collections for both optimization (e.g. `memoryTrees`) and strictly functional purposes (e.g. `treesByRoot`). Therefore, the use of shared data structure as communication channels seems to be the natural choice. Moreover, by doing so, we leave room for optimization towards a scalable parallel algorithm, since we can improve the design of the data structures and extremely specialize it in order to fit specific needs. Therefore, we choose to implement communication channels by means of shared data structures.

3.3 Thread-Safe Data Structures Design

In order to move forward with the design of the parallel algorithm, it is imperative to predispose shared and centralized data structures to concurrent use, thus making them thread-safe. To this end, we discouraged the use of data structures offered by the *Standard Template Library* then protected by high-level locks, since they have been deemed particularly inefficient for the current purposes that require maximum performance. Therefore, through the use of *POSIX-native* synchronization mechanisms, we preferred to design ad-hoc data structures, highly specialized in performing the operations required by the current operational scenario, exploiting its limits and peculiarities.

The implementation details behind the following data structures are extensively described in Appendix B.2. In the following, we only refer to the classes implementing each object and relate them to the requirements dictated by the working scenario.

Specifically, the following data structures have been implemented:

1. **`memoryTrees`**. It is implemented by `CLSet_TS<T, Hash, KeyEqual>` (ref. Appendix B.2.4), which is a hash set involving only one operation: insertion with a simultaneous check for any duplicates. In this regard, we want any thread that is running GAM to know with total confidence whether a tree has already been discovered and explored at any given time. Likewise, we don't want read requests resulting from duplicate checking to block concurrent writes. Hence, a trade-off has been found to provide these guarantees without affecting the scalability of the data structure.

2. **`queryAnswers`**. It is an object of class `CLList_TS<T>`, which is outlined in Appendix B.2.2 as a list-type class implemented like a relocatable array, involving insertion and random access operator. However, `CLList_TS<T>` is also employed to store `Merge` candidates, implying that whenever we try to `Merge` we should iterate on the list of interest. To this end, following approaches similar to *copy-on-write* techniques, the class allows us to obtain a local, independent copy of the list holding `Merge` candidates. In general, we might assert that the higher the average number of `Merge` candidates the higher the performance improvement resulting from such an approach.

3. **`treesByRoot`**. It is implemented by `CLUnorderedMultimap_TS<K, V, Hash, KeyEqual>` (ref. Appendix B.2.6), which is a multimap with a

hashed index built on the keys. We expect both the keys (i.e. the nodes) and the values (i.e. the trees) to be unique. However, any duplicate is inherently granted to be avoided for the values, since tress are only inserted in this map after verifying that they have not been found before (i.e. checking is delegated to `memoryTrees`).

4. `memoryPairs`. Since the operations demanded to the data structure are the same as the ones requested to `memoryTrees`, we can use the same design principles without any problems also in this setting. However, in order to better distribute the high workload, we add another level of indexing, thus ending up with a double-indexed hash set. Practically, this is easily accomplished by creating an array of `CLSet_TS<T, Hash, KeyEqual>` (ref. Appendix B.2.4).

3.4 P-GAM Algorithm Outline

We split the discussion into two parts. In the first part (Section 3.4.1), we sketch a first version of the algorithm according to the design principles outlined in Section 3.2. The second part (Section 3.4.2) seeks to release the constraints on strict initial data partitioning, to evenly distribute the workload among threads.

3.4.1 Naive Algorithm

The algorithm is launched by a bootstrap routine (outlined in Algorithm 1) which primarily sets up all the data structures, both local and shared. Therefore, it initializes or cleans up the memory of all the explored trees, the list of query answers, and the memory of all the pairs inserted in the priority queues. Regarding the latter, it should be pointed out that it will only be mentioned here and then set aside, in order to exhibit a simpler overview of the algorithm. However, it is straightforward to understand where `memoryPairs` might come into play, that is: before inserting any $(tree, edge)$ pair into any priority queue, thus assuming a role analogous to `memoryTrees`'s.

Furthermore, the bootstrap routine creates create *num_threads*, as many as available based on the computing hardware resources, thus taking care of initializing or cleaning up threads' priority queues.

At this stage, the actual search already begins: an index is probed (line 3) to search for all nodes matching at least one query keyword, thus building the corresponding 1-node trees. Starting from such trees, we build `Grow`

opportunities and distribute them evenly among $\texttt{pQueue}_i$, with $1 \leq i \leq num\_threads$ (e.g. round-robin). It is good to notice that, if necessary, it is possible to probe the index and generate the initial `Grow` opportunities also in parallel, with minimal modifications to Algorithm 1.

Algorithm 1: P-GAM bootstrap routine

Input: $G = (N, E)$, query $Q = \{w_1, \ldots, w_m\}$, maximum number of solutions M, time limit T

Output: Answer trees for Q on G

1 initialize/clean `memoryTrees`, `queryAnswers` (and `memoryPairs`, hereinafter omitted for simplicity);
2 initialize/clean $\texttt{pQueue}_i$, $1 \leq i \leq num\_threads$;
3 $init\_nodes_Q \leftarrow \cup_{w_i \in Q}$ `kwd_index_.lookup(`w_i`)`;
4 **for** $node \in init\_nodes_Q$ **do**
5 **for** *edge adjacent to node* **do** push $(node, edge)$ on $\texttt{pQueue}_i$, with $i \sim \mathcal{U}\{1, num\_threads\}$;
6 launch $num\_threads$ P-GAM workers, running Algorithm 2;

Next, $num\_threads$ threads run in parallel Algorithm 2: each of them keeps working until a global stop condition is met, which could happen after a timeout or when the maximum number of solutions has been reached or the priority queue ends up being empty.

Essentially, any worker repeatedly picks the highest-priority $(tree, edge)$ pair from its queue and applies `Grow` on it (lines 2 and 3 in Algorithm 2), leading to a 1-edge larger tree. Thus, the stack priority orders the possible `Grow` steps at a certain point during the search, as outlined in Section 1.3.2.

Successively, if the `Grow` result tree had not been found before (this is determined after looking at `memoryTrees`), the worker tries to `Merge` it with all compatible trees, found within `treesByRoot` (line 9 in Algorithm 2). It is worth underlining that `Merge` partners should match disjoint sets of keywords, as this condition ensures minimality of the solution.

Merging potentially translates into recursive calls to `Aggressive Merge`, outlined in Algorithm 3. As a matter of fact, we immediately try to `Merge` any newly merged tree with any possible candidate. In other words, `Merge` results are repeatedly merged with all compatible trees explored so far as soon as they are found since in this way we are able to detect as quickly as possible if some of our trees might form an answer. Essentially, the thread switches back to `Grow` only when no new `Merge` on the same root is possible.

Algorithm 2: P-GAM thread$_i$ routine

Input: pQueue$_i$, memoryTrees, treesByRoot, queryAnswers
Output: Answer trees for Q on G added to queryAnswers

```
1  repeat
2      opportunity ← pop (tree, edge), the highest-priority pair in
         pQueue_i;
3      t_G ← Grow(tree, edge);
4      if t_G ∉ memoryTrees then
5          add t_G to memoryTrees;
6          if t_G.keywords ∩ Q = Q then add t_G to queryAnswers;
7          else
8              for edge_G adjacent to the root of t_G do push (t_G, edge_G) in
                 pQueue_i;
9              for tree_cand ∈ treesByRoot.get(t_G.root) AND
                 t_G.keywords ∩ tree_cand.keywords = ∅ do
10                 Aggressive Merge(t_G, tree_cand) → Algorithm 3;
11 until reached time limit T OR found M solutions OR pQueue_i empty;
```

Algorithm 3: Aggressive Merge($tree, tree_{cand}$) recursively called by thread$_i$

Input: pQueue$_i$, memoryTrees, treesByRoot, queryAnswers
Output: Answer trees for Q on G added to queryAnswers

```
1  t_M ← Merge(tree, tree_cand);
2  if t_M ∉ memoryTrees then
3      add t_M to memoryTrees;
4      if t_M.keywords ∩ Q = Q then add t_M to queryAnswers;
5      else
6          for edge_M adjacent to the root of t_M do push (t_M, edge_M) in
             pQueue_i;
7          for tree_cand_rec ∈ treesByRoot.get(t_G.root) AND
             t_M.keywords ∩ tree_cand_rec.keywords = ∅ do
8              Aggressive Merge(t_M, tree_cand_rec);
```

In any case, after a newly created tree has been seen for the first time and it is not a solution, `Grow` opportunities are generated and push into the worker's queue (line 8 in Algorithm 2 and line 6 in Algorithm 3). Otherwise, if such a new tree matches all the query keywords, it is added to the solution set and not pushed in any queue (line 6 and line 4 in Algorithm 3).

It should be noted that the threads intensely compete for access to `memoryTrees` and `treesByRoot`. As we demonstrate in Section 4.3.2, our design allows excellent scalability as the number of threads increases.

3.4.2 Work Stealing Implementation

The current implementation hides a critical bottleneck which is be exemplified in the following. Suppose that, based on the computing hardware resources of a given machine, we can run $num\_threads = 10$. Suppose that we have 2 nodes matching some query keywords at the extremes of a highly-connected graph and that those nodes have only one adjacent edge: 2 $(tree, edge)$ pairs are generated in the bootstrap phase. According to the algorithm outlined in Section 3.4.1, GAM will run just on 2 parallel threads (each of them having a priority queue fed with one of the 2 previous `Grow` opportunities) and therefore we will have 8 idle threads. Furthermore, each of these `Grow` opportunities will generate an increasing number of trees and `Grow` opportunities that will be spread only across these 2 threads, even though the degree of parallelism allowed by the current machine is much higher.

Essentially, we instantiate as many threads as the number of $(tree, edge)$ pairs generated in the bootstrap phase, which tells us nothing about the computational scenario that may arise in the future. Not only is the computational load determined at the beginning, but it may stay distributed in a totally uneven manner. All this results in underused computational possibilities. A further example might consist of a given thread that, for whatever reason, runs out of all $(tree, edge)$ pairs in its priority queue and stays idle while everyone else is still performing intensive computations.

It is clear that we need to make inter-thread barriers more permeable so that it is possible to even the workload in case of disparity. As a matter of fact, such an issue is inherently related to data partitioning issue we have several options to address it. For instance, when a $(tree, edge)$ pair is generated, we might push it into a given $\texttt{pQueue}_j$, with $j = \mathcal{H}(tree, edge)$ and $\mathcal{H}$ being an hash function taking a `Grow` opportunity as input and returning $j \in \mathbb{N} \cap [1, num\_threads]$. Following a radically different approach, we

could instead bypass data partitioning and delegate $(tree, edge)$ pairs redistribution to a routine that periodically aligns the priority queues. However, both these methods significantly increase contention on priority queues and massively require lock mechanisms, threatening the scalability of the entire algorithm. Therefore, we propose a solution aimed at minimizing contention.

In Algorithm 4 we rewrite Algorithm 2 tackling issues related to data partitioning by means of work stealing policies. An alternative visualization of the situation is shown in Figure 3.4, as a flowchart displaying the local viewpoint for a given thread_x.

Essentially, the core of the algorithm remains almost unchanged, but searching for $(tree, edge)$ pairs and checking for termination conditions vary slightly. In detail:

1. Each thread_i looks at its own priority queue $\texttt{pQueue}_i$ (line 2): as long as it's not empty, it pulls a $(tree, edge)$ pair at a time and processes it, pushing any resulting `Grow` opportunity in $\texttt{pQueue}_i$ again;

2. When $\texttt{pQueue}_i$ is empty, thread_i may steal a pair from any $\texttt{pQueue}_j$, with $1 \leq j \leq num\_threads$ (line 4), then $\rightarrow 1$;

3. If thread_i has nothing to steal, it checks for termination (line 6); if termination conditions are not satisfied yet, thread_i waits for another pair to be inserted in any $\texttt{pQueue}_j$, with $1 \leq j \leq num\_threads$ (line 7): if thread_i is able to steal the pair $\rightarrow 1$; otherwise $\rightarrow 3$ again.

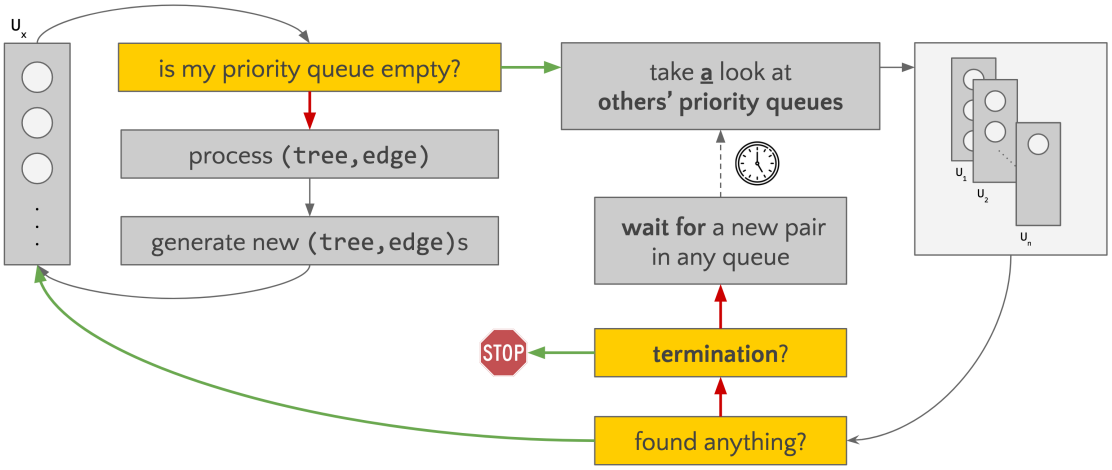


Figure 3.4: Flowchart for thread_x executing P-GAM with work stealing

We can easily notice that work stealing goes much easier on priority queues than any data partitioning or redistribution approach. Indeed, once a pair has been stolen by thread$_i$, it is very likely to bring back many other `Grow` opportunities, which will be pushed to `pQueue`$_i$ itself. Performance improvements are experimentally proved in Section 4.3.2.

Algorithm 4: P-GAM thread$_i$ routine with work stealing

Input: `pQueue`$_i$, `memoryTrees`, `treesByRoot`, `queryAnswers`
Output: Answer trees for Q on G added to `queryAnswers`

```
1  repeat
2      if pQueue_i not empty then opportunity ← pop (tree, edge), the
         highest-priority pair in pQueue_i ;
3      else
4          opportunity ← try to steal (tree, edge) from pQueue_j,
             1 ≤ j ≤ num_threads;
5          while opportunity not found do
6              if reached time limit T OR found M solutions OR all
                 pQueue_j empty, with 1 ≤ j ≤ num_threads then return;
7              wait for (tree, edge) to be inserted in any pQueue_j,
                 1 ≤ i ≤ num_threads;
8              opportunity ← try to steal (tree, edge) from pQueue_j,
                 1 ≤ j ≤ num_threads;
9      t_G ← Grow(tree, edge);
10     if t_G ∉ memoryTrees then
11         add t_G to memoryTrees;
12         if t_G.keywords ∩ Q = Q then add t_G to queryAnswers;
13         else
14             for edge_G adjacent to the root of t_G do push (t_G, edge_G) in
                 pQueue_i;
15             for tree_cand ∈ treesByRoot.get(t_G.root) AND
                 t_G.keywords ∩ tree_cand.keywords = ∅ do
16                 Aggressive Merge(t_G, tree_cand) → Algorithm 3;
17 until reached time limit T OR found M solutions OR all pQueue_j
     empty, with 1 ≤ j ≤ num_threads;
```

Chapter 4

Experimental Results

In the present chapter, we test *CLMem* system and P-GAM algorithm with work stealing described in the previous pages of this work, precisely in the context of the pipeline illustrated in Figure 4.1. In detail, the graph loaded by *CLMem* (Chapter 2) and processed by P-GAM (Chapter 3) has always to be intended as the result of graph construction process carried by *CL* (Chapter 1). Therefore, in the following, we omit the description of the data sources ingestion process and we focus solely on comparing performance on query execution.

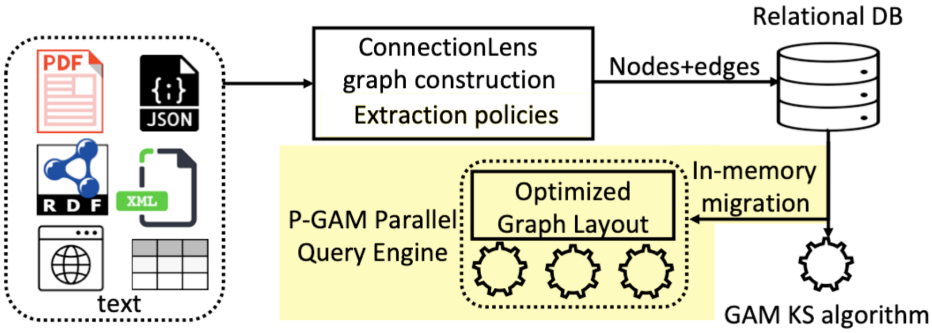


Figure 4.1: Overview on data analysis pipeline

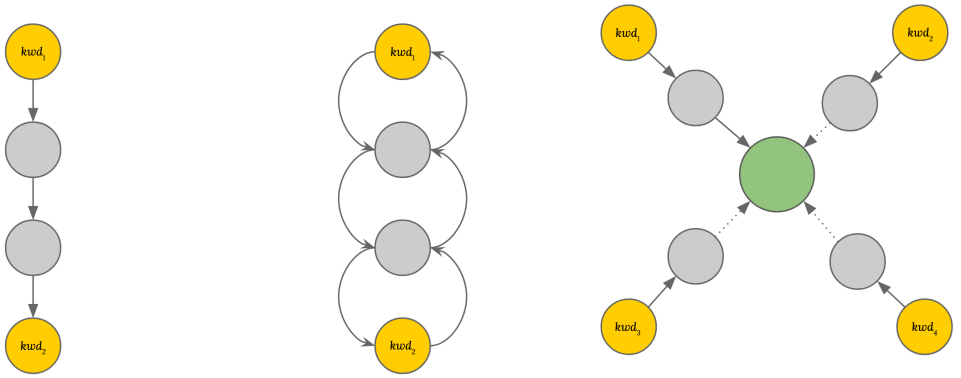
We deployed the following experiments on a server with a 2x10-core Intel Xeon E5-2640 v4 (Broadwell) CPUs clocked at 2.4GHz, and 128GB of DRAM. We do not use Intel Hyper-Threading and we bound every CPU core to a single worker thread. The query engine is a *NUMA-aware* multi-threaded *C++* application. Furthermore, throughout the chapter, we frequently employ the terminology *exhaustive query* to designate a search that comes to an end just because all pairs in every $\texttt{pQueue}_i$, with $1 \leq i \leq num\_threads$

have been consumed, and thus regardless of any termination condition.

Firstly, we introduce the synthetic graphs (Section 4.1) employed to compare single-threaded P-GAM (*as implemented* by *CLMem*) and GAM (*as implemented* by *CL*). After, we test P-GAM performance on the same synthetic topologies (Section 4.3, exploiting the maximum parallelization degree offered by our server. Finally, we build (Section 4.4) and query (Section 4.5) a 20M+ nodes graph, based on a real-life use case scenario involving conflict of interest.

4.1 Synthetic Graphs

First of all, we introduce different synthetic graphs, whose size and topology we can fully control. Next Section 4.3 will deal with querying those graphs by keywords that are matched exclusively by the nodes at the extremes. Examples are shown in Figure 4.2, with matching nodes colored in yellow.



(a) Linear graph with 4 nodes

(b) Chain graph with 4 nodes

(c) Star graph with 4 branches, each of them 2-nodes long

Figure 4.2: Illustrations of synthetic graph topologies introduced for experiments.

The topologies at hand are as follows:

- **Linear graph** $[\![\texttt{line}(n)]\!]$. It consists of $(n-2)$ triples chained together in a single path, plus 2 additional triples attaching the keyword nodes to the first and the last node in the line, matching w_1 and w_2 respectively. An example is shown in Figure 4.2a. On this graph, the query $Q = \{w_1, w_2\}$ has only 1 solution which is the complete graph. Furthermore, we can state that there exist $2n + 1$ partial trees.

- **Chain graph** $[\![\texttt{chain}(n)]\!]$. It is totally similar to `line(`n`)`, with the only difference that edges are doubled. Therefore, we end up with a pair of opposite edges between every pair of adjacent nodes. At the extreme of the chain, w_1 and w_2 are matched. An example is shown in Figure 4.2b. On this graph, the query $Q = \{w_1, w_2\}$ has 2^k solutions, since any two neighbor nodes can be connected by two opposite edges; furthermore, $2^{k+1}-2$ partial trees are built, each containing one keyword plus a path growing toward (but not reaching) the other.

- **Star graph** $[\![\texttt{star}(n,k)]\!]$. It is composed by k `line(`n`)` graphs, each of them having a keyword $w_i, 1 \leq i \leq k$ at one extremity; at the other extremity, all lines have w_c. As explained in Section 1.1, *CL* recognizes that all the nodes matching w_c are equivalent. Therefore, one of them is designated to be their representative while and the others are connected to it through `SAME_AS` edges. An example is shown in Figure 4.2c, where the nodes matching the query keyword are coloured in yellow as usual, while the node matching w_c is displayed in green. On this graph, the query $Q = \{w_1, w_2, \ldots, w_k\}$ has exactly 1 solution which is the complete graph. Furthermore, we can state that there exist $\mathcal{O}(n+1)2^k$ partial trees.

4.2 Single-Threaded P-GAM vs. GAM

We start by comparing single-threaded P-GAM (i.e. $num\_threads = 1$) implemented by *CLMem* with the Java-based GAM implemented by *CL*, which is single-threaded by design and accesses the graph from a *PostgreSQL* database. We run the two algorithms on the synthetic topologies and queries described in Section 4.1, with a time limit T set to *15 minutes*: both can stop earlier if they exhaust the search space. In Table 4.1 we compare the number of solutions S, the time T^1_{P-GAM} (ms) until the first solution is found by single-threaded P-GAM and its total running time T_{P-GAM} (s), as well as the corresponding times T^1_{GAM} (ms) and T_{GAM} (s) for GAM.

On these tiny graphs, both algorithms find all the expected solutions. However, even without parallelism, P-GAM is 10× to more than 100× faster. In particular, on all but the 3 smallest graphs, GAM can not exhaust its search space in *15 minutes*. For instance, it's interesting to notice that P-GAM takes 44.7 s to exhaustively query `star(`5000,4`)`, while *CL* takes about 108 s to find the first (and unique) solution but it's not able to explore the entire search space in 15 minutes.

Graph	S	T^1_{P-GAM} (ms)	T_{P-GAM} (s)	T^1_{GAM} (ms)	T_{GAM} (s)
`chain(12)`	4096	2	0.8	160	674.5
`chain(13)`	8192	4	3.8	203	900.0
`chain(14)`	16384	4	13.7	234	900.0
`chain(15)`	32768	8	53.2	315	900.0
`star(1000,4)`	1	233	0.6	4063	60.2
`star(2000,4)`	1	969	3.3	12580	243.9
`star(3000,4)`	1	2469	10.1	36261	900.0
`star(4000,4)`	1	5149	23.3	67984	900.0
`star(5000,4)`	1	9111	44.7	108960	900.0

Table 4.1: Single-thread P-GAM (i.e. *CLMem*) and GAM (i.e. *CL*) comparison

The present experiment validates the expected orders of magnitude speed-up of a carefully designed in-memory implementation as described in Section 2.2, even without parallelism (since we restricted P-GAM to run on just 1 thread).

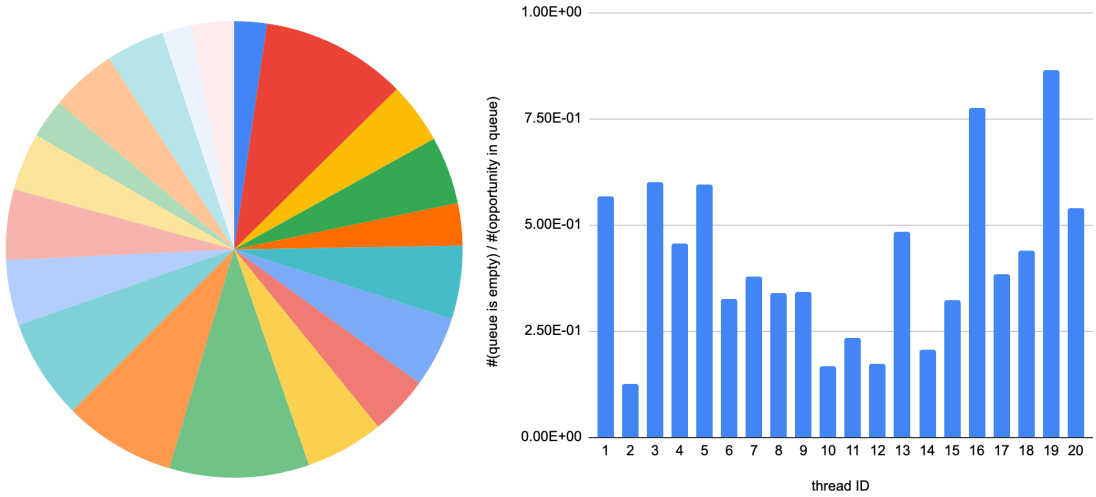
4.3 Scalability Analysis

We perform scalability analysis on synthetic topologies introduced in Section 4.1. Each of them aims at testing a feature of the system and it is associated with a specific query leading to very different search space sizes. In detail, in the following experiments, we gradually increase the size of each graph.

4.3.1 Theoretical Limits

We illustrate the performance measurements resulting from an exhaustive query $Q = \{kwd_0, kwd_1\}$ on `line(1000)` (Figure 4.2a). Here, we want to study the scalability of the algorithm as we increase the graph size.

We recall that work stealing policies are the only agents responsible for data partitioning between threads, as outlined in Section 3.4.2. We immediately notice that they perform quite evenly: the pie chart in Figure 4.3a shows the distribution of $(tree, edge)$ opportunities across threads, with $num\_threads = 20$ (i.e. 1 color per thread). However, Figure 4.3b starts to show some serious criticalities since threads find their priority queues empty



(a) Distribution of $(tree, edge)$ pairs (b) Frequency of empty priority queues

Figure 4.3: Work stealing performance for an exhaustive query on `line(1000)`, with $num\_threads = 20$

with dramatic frequencies: thread_{19} is rarely able to find any `Grow` opportunity in its own $\texttt{pQueue}_{19}$! This results in tremendously aggressive stealing by the threads against each other.

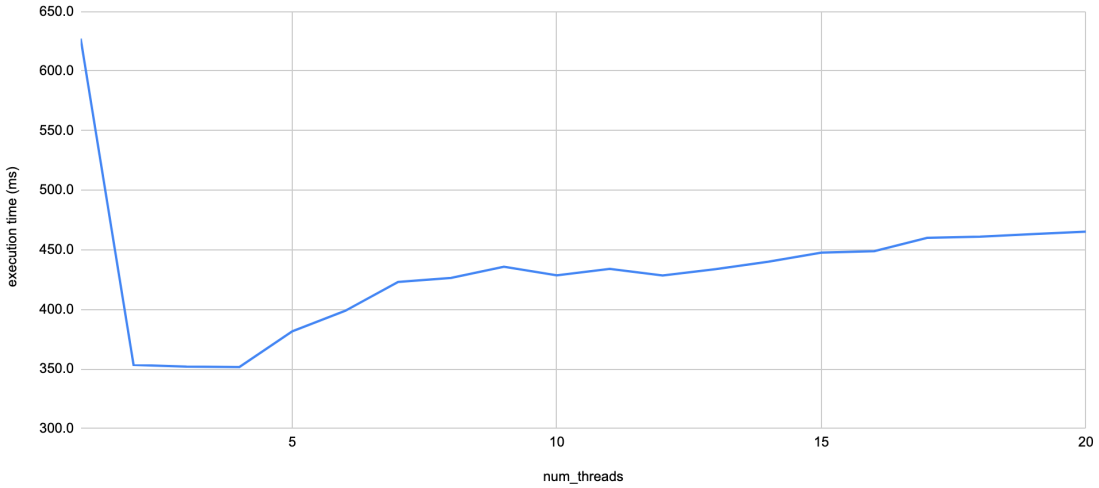


Figure 4.4: Execution time for an exhaustive query on `line(1000)`, as the number of threads increases

Furthermore, Figure 4.4 plots the execution time (ms) as the number of

threads increases from 1 to 20: at first glance, the system doesn't seem to be scalable. The execution time approximately halves in going from 1 to 2 threads, as we would expect from a scalable system. However, performance doesn't improve beyond such a threshold, ending up worsening as the number of threads increases.

As a matter of fact, we underline that the maximum parallelization degree for the query at hand is exactly equal to 2 workers, due to the algorithm design and the graph topology. Indeed, there is no work to be distributed in such a case: every `Grow` opportunity generates just another one, so there are always ≤ 2 $(tree, edge)$ pairs into play. We end up iterating over all $\texttt{pQueue}_j$, with $1 \leq j \leq num\_threads$, looking for a `Grow` opportunity to steal for no good: we just waste resources in order to acquire and release locks.

This leads us to the conclusion that, as far as any linear graph is concerned, having $num\_threads > 2$ not only doesn't bring any performance improvement but it also increases the execution time due to threads violently operating lock mechanisms.

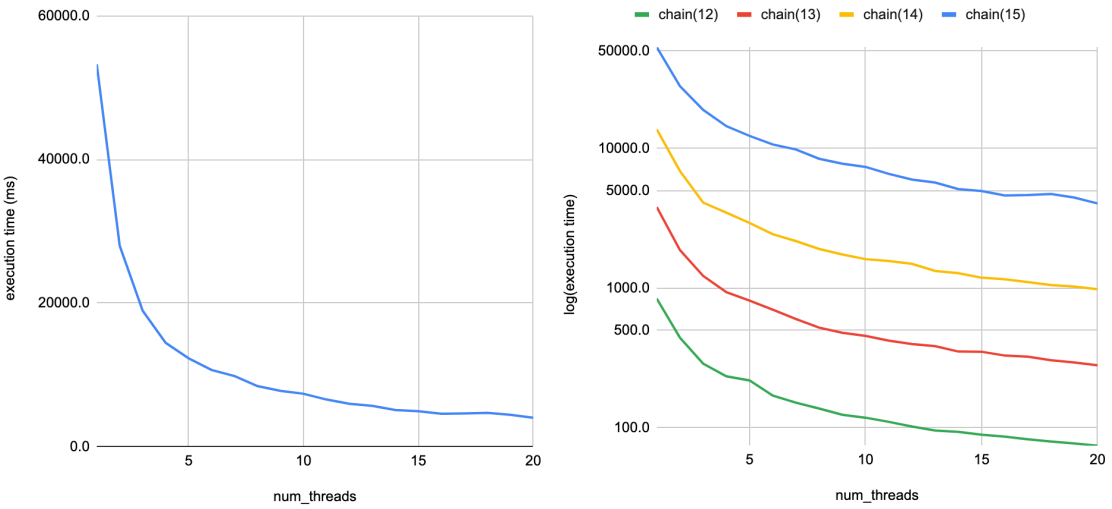
4.3.2 Contention on Shared Data Structures

Next, we illustrate the performance measurements resulting from exhaustive queries on `chain(`n`)`, with $12 \leq j \leq 15$ (Figure 4.2b). Here, we want to study the behavior of the algorithm as we increase the chances for `Grow` and `Merge`. In other words, we want to study to what extent contention in concurrent access to data structures impacts scalability.

In Figure 4.5a we report the execution time (ms) for a exhaustive query $Q = \{kwd_0, kwd_1\}$ on `chain(15)`, as we increase the number of worker threads from 1 to 20. Furthermore, in Figure 4.5b, we also compare the execution time (in logarithmic scale) for a exhaustive query $Q = \{kwd_0, kwd_1\}$ on different `chain(`j`)`, with $12 \leq j \leq 15$.

For every topology, we observe a clear speedup as the number of threads increases, which is on average 13×. The speedup is not exactly linear: 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.

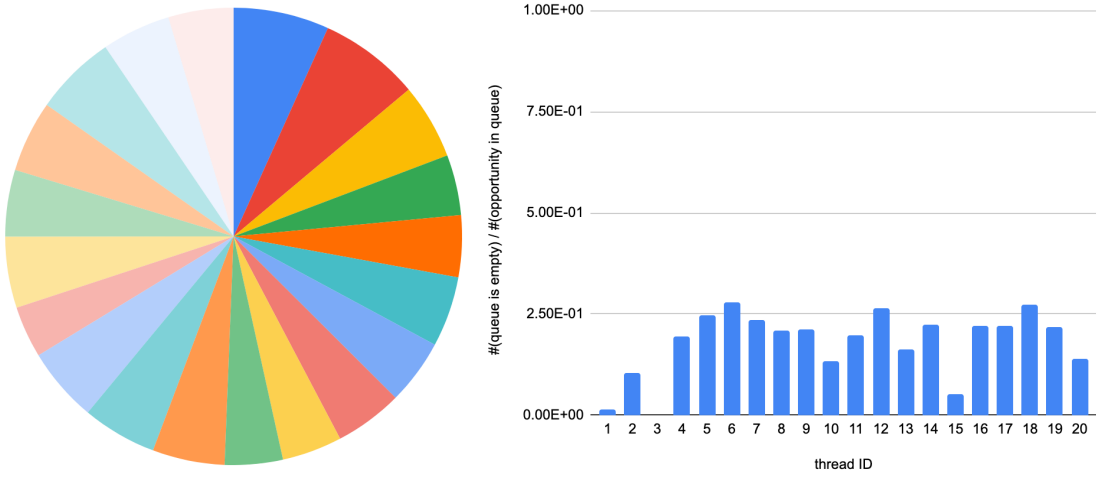
Furthermore, we revisit work stealing performance in the present context, benchmarking an exhaustive query on `chain(12)`. The pie chart in Figure 4.6a shows that $(tree, edge)$ pairs are perfectly distributed across



(a) Query execution time on `chain(15)`

(b) Logarithmic comparison of query execution times on `chain(`j`)`, $12 \leq j \leq 15$

Figure 4.5: Performance plots for exhaustive queries on chain graphs, as the number of threads increases



(a) Distribution of $(tree, edge)$ pairs

(b) Frequency of empty priority queues

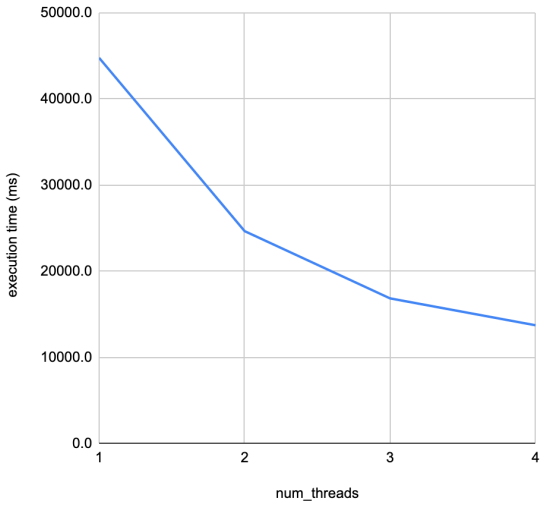
Figure 4.6: Work stealing performance for an exhaustive query on `chain(12)`, with $num\_threads = 20$

threads; at the same time, Figure 4.6b displays that is extremely unlikely for any thread_j to find its own $\texttt{pQueue}_j$ empty. Therefore, we demonstrate

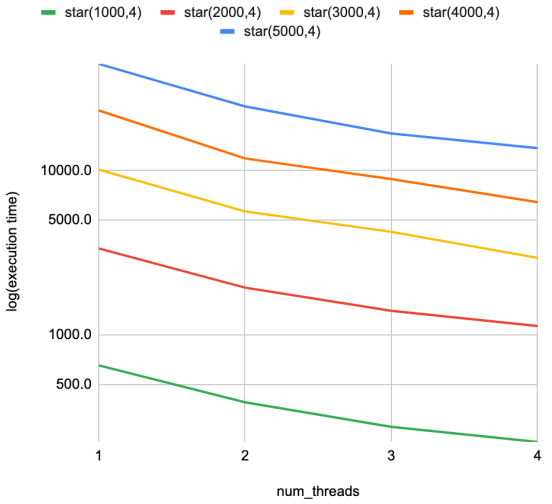
that the policies in charge of distributing the workload are indeed efficient and operate fairly. Such an experimental result can be extended to any future scenario and to the entire system in general.

4.3.3 Graph Size

Finally, we outline the performance measurements resulting from exhaustive queries on `star(`n`,4)` with $n \in \{1000, 2000, 3000, 4000, 5000\}$ (Figure 4.2b). Here, we want to study again the scalability of the algorithm as we increase the graph size. In fact, after having isolated and studied the limitations of the algorithm on linear topologies as we did in Section 4.3.1, we rather exploit its constraints by building a synthetic scenario that can serve our intended purposes.



(a) Query execution time on `star(5000,4)`



(b) Logarithmic comparison of query execution times on `star(`n`,4)` with $n = 1000k, 1 \leq k \leq 5$

Figure 4.7: Performance plots for exhaustive queries on star graphs, as the number of threads increases

In Figure 4.7a we report the execution time (ms) for an exhaustive query $Q = \{kwd_0, kwd_1, kwd_3, kwd_4\}$ on `star(5000,4)`, as we increase the number of worker threads from 1 to 4. Furthermore, in Figure 4.7b, we also compare the execution time (in logarithmic scale) for a exhaustive query $Q = \{kwd_0, kwd_1, kwd_3, kwd_4\}$ on different `star(`n`,4)` with $n \in \{1000, 2000, 3000, 4000, 5000\}$. We obtain an average speed-up of 3.2×

with $num\_threads = 4$, 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 outlined in Section 4.3.2. We need to point out that, in the present 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 the nodes matching the query keywords are poorly connected (e.g. in our star graphs, at the end of simple paths) P-GAM 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 one thread or another. Indeed, on any `chain(`n`)` topology, the presence of 2 edges between any adjacent nodes multiplies the `Grow` and `Merge` opportunities, which can be shared by many threads. This is why on chain graphs we see scalability up to the maximum number of threads that our server supports.

4.4 Real-Life Use Case Scenario

In the current section, we introduce the real-life data that we query in the experiments described in Section 4.5. In detail, we focus on a particular use case, that is conflicts of interest (*CoI*, in short) in the biomedical domain.

As a matter of fact, 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, investigative journalism work has shown that disclosure information is often scattered across multiple data sources, hindering access to this information [15]–[17].

In the following, we refer to Figure 1.1 reported in Section 1.1 in order to show a tiny fragment of data that can be used to find connections between scientists and companies. For the current purposes, we 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 1.1.

Thus, hundreds of millions of bibliographic notices (in XML) are published on the *PubMed* website; the site also links to research (in PDF). In recent years, *PubMed* has included an optional *CoI statement* 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 (i.e. "The authors declare no conflict of interest"). Nevertheless, within the PDF papers themselves, paragraphs like "Acknowledgments" and "Disclosure statement" may contain such information, even if the *CoI statement* is absent or empty. This information is accessible if one converts the PDF in a format such as JSON. In Figure 1.1, Alice declares her consulting for *ABCPharma* in XML, yet the "Acknowledgments" paragraph in her PDF paper mentions *HealthStar*.

Furthermore, 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 a scenario related to investigative journalism are often missing from such knowledge bases (e.g. *HealthStar*, in our example). Specialized data sources, such as a trade catalog or a *Wiki website* built by other investigative journalists may provide information on some such actors: in our example, the *PharmaLeaks* website shows that *HealthStar* is also funded by the industry. Such a site which is 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 investigative journalism work. In this and many scenarios, sources are highly heterogeneous; at the same time, time, skills, and resources to curate, clean, or structure the data are not available.

In short, a database of known relationships between experts and interested companies, built by integrating heterogeneous data sources, would be a very valuable asset. For instance, in Europe, such a database could be used to select, for a committee advising EU officials on industrial pollutants, experts with few or no such relationships. In the US, the Sunshine Act [18] requires manufacturers of drugs and medical devices to declare such information, but this does not extend to companies from other sectors.

For this reason, we loaded 400,000 *PubMed* bibliographic notices (XML), corresponding to articles from 2019 and 2020: they occupy 803MB on disk. Then, we have downloaded 85,400 PDF articles corresponding to these notices (those that were available in *Open Access*), transformed them into JSON using an extraction script integrated in *ConnectionsLens*' pipeline. Furthermore, we preserved only those paragraphs starting with a set of keywords (e.g. "Disclosure", "Competing Interest", "Acknowlegments", etc.) which have been shown to encode potentially interesting participation of people

(other than authors) and organizations in an article [19]. Together, these JSON fragments occupy 173MB 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 1.1. Finally, we have crawled 375 HTML webpages from a set of websites describing people and organizations previously involved in scientific expertise on sensitive topics (such as tobacco or endocrine disruptors), specifically: DeSmog, TobaccoTactics, WikiCorporates and SourceWatch. These pages total 32MB.

Table 4.2 shows the number of edges $|E|$ and the numbers of nodes $|N|$ resulting from loading the data sources; furthermore, it displays the number of nodes of type `ENTITY_PERSON`, `ENTITY_ORGANIZATION` and `ENTITY_LOCATION` in columns $|N_P|$, $|N_O|$ and $|N_L|$ respectively. The statistics are split by data model and overall.

Source	$\|E\|$	$\|N\|$	$\|N_P\|$	$\|N_O\|$	$\|N_L\|$
XML	32,028,429	19,851,90	1,483,631	584,734	126,629
JSON	1,025,307	432,303	75,297	7,320	4,139
HTML	246,636	185,479	3,726	7,227	320
Total	33,300,372	20,469,686	1,562,654	665,167	131,088

Table 4.2: Statistics on *Conflict of Interest* graph.

4.5 Querying Real-Life Data

We want to test *CLMem* on the database that we built by loading real-life data described in Section 4.4.

Table 4.3 shows the results of executing 15 queries, until we get $M = 1000$ solutions or a time limit $T = 60000$ ms is reached. From left to right, the columns show: a query identifier, the query keywords Q, 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. We pay particular attention to the last column $\#DS$, which describes the distribution of the number of data sources spanned by the solutions, and we underline the most frequent element: for instance, the solutions for the query #1 span at least 2 and at most 10 data sources, while most solutions spanned over 6 sources.

We have anonymized the keywords that we use, not to single out individuals or corporations 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,

#	Keywords	T^1	T^{last}	T	S	$\#DS$
1	A1, A2	4462	5315	5316	1000	2-10, **6**
2	A3, H1	4671	5140	5140	1000	3-7, **6**
3	U1, H1	4832	4981	4981	1000	2-5, **5**
4	A4, I1	8520	13711	13712	1000	2-5, **5**
5	A5, I2	5800	6366	6366	1000	2-8, **8**
6	A6, I3, P1	4657	5072	60000	16	4, **4**
7	A7, I3, P2	44256	44273	60000	10	5, **5**
8	A8, I4, P3	12560	12560	60000	2	5, **5**
9	A9, I4, P3	28982	33435	60000	3	5, **5**
10	A10, U1, I3	7577	17383	17383	1000	4-6, **6**
11	A11, I4, I5	10396	32320	60000	6	3, **3**
12	A12, I4, I6	7320	7467	60000	24	4, **4**
13	A3, A13, U2, P4	15759	35025	60000	5	5-6, 8, **6 and 8**
14	A3, A14, U3, G1	10711	10711	60000	1	7, **7**
15	A3, A15, U4, P4	8560	9942	60000	16	9, **9**

Table 4.3: Statistics on *Conflict of Interest* graph.

G for government service, H for hospital, P for country, U for university, and I for industry (i.e. company).

We make several observations based on the results. The termination conditions were set here based on what we consider as an interactive query response time and a number of solutions that allow further exploration by the users (e.g. through an interactive GUI available in *CL*). Furthermore, 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 amount of keywords and the system remains responsive within the same time bounds, despite the increasing query complexity.

Chapter 5

Related Work

In the present chapter, we review and discuss existing literature pertaining to our work. As a matter of fact, *ConnectionLens* (along with *CLMem*) is placed at the intersection of diverse research areas. Therefore, Section 5.1 addresses the matter of data integration from diverse and heterogeneous sources. Subsequently, in Section 5.2, we review several studies related to keyword search on a variety of documents and databases. Finally, Section 5.3 goes through relevant work in the field of graph processing, with a specific focus on in-memory systems.

5.1 Data Integration

The *ConnectionLens* project and software system discussed in the present work and all its components (including *CLMem*) belong to the area of *data integration*. As outlined by Doan et at. [20], *data integration* is the process of abstracting away the fact that data come from several data sources, possibly with different schemas. They distinguish two main possible approaches, with hybrid and intermediate architectures in between: (*i*) *data warehousing* extracts, transforms, and loads data from heterogeneous sources into a single physical warehouse; (*ii*) *virtual integration* preserves the data in their original repositories and accesses them only at query time by means of *mediators*, which are specialized modules that distribute the work to each source and combine the results.

The first prototype of *ConnectionLens* was indeed a mediator [21], deploying solutions that are very similar to those of *polystores*. As described by Duggan et al. [22], a *polystore* is meant to unify querying over multiple data models by enabling: (*i*) *location transparency*, allowing users to

ask queries without having to deal with the physical locations of the different data management system and the work distribution among them; (*ii*) *semantic completeness*, which preserves any capability provided by the underlying storage engines. Similar work has been proposed by Kolev et at. [23], which introduce a functional SQL-like language, capable of querying and integrating heterogeneous cloud data stores within a single query, supporting nested queries across data stores, schema independence and optimizability; furthermore, this comes with a fully distributed query engine.

However, the ultimate applicative scenario of the present work must not be disregarded. In fact, *ConnectionLens* certainly originates as a research project with all of its inherent challenges, but it also faces the practical needs of investigative journalists, as described in Chapter 1. Therefore, 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. Consolidating the graph in a single store and centralizing GAM algorithm [1] greatly sped up and simplified the tool, whose performance is further improved in the present work.

We share the goal of exploring and connecting data with data discovery methods [24]–[27], which mostly focus on tabular data. While our data is heterogeneous, focusing on an investigative journalism application partially eliminates risks of ambiguity, since in our context the name of one person or organization typically denotes a single concept.

5.2 Keyword Search

ConnectionLens' first iteration [3] obviated the need for knowing the schema underlying the data by introducing *keyword search* (hereinafter also KS, in short). As a matter of fact, KS is indubitably popular among search engines and in general for unstructured data. On the other hand, as databases continue to grow, Yu et at. [28] propose KS as a search method for structured data as well, which is particularly effective if users do not know exactly what kind of data they are querying and its underlying schema. According to the authors, structural keyword search is about finding interconnection among object structures and answers enabled by several tuple connection networks.

Congruently, Hristidis et al. [29] represent tuples as nodes, joined by means of their primary and foreign keys, thus returning a network of tuples containing all the query keywords. De Oliveira et al. [30] perform a similar work with

a focus on the size of the search space and the set of candidate networks. The authors propose ranking as a way to identify relevant answers that might be evaluated and eventually returned to the user. Furthermore, Yan et al. [31] describe an active learning approach that incorporates user feedback in order to return informative and relevant answers. Vu et at. [32] extend KS on relational data by providing support for keyword-based data sharing and querying over multiple database management systems. Similarly, Sayyadian et al. [33] integrate tuples from multiple and heterogeneous databases, introducing schema matching techniques that establish connections based on similarity of values for different attributes. However, as outlined in Section 1.3, we process edges as if they were undirected and, in general, paths ends up being much longer than those based only on primary key/foreign key.

Guo et al. [34] study keyword-based search on XML documents, targeting their peculiarities. In particular, they point to the need for identifying a lower granularity in the computation of ranks, which no longer operates at the level of documents (e.g. HTML pages) but rather at the level of nested elements. Similarly, Liu et al. [35] focus on formulating an appropriate return clause that is able to mirror the semantics of the query by enumerating eligible and desirable nodes with no need for the user to explicitly give preferences. However, querying XML documents based on keywords has a lower complexity than our problem's: (*i*) an XML document defines a tree, with just a single hierarchical edge between any pair of nodes; (*ii*) the search space for a given answer is bounded to the height of an XML tree.

Keyword-based search on RDF graph has been studied by Elbassuoni et al. [36], which propose backtracking algorithms to retrieve subgraphs that match the query keywords, ranked by means of structure-aware statistical language models. Even more, Le et at. [37] introduce summarization techniques over RDF data which eventually leads to pruning the search space for exploratory KS, resulting in scalability. However, the work imposes some degree of regularity on the graph structure whose exploration is ruled by the direction of the edges, as well as in [36].

In conclusion, our KS 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; (*iii*) we make no assumption on the score function, invalidating dynamic programming methods such as [35] and other similar pruning techniques. In particular, researcher from CEDAR team has shown in [2] 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 dynamic programming.

Works on parallel keyword search in graphs either consider a different setting, returning a certain class of subgraphs instead of trees [38], or standard graph traversal algorithms like BFS [39]–[41]. To the best of our knowledge, GAM is the first keyword search algorithm for the specific problem that we consider.

5.3 Graph Processing Systems

In the present work, we have parallelized GAM into P-GAM, by drawing inspiration and addressing common challenges raised in graph processing systems in the literature.

Lissandrini et al. [42] provide a comprehensive study of the existing *graph database systems*. In particular, the authors distinguish *graph processing systems* (i.e. which analyze graphs to discover characteristic properties) and *graph database*s (i.e. which focus on storage, high-throughput querying tasks and transactional operations) and they direct their analysis specifically at *graph databases*. In designing the evaluation methodology, the authors follow a *microbenchmarking* approach, which is an evaluation model based on primitive operators derived by decomposing complex queries and allowing to identify the exact components that underperform for a given system. Therefore, the evaluation methodology has been applied on the major graph databases available today, using different real and synthetic datasets. There follows an exhaustive list of the systems considered in the present work: *ArangoDB* [43], *BlazeGraph* [44], *Neo4J* [45], *OrientDB* [46], *Sparksee* [47], *SQLG* [48], *Titan* [49]. Essentially, the work is a compendium of state-of-the-art systems and the primitive operators implemented by them. On the other hand, it allows us to identify which of such operations are in fact implemented and required by *CLMem*. As a matter of fact, we don't need to support any database modification operation since we assume that our graph is immutable once it has gone through the migration pipeline described in Section 2.1. Furthermore, many query operators making up the microbenchmarks described in [42] have nothing or very few points in common with GAM/P-GAM. In conclusion, designing a native query engine allows us to relax many requirements that do not serve the context at hand and rather specialize in addressing its peculiarities.

In [50], Malicevic et al. state that improvements in algorithm execution

time often come at the cost of increased *pre-processing time*. In detail, pre-processing costs are a direct consequence of data layout: indeed, realizing the most straightforward data layout translates into $\mathcal{O}(n)$, with n being the size of the data. Therefore, the authors study and benchmark various data structures to represent the graph in memory, various approaches to pre-processing and various ways to structure the graph computation. They also investigate approaches to improve cache locality, synchronisation, and NUMA-awareness. In doing so, they take a cue from a number of graph processing systems and implement the techniques they propose in a single system. The paper identifies in the literature 3 main approaches to graph data layout: (*i*) edge arrays, (*ii*) adjacency lists (i.e. per-node edge arrays), (*iii*) grid. Then, the authors differentiate *edge-centric* algorithms (e.g. *PageRank*) from *node-centric* algorithms (e.g. *BFS*). Eventually, they prove that unsorted adjacency lists represent the solution with the best end-to-end execution time. In particular, sorting the per-node edge arrays degrades the end-to-end performance with respect to unsorted adjacency lists: the pre-processing time increases, while the algorithm execution time does not decrease and the cache miss ratio does not improve. As outlined in Section 1.3, when executing GAM/P-GAM, we explore the graph by building `Grow` opportunities: we iterate over all the edges adjacent to the current root before processing another node. Therefore, GAM/P-GAM is a *node-centric* algorithm. As a consequence, adopting adjacency lists as data layout allows us to bring into last level cache only and exactly the data we are interested in when building `Grow` opportunities starting from a tree rooted in a given node. Therefore, although GAM/P-GAM has substantial differences with any algorithm that is benchmarked in [50], adjacency lists represent the layout that is realized in Section 2.2.

Elyasi et al. [10] carry out a study involving emerging storage devices (e.g. SSDs) as opposed to relying on DRAM to perform large-scale graph processing. By proposing a specific methodology for partitioning vertex data to reduce computational overhead, the authors prove that their optimizations outperform the state-of-the-art solutions by a factor of 2X. Future developments for *CLMem* could certainly consider emerging storage devices as an alternative to DRAM. However, we highlight that [10] draws its conclusions based on classical algorithms such as *BFS* and *PageRank* as benchmarks.

Several works have focused on studying and leveraging storage bandwidth. In [6], Boncz et at. investigate the impact of main memory access as the main performance bottleneck in database applications. Thus, the authors propose a set of guidelines in terms of data structure and algorithm design. Similarly,

Ahn et al. [9] highlight the critical points of graph processing systems working in main memory, in particular their scalability challenges due to severe memory bandwidth limitations. In this regard, the authors propose a programmable accelerator that is specialized for graph processing and able to exploit internal memory bandwidth efficiently. Also, in [12], Roy et al. propose an *edge-centric* approach to the *scatter-gather* programming model. The authors state that exploiting sequential bandwidth for a given storage media by streaming completely unordered edge lists might perform better than indexing the edge list, thus performing random access. Furthermore, Hong et al. [11] state that, in general, distributed environments end up being less performative than single-machine systems making efficient use of resources. Then, they outline a distributed graph process engine which outperforms an implementation optimized for single-machine execution.

Finally, we followed the work done by Binnig et al. [51] in order to index node labels while allowing for prefix matching. However, we extended the structure to support characters of type `wchar_t`, since ASCII encoding is not sufficient to represent our strings.

Chapter 6

Conclusion

The *ConnectionLens* project developed by the CEDAR team at INRIA answers the practical need to aggregate and link information coming from different sources, in order to support investigative journalism based on digital data. In this regard, *CL* is a software capable of receiving more or less structured data sources (e.g. JSON, HTML, PDF, RDF, textual files, etc.) and connecting them in the form of a graph [2]. Moreover, by means of natural language processing techniques, the system is able to isolate and disambiguate entities (e.g. people, organizations, universities, etc.). Such a graph is then stored in a relational database (i.e. *PostgreSQL*). Therefore, the GAM keyword search algorithm has been developed to answer queries. However, the responsiveness of the system is largely bounded by *PostgreSQL* and, in general, by disk I/O. This is even more dramatic in the context of large amounts of data that characterize the present context.

The goal of the present work has been to improve *CL*'s pipeline in terms of scalability. To that end, firstly we developed a data migration module that retrieves the graph from *PostgreSQL* and loads it into a scalable in-memory store (i.e. *CLMem*). Special attention has been paid to memory layout, in order to favor spatial locality of reference and limit cache evictions.

Then, a parallel, in-memory variant of the GAM algorithm has been designed and implemented (i.e. P-GAM), which also enabled a completeness analysis of a previously proposed pruning heuristic. In this context, sequential and concurrent data structures have also been developed and highly specialized in performing the specific operations required by the operational scenario at hand.

Subsequently, we have tested *CLMem*'s performance on synthetic graph topologies, in order to validate scalability in a controlled environment. We

verified that *CLMem* scales successfully on both the graph size and the data structures. Finally, we built real-life graphs out of data sources suggested by our partnership with Le Monde, and we confirmed *CLMem*'s features.

The work presented here and carried during the candidate's internship at Inria contributed to a full paper submitted for evaluation at the PVLDB Journal 2021 (Scalable Data Science track) and at the IEEE Bulletin of the Technical Committee on Data Engineering.

References

[1] O. Balalau, C. Conceição, H. Galhardas, I. Manolescu, T. Merabti, J. You, and Y. Youssef, *Graph integration of structured, semistructured and unstructured data for data journalism*, Informal publication only (publication informelle), Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-02904797.

[2] A. C. Anadiotis, M. Y. Haddad, and I. Manolescu, «Graph-based keyword search in heterogeneous data sources», in *BDA 2020 - 36ème Conférence sur la Gestion de Données – Principes, Technologies et Applications*, Informal publication only, Online, France, Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-02934277.

[3] C. Chanial, R. Dziri, H. Galhardas, J. Leblay, M.-H. Le Nguyen, and I. Manolescu, «ConnectionLens: Finding Connections Across Heterogeneous Data Sources», *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, p. 4, 2018. DOI: 10.14778/3229863.3236252. [Online]. Available: https://hal.inria.fr/hal-01841009.

[4] A.-C. Anadiotis, O. Balalau, T. Bouganim, F. Chimienti, H. Galhardas, M. Y. Haddad, S. Horel, I. Manolescu, and Y. Youssef, *Empowering Investigative Journalism with Graph-based Heterogeneous Data Management*, 2021. arXiv: 2102.04141 [cs.DB].

[5] *Amazon Elastic Compute Cloud Documentation*. [Online]. Available: https://docs.aws.amazon.com/ec2.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten, «Database Architecture Optimized for the New Bottleneck: Memory Access», in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 54–65, ISBN: 1558606157.

[7] *Psycopg – PostgreSQL database adapter for Python*. [Online]. Available: https://www.psycopg.org/docs/.

[8] *Protocol Buffers - Language Guide*. [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview.

[9] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, «A scalable processing-in-memory accelerator for parallel graph processing», in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117. DOI: 10.1145/2749469.2750386.

[10] N. Elyasi, C. Choi, and A. Sivasubramaniam, «Large-Scale Graph Processing on Emerging Storage Devices», in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 309–316, ISBN: 978-1-939133-09-0. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/elyasi.

[11] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, «PGX.D: A Fast Distributed Graph Processing Engine», in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, Texas: Association for Computing Machinery, 2015, ISBN: 9781450337236. DOI: 10.1145/2807591.2807620. [Online]. Available: https://doi.org/10.1145/2807591.2807620.

[12] A. Roy, I. Mihailovic, and W. Zwaenepoel, «X-Stream: Edge-Centric Graph Processing Using Streaming Partitions», in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 472–488, ISBN: 9781450323888. DOI: 10.1145/2517349.2522740. [Online]. Available: https://doi.org/10.1145/2517349.2522740.

[13] B. Ding, J. Xu Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, «Finding Top-k Min-Cost Connected Trees in Databases», in *2007 IEEE 23rd International Conference on Data Engineering*, 2007, pp. 836–845. DOI: 10.1109/ICDE.2007.367929.

[14] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0201575949.

[15] S. Horel, *Lobbytomie*, French. France: La Découverte, 2018, ISBN: 9782707194121.

[16] ——, (2020). «Petites ficelles et grandes manœuvres de l'industrie du tabac pour réhabiliter la nicotine». French, [Online]. Available: https://www.lemonde.fr/planete/article/2020/12/19/petites-ficelles-et-grandes-man-uvres-de-l-industrie-du-tabac-pour-rehabiliter-la-nicotine_6063922_3244.html (visited on 02/28/2021).

[17] F. Nargesian, K. Q. Pu, E. Zhu, B. Ghadiri Bashardoost, and R. J. Miller, «Organizing Data Lakes for Navigation», in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1939–1950, ISBN: 9781450367356. DOI: 10.1145/3318464.3380605. [Online]. Available: https://doi.org/10.1145/3318464.3380605.

[18] C. Grassley. (2009). «Physician Payments Sunshine Act of 2009», [Online]. Available: https://www.congress.gov/bill/111th-congress/senate-bill/301 (visited on 02/28/2021).

[19] S. Horel and S. Foucart. (2018). «Monsanto Papers». French, [Online]. Available: https://www.europeanpressprize.com/article/monsanto-papers/ (visited on 02/28/2021).

[20] A. Doan, A. Y. Halevy, and Z. G. Ives, *Principles of Data Integration*. Morgan Kaufmann, 2012, ISBN: 978-0-12-416044-6. [Online]. Available: http://research.cs.wisc.edu/dibook/.

[21] 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», in *VLDB*, New Delhi, India, Sep. 2016. [Online]. Available: https://hal.inria.fr/hal-01321201.

[22] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, «The BigDAWG Polystore System», *SIGMOD Rec.*, vol. 44, no. 2, pp. 11–16, Aug. 2015, ISSN: 0163-5808. DOI: 10.1145/2814710.2814713. [Online]. Available: https://doi.org/10.1145/2814710.2814713.

[23] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. O. Pereira, «CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language», *Distributed and Parallel Databases*, vol. 34, no. 4, pp. 463–503, Dec. 2016. DOI: 10.1007/s10619-015-

7185-y. [Online]. Available: https://hal-lirmm.ccsd.cnrs.fr/lirmm-01184016.

[24] A. Das Sarma, L. Fang, N. Gupta, A. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu, «Finding Related Tables», in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 817–828, ISBN: 9781450312479. DOI: 10.1145/2213836.2213962. [Online]. Available: https://doi.org/10.1145/2213836.2213962.

[25] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, «Aurum: A Data Discovery System», in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1001–1012. DOI: 10.1109/ICDE.2018.00094.

[26] R. Castro Fernandez, E. Mansour, A. A. Qahtan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang, «Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery», in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 989–1000. DOI: 10.1109/ICDE.2018.00093.

[27] M. Ota, H. Müller, J. Freire, and D. Srivastava, «Data-Driven Domain Discovery for Structured Datasets», *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 953–967, Mar. 2020, ISSN: 2150-8097. DOI: 10.14778/3384345.3384346. [Online]. Available: https://doi.org/10.14778/3384345.3384346.

[28] J. X. Yu, L. Qin, and L. Chang, «Keyword Search in Databases», *Synthesis Lectures on Data Management*, vol. 1, no. 1, pp. 1–155, 2009. DOI: 10.2200/S00231ED1V01Y200912DTM001. eprint: https://doi.org/10.2200/S00231ED1V01Y200912DTM001. [Online]. Available: https://doi.org/10.2200/S00231ED1V01Y200912DTM001.

[29] V. Hristidis and Y. Papakonstantinou, «Discover: Keyword Search in Relational Databases», in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02, Hong Kong, China: VLDB Endowment, 2002, pp. 670–681.

[30] P. de Oliveira, A. da Silva, and E. de Moura, «Ranking Candidate Networks of relations to improve keyword search over relational databases», in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 399–410. DOI: 10.1109/ICDE.2015.7113301.

[31] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu, «Active Learning in Keyword Search-Based Data Integration», *The VLDB Journal*, vol. 24, no. 5, pp. 611–631, Oct. 2015, ISSN: 1066-8888. DOI: 10.1007/s00778-014-0374-x. [Online]. Available: https://doi.org/10.1007/s00778-014-0374-x.

[32] 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 *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, pp. 915–926, ISBN: 9781605581026. DOI: 10.1145/1376616.1376707. [Online]. Available: https://doi.org/10.1145/1376616.1376707.

[33] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano, «Efficient Keyword Search Across Heterogeneous Relational Databases», in *2007 IEEE 23rd International Conference on Data Engineering*, 2007, pp. 346–355. DOI: 10.1109/ICDE.2007.367880.

[34] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, «XRANK: ranked keyword search over XML documents», *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Apr. 2003. DOI: 10.1145/872757.872762.

[35] Z. Liu and Y. Chen, «Identifying meaningful return information for XML keyword search», Jan. 2007, pp. 329–340. DOI: 10.1145/1247480.1247518.

[36] S. Elbassuoni and R. Blanco, «Keyword Search over RDF Graphs», in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11, Glasgow, Scotland, UK: Association for Computing Machinery, 2011, pp. 237–242, ISBN: 9781450307178. DOI: 10.1145/2063576.2063615. [Online]. Available: https://doi.org/10.1145/2063576.2063615.

[37] W. Le, F. Li, A. Kementsietsidis, and S. Duan, «Scalable Keyword Search on Large RDF Data», *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 11, pp. 2774–2788, 2014. DOI: 10.1109/TKDE.2014.2302294.

[38] Y. Yang, D. Agrawal, H. V. Jagadish, A. K. H. Tung, and S. Wu, «An Efficient Parallel Keyword Search Engine on Knowledge Graphs», in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 338–349. DOI: 10.1109/ICDE.2019.00038.

[39] S. Hong, T. Oguntebi, and K. Olukotun, «Efficient Parallel Graph Exploration on Multi-Core CPU and GPU», in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 78–88. DOI: 10.1109/PACT.2011.14.

[40] L. Dhulipala, G. Blelloch, and J. Shun, «Julienne: A Framework for Parallel Graph Algorithms Using Work-Efficient Bucketing», in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17, Washington, DC, USA: Association for Computing Machinery, 2017, pp. 293–304, ISBN: 9781450345934. DOI: 10.1145/3087556.3087580. [Online]. Available: https://doi.org/10.1145/3087556.3087580.

[41] 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 *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10, Thira, Santorini, Greece: Association for Computing Machinery, 2010, pp. 303–314, ISBN: 9781450300797. DOI: 10.1145/1810479.1810534. [Online]. Available: https://doi.org/10.1145/1810479.1810534.

[42] M. Lissandrini, M. Brugnara, and Y. Velegrakis, «Beyond macrobenchmarks: microbenchmark-based graph database evaluation», *Proceedings of the VLDB Endowment*, vol. 12, pp. 390–403, Dec. 2018. DOI: 10.14778/3297753.3297759.

[43] *ArangoDB - Documentation Overview*. [Online]. Available: https://www.arangodb.com/documentation.

[44] *Blazegraph Database*. [Online]. Available: https://blazegraph.com.

[45] *https://neo4j.com*. [Online]. Available: https://github.com/blazegraph/database/wiki.

[46] *OrientDB - Community Edition*. [Online]. Available: https://orientdb.org.

[47] *Sparksee - Out-of-core graph database for edge computing*. [Online]. Available: https://www.sparsity-technologies.com/#sparksee.

[48] *Sqlg Documentation*. [Online]. Available: http://www.sqlg.org/docs/2.1.0.

[49] *Titan Documentation*. [Online]. Available: http://s3.thinkaurelius.com/docs/titan/1.0.0.

[50] J. Malicevic, B. Lepers, and W. Zwaenepoel, «Everything you always wanted to know about multicore graph processing but were afraid to ask», in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, Jul. 2017, pp. 631–643, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/malicevic.

[51] C. Binnig, S. Hildenbrand, and F. Färber, «Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores», in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 283–296, ISBN: 9781605585512. DOI: 10.1145/1559845.1559877. [Online]. Available: https://doi.org/10.1145/1559845.1559877.

[52] A. P. Nyrkov, K. A. Ianiushkin, A. A. Nyrkov, Y. N. Romanova, and V. D. Gaskarov, «Dynamic Shared Memory Pool Management Method in Soft Real-Time Systems», in *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 2020, pp. 438–440. DOI: 10.1109/EIConRus49466.2020.9039354.

[53] B. Kenwright, «Fast Efficient Fixed-Size Memory Pool: No Loops and No Overhead», 2012.

[54] *Fast Memory Pool (2.10) - Documentation*. [Online]. Available: https://www.pjsip.org/docs/latest/pjlib/docs/html/group__PJ__POOL__GROUP.htm.

[55] *posix_memalign - The Open Group Base Specifications Issue 7, 2018 edition*. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/functions/posix_memalign.html.

Appendix A

Memory Pool

In the present chapter, we outline the principles behind *CLMem*'s *memory pool*. Inspired by related work [52]–[54], we have equipped our system with a software abstraction that allows efficient management of the memory allocated by the operating system to a given process. Indeed, a *memory pool* implements a collection of *ready-to-use* memory resources thus avoiding constantly requesting services to the operating system.

In general, crossing the barrier of the system call often requires a *context switch*, whose costs can easily turn out to be unsustainable, especially in the context of *graph processing systems*. Clearly, in order for a process to acquire a given resource, such a barrier must be crossed at least once. However, as soon as a resource stops being useful and used, a *memory pool* allows the process not to return it to the operating system, but rather to keep and manage it internally. In this way, requesting such a resource will not be as expensive as executing a system call, instead it will be a matter of calling a function of the memory pool and managing some of its internal data structures. Indeed, everything happens within the address space of the process itself and there is no need to perform any context switch. Thus, all memory requests end up being centralized and meant to be made to the memory pool.

Every memory request made the memory pool to the operating system is intended to be performed by calling `int posix_memalign(void **memptr, size_t alignment, size_t size)` [55], which allocates `size` bytes aligned according to `alignment` and returns the address of the allocated memory chunk in `*memptr`. Clearly, this makes *CLMem* compilable and runnable only on POSIX-compliant systems, as enforced by synchronization mechanisms used on shared data structures mentioned in Section 3.3 and B.2.

Also, our memory pool is actually a collection of pools, i.e. one pool per allocation method, further divided into one pool per thread. As a matter of fact, we deal with a *shared pools*, which allow for a block owned by a given thread to be freed by any other worker. As a result, each pool is protected by a dedicated lock.

Finally, it is worth noting that the more likely a given memory resource is to be reused, the more beneficial this software abstraction is. For this reason, memory pools usually handle fixed-size blocks; however, this inevitably leads to internal fragmentation. Therefore, we decide to adopt a hybrid strategy consisting of the methods that follows.

A.1 Fixed-Size Page Groups

A *page group* is an array of memory pages, for which contiguous allocation is granted. Therefore, we set the size of a page group so that it can match the size of *L1 cache* and we align its address to multiples of the size of a memory page. Effectively, the aim is not just to catch the right and fair size but to limit evictions: by allocating contiguous address space, we have some guarantees that a page will not evict its next one within a group. As a matter of fact, it boils down to the OS allocation policies establishing whether physical addresses are indeed contiguous, even if they usually end up being so.

From an implementation perspective, when a given thread asks for a page group and its pool turns out to be empty, it requests memory blocks to the operating system in order to fill the pool. In doing so, the process doesn't request just a single page group, rather it makes as many requests as set by a properly tuned constant. In this way, we conservatively ask for a certain amount of page groups, assuming that we will need them later.

It goes without saying that, if the pool is not empty, we pull a page group from there. Similarly, when we free a page group, we push it into the owner's pool.

A.2 Fixed-Size Small Page Groups

A *small page group* is just a different way of calling a memory page, for which contiguous allocation is granted by definition. Therefore, also in this case, we align the address of a small page group to multiples of the size of a memory page.

The memory pool provides two sets of APIs allowing to acquire and release small page groups. The first one operates in the exact same way as it does for page groups, as described in Section A.1. Furthermore, a second interface implements a deferred and cumulative release policy. In detail, when a small page group is requested, as usual we check if it is already available in the pool and possibly request it to the operating system. In any case, before being returned to the user, it is stored in a buffer keeping track of all small page groups currently in use by a given thread which are meant to be released cumulatively. Thus, when the cumulative release method is finally called, every small page group stored in a given thread's buffer is released to the pool and the buffer is eventually cleared. The above policy has been implemented to speed up massive cleaning operations of certain data structures based on small page groups, such as `memoryTrees` mentioned in Section 3.3.

A.3 Variable-Size Allocation

Our memory pool also supports variable-size memory allocation in order to potentially limit internal fragmentation when needed. Thus, it is possible to request a block of memory aligned to multiples of the size of the cache line. At this point, a hybrid policy first attempts a static allocation, and eventually a dynamic allocation.

In detail, given a constant size factor $size_{static}$, there exist cnt_{static} pools, each pool $pool_i$ storing blocks of size $size_i$ such that $size_i = i * size_{static}$, with $1 \leq i \leq cnt_{static}$. Thus, static allocation takes place in case the size $size_{block}$ of the requested block is multiple of $size_{static}$ and $size_{block} < cnt_{static} * size_{static}$. In this case, we try to pull a memory block from $pool_t$, with $t = \frac{size_{block}}{size_{static}}$; if $pool_t$ is empty, we request memory to the operating system.

If conditions for static allocation are not satisfied, we move on to dynamic allocation. Then, the pool turns out to be a hash index that associates a given size with any possible address of a usable block of memory of that size. If we cannot find anything in the pool, we eventually request memory to the operating system.

When releasing a variable-size memory block, its size must be passed to the memory pool. Indeed, by means of this information, it is possible to state whether the block has been allocated statically or dynamically, and thus place it in the correct pool for later usage.

A.4 Explicit Allocation

The explicit acquire method allows us to request to the operating system a contiguous block of memory of any size whose alignment can eventually be set by the user (by default, it is aligned to multiples of the size of the *cache line*). On the other hand, the release method bypasses the pools and returns a given block of memory directly to the operating system. In practice, this interface is deployed for the sole purpose of fulfilling the design principle of operationally centralizing memory management via the memory pool. No pool is actually involved in explicit allocation.

Appendix B

Data Structures Design

The present chapter outlines the work that has been carried out in the domain of data structure design.

Each of the following classes originates in a definite context that requires specific functionalities to support certain algorithms or operations. As a matter of fact, we take a highly specializing approach that aims to identify the essential services required by a given operational context. Eventually, this leads to the possibility of pinpointing possible optimizations in terms of memory resources and computational performance in order to properly tune the backbone of the software system to the specific needs. This chapter also serves as a developer guide, in that it describes the interfaces and operations allowed by each data structure.

Therefore, Section B.1 describes containers operating in a single-threaded context, while Section B.2 outlines thread-safe classes designed for concurrent use.

B.1 Sequential Data Structures

In the following section, we outline containers designed to operate in sequential environments. Section B.1.1 describes two possible implementations of a fixed-size array, while Section B.1.2 delineates a pair of classes implementing variable-size arrays. Finally, Section B.1.3 details the principles behind the implementation of a hash set.

B.1.1 `CLImmutableArray<T>` and `CLIntegerMap<T>`

`CLImmutableArray<T>` implements a finite, fixed-size collection of fixed-size elements, which are contiguously stored in memory. When the object is instantiated, it explicitly asks the memory pool for a given contiguous memory segment (ref. Appendix A.4) of size equal to the maximum number of elements multiplied by `sizeof(T)`. The class exhibits the random read access operator and a *push back* method, allowed by an internal element counter. No checks are performed on the boundaries, and it is not possible to store elements at any position or to request the size of the container. As a matter of fact, the interface forces the user to sequentially insert new objects and requires the information about the size to be stored and managed externally to the class itself.

`CLIntegerMap<T>` also implements a finite, fixed-size collection of fixed-size elements stored contiguously in memory, but it exhibits a slightly different interface. In fact, as the name suggests, the class explicitly provides the *integer-key map* semantics, thus supporting the random access operator not only for reads but also for writes, although no checks are performed on the boundaries during both operations. Therefore, unlike `CLImmutableArray<T>`, the user is able to write at any position. As `CLIntegerMap<T>` internally stores the size of the map, it can also exhibits forward iterators.

B.1.2 `CLList<T>` and `CLResizableArray<T>`

`CLList<T>` implements a finite, variable-size collection of fixed-size objects contiguously stored in memory. Therefore, upon object creation, we ask the memory pool for a block (ref. Appendix A.3) of a given size representing the resize factor of the list. Then, we keep track of such a factor along with the capacity of the list and the thread owning the block. The class exhibits a *push back* method that, via a counter keeping track of the last appended element, allows us either (*i*) to store the element directly at a proper address or (*ii*) to resize the list in case we have filled the available memory. This is achieved by asking for a new block of the same size as the current one plus a resize factor, so that we can copy the array of objects to the new memory address and release the source to the memory pool. Finally, we can keep track of the owner of the new block. As a matter of fact, we have performed a memory reallocation, which now allows us to insert the required item at a valid address. Those features come at the additional cost of checks on

indexes and list size for each insertion. `CLList<T>` also exhibits random read access operator and a *clean* method which keeps the allocated memory but sets the element counter to 0. Furthermore, we can get forward iterators and check whether a given object is contained in the list with linear complexity in the number of elements.

`CLResizableArray<T>` is very similar to `CLList<T>`, as it also implements a finite, variable-size collection of fixed-size objects contiguously stored in memory. On the other hand, it exhibits the same *integer-key map* semantics as in `CLIntegerMap<T>` outlined in Section B.1.1: as a consequence, rather than appending a new object, we insert it at a given position. If such a position ends up being beyond the boundaries defined by the size of the current allocated memory chunk, we perform reallocation following the same policies described above. However, suppose that we have an object of class `CLResizableArray<T>` with resize factor $resize_f$; therefore, each memory segment of size $resize_f$ can host $resize_e = \frac{resize_f}{\texttt{sizeof(T)}}$ consecutive elements. Then, suppose that we want to insert a new element at position *pos* such that $\frac{pos}{resize_e} > 1$. As a result, we may end up with several empty positions between *pos* and the last element inserted into the array. Essentially, by employing this class, the users account for the possibility of sub-optimal memory usage or, equivalently, they take responsibility for using every slot in the array.

B.1.3 `CLSet<T, Hash, KeyEqual>`

`CLSet<T, Hash, KeyEqual>` implements an associative container holding a set of unique objects of type `T`. The elements are organized into buckets according to their hash value computed by `Hash` (`std::hash<T>` by default), while their identity is determined by `KeyEqual` (`std::equal_to<T>` by default).

Internally, the buckets are an abstraction built over *small page groups* (ref. Appendix A.2). Specifically, when the container is instantiated, we ask a small page group to the memory pool and we store its address and its owner in two objects of class `CLResizableArray<T>` (ref. Appendix B.1.2), which we call `allocated_buckets` and `buckets_owners` respectively. A small page group is then divided into an array of b buckets such that, for a given *elem* of type `T`, we can find its bucket by `Hash(`*elem*`)` mod b. So, if a small page group has size sz_{small_PG}, each bucket will have size $sz_{bucket} = \frac{sz_{small\_PG}}{b}$ and contain $max\_elem_{bucket} = \frac{sz_{bucket}}{\texttt{sizeof(T)}}$ elements. Furthermore, we keep track of the number of elements num_elem_i contained in each bucket, with $1 \leq i \leq b$.

Therefore, we can insert a new element following two different approaches, i.e. checking for duplicates or assuming that none exist. In both cases, in the actual moment the new element *elem* will be inserted inside bucket $i = \texttt{Hash}(elem) \bmod b$, its position is going to be determined by two pieces of information: (*i*) the selector for the small page group $pos_{sel} = \frac{num\_elem_i}{max\_elem_{bucket}}$ inside `allocated_buckets`; (*ii*) the index $pos_{idx} = num\_elem_i \bmod max\_elem_{bucket}$ of *elem* inside the selected small page group `allocated_buckets[`pos_{sel}`]`. If pos_{sel} is greater than the number of elements contained in `allocated_buckets`, we have practically filled the current bucket and we need to allocate a new small page group which is going to expand every bucket by a factor equal to sz_{bucket}; we append the new small page group and its owners to `allocated_buckets` and `buckets_owners` respectively. Otherwise, we simply insert the new element at $offset = \texttt{allocated\_buckets[}pos_{sel}\texttt{]} + i \cdot sz_{bucket} + pos_{idx} \cdot \texttt{sizeof(T)}$. Eventually, we update any possible counter. Clearly, if we want to check for duplicates, we verify that the set does not contain *elem* beforehand by applying `KeyEqual`. On the other hand, inserting new elements without checks for duplicates clearly threatens to break the whole semantics of sets: in fact, in this case the responsibility is completely delegated to the user.

The internals on the insert operation give many technical insights into the underlying design idea of the class. By iterating over the buckets, the structure also implements union with another set, with the aforementioned optional checks for duplicates. In addition, the interface allows to check for equality between two sets or their disjointedness. Finally, forward iterators are also exhibited.

B.2 Concurrent Data Structures

In the following, we outline relevant internals of thread-safe data strctures. Synchronization mechanisms, such as latches or condition variables, are always implemented with the direct support of the POSIX standard, with the only exception of atomic variables provided by the Standard Template Library (e.g. `std::atomic<T>` defined in `<atomic>`).

Most of the classes described below extend their counterpart operating in a single-threaded environment. Section B.2.1 outlines the implementation of a variable-size array, while Section B.2.2 describes a list-type class. Furthermore, a thread-safe hash set is documented in Section B.2.4. In these cases, we will point to the reference class and highlight any major differences.

On the other hand, a few more containers do not have a sequential counterpart and will be described in appropriate detail. Thus, Section B.2.5 outlines a class implementing a list of fixed-size arrays, while Section B.2.6 describes a hash index with duplicated keys.

B.2.1 `CLResizableArray_TS<T>`

`CLResizableArray_TS<T>` is the thread-safe counterpart of the class `CLResizableArray<T>` outlined in Appendix B.1.2. However, the present data structure is completely lock-free, in order to allow for scalability even in case of extreme contention. In particular, we assume that reads at a given position never occur simultaneously with a write request at the same position. Also, we mention that the *integer-key map* semantics allow us to insert elements at any position. Following this, we underline that the class does not provide any guarantee as to the outcome of two concurrent write requests operating at the same position. Therefore, the user is responsible for neither reading nor writing at any position where another thread might be writing at the same time.

On the other hand, all writes and reads are protected from any possible reallocation of the array. This is achieved through memory barriers implemented by means of atomic variables. In particular, through a meticulous usage of *release-acquire* memory ordering, we enable threads to write and read to addresses that are always valid and ensure that no write is lost.

B.2.2 `CLList_TS<T>`

`CLList_TS<T>` is the thread-safe counterpart of `CLList<T>` outlined in Appendix B.1.2. As for `CLResizableArray_TS<T>` described in Appendix B.2.1, also here we assume that reads at a given position never occur simultaneously with a write at the same position. Indeed, the class has been optimized for reads, as required by a specific operational context; alternatively, each read request would have to wait for each concurrent write to complete, even though these two transactions are operating on conflict-free positions. However, the class guarantees that reads always occur in mutual exclusion with possible resize operations of the list and thus reallocations. Finally, in other words, the class guarantees that each read occurs at a valid memory address, but does not guarantee that the read data is valid. On the other hand, writes always occur in mutual exclusion with each other, and in mutual exclusion with reallocations.

This class is also employed in `CLUnorderedMultimap_TS<T, V, Hash, KeyEqual>` (ref. Appendix B.2.6) to store lists of `V`. As outlined in Section 3.3, such lists are intensively read at the core of P-GAM. Therefore, we can adopt an approach inspired by *copy-on-write* techniques, by enabling the creation of a copy of the list in a separate object of class `CLImmutableArray<T>` (ref. Appendix B.1.1). This way, we require the lock guaranteeing mutual exclusion with reallocations just once and keep it as long as the time span of the copy operation (i.e. `std::memcpy`); thereafter, the copy is completely independent on the original object of class `CLList_TS<T>`. Clearly, this granularity of locks is suitable only in case we don't need to read the most recent status of the list.

B.2.3 `CLResizableArrayList_TS<T>`

`CLResizableArrayList_TS<T>` implements a finite, variable-size collection of fixed-size objects. As we instantiate an object, we ask the memory pool for a block (ref. Appendix A.3) of a given size representing the resize factor of the array. Then, we store the address of such a block and its owner in two objects of class `CLResizableArray_TS<T>` (ref. Appendix B.2.1), which we call `allocated_chunks` and `chunks_owners` respectively; also, we keep track of the resize factor of the array. Differently from `CLList_TS<T>` outlined in Appendix B.2.2, the class supports the *integer-key map* semantics, thus allowing to insert a new element at any position. Therefore, suppose that we have an object of class `CLResizableArrayList_TS<T>` with resize factor $resize_f$, such that each block of size $resize_f$ can hold $resize_e = \frac{resize_f}{\texttt{sizeof(T)}}$ consecutive elements. Also, suppose that `allocated_chunks` has size num_chunks and that we want to insert a new element at position $pos \geq resize_e \cdot num\_chunks$. In this case, we avoid reallocating the entire structure to a bigger memory block by just asking for another chunk of size $resize_f$ to the memory pool, thus appending it to `allocated_chunks`. Essentially, the interface handles the class as if it was an array when in fact it implements a list of arrays, as the name suggests.

Therefore, we no longer require reads to be mutually exclusive with any reallocation, as there is no reallocation in the first place. Reads end up being completely lock-free: however, the guarantees and responsibilities mentioned for `CLList_TS<T>` apply here as well. Writes are also lock-free in case they occur at any position $pos < resize_e \cdot num\_chunks$. So again, the user is responsible for avoiding race conditions in this case. On the other hand, mutual exclusion is guaranteed if concurrent write requests operate at positions

that require additional memory chunks. In this case, it is necessary to wait until there are no other concurrent memory requests that might have already extended the boundaries as needed.

B.2.4 `CLSet_TS<T, Hash, KeyEqual>`

`CLSet_TS<T, Hash, KeyEqual>` is the thread-safe counterpart of `CLSet<T, Hash, KeyEqual>` outlined in Appendix B.1.3, with a reduced interface in order to boost performance where strictly required. In particular, the union, the check for equality and disjointedness between two different sets are not implemented.

As a matter of fact, the class implements only insertion with a simultaneous check for any duplicates. However, locking an entire bucket while reading would be extremely expensive, even more since a writing request always implies a reading request. On the other hand, the absence of locking mechanisms while reading would be dangerous, since not only would we have no guarantee of correctness but we might also end up inserting duplicates which result in wasted memory and loss of set semantics. For this reason, we find a trade-off between minimal synchronization cost and complete information consistency.

In detail, given an object of class `CLSet_TS<T, Hash, KeyEqual>` with b buckets, before inserting an element $elem$ in a given bucket $i =$ `Hash(`$elem$`)` mod b, we atomically read the index $last_i$ of the last element inserted in that bucket. Thus, we check without any kind of lock whether $elem$ is present in bucket i up to $last_i$, by applying `KeyEqual`. If $elem$ has not been found then we acquire a lock in mutual exclusion with any possible concurrent writers. Then, we check again from $last_i$ to the index of the last element inserted in bucket i, which might have been changed. If we don't find any duplicate, we finally insert $elem$ element according to the policies for bucket expansion outlined in Appendix B.1.3. Lastly, we atomically update the index of the last element contained in bucket i and release the lock.

B.2.5 `CLImmutableArrayHeap_TS<T>`

Given a constant number of elements th, `CLImmutableArrayHeap_TS<T>` implements a finite, fixed-size collection of fixed-size e_i elements (with $1 \leq i \leq num\_elem$) which are: (i) if $num\_elem \leq th$, stored contiguously in memory; or (ii) if $num\_elem > th$, split into two subgroups $G_1 = \{e_1, \ldots, e_{th}\}$ and $G_1 = \{e_{th}, \ldots, e_{num\_elem}\}$, both stored contiguously in two separate

memory areas.

As it has been described, this class may not seem particularly interesting. However, it provides a constructor that does not take care of allocating memory itself, but rather receives the addresses of the two memory areas being allocated beforehand. This means that we can allocate memory blocks in an external context whereby we have prior knowledge of the available memory resources and the way we want to manage them, thus handle such blocks through this interface. Indeed, the present class has a central role in realizing the data layout outlined in Section 2.2, as it allows to contiguously allocate fixed-size `CLNode` object and overflowing excess connections into a separate heap area.

By means of an internal counter of class `std::atomic_uint64_t`, `CLImmutableArrayHeap_TS<T>` allows to insert new elements in *push back* mode. Therefore, writes always occur in mutual exclusion and race conditions are prevented by construction. On the other hand, reads occur without checking the state of any concurrent writes or the number of elements currently in the container. Therefore, the user is responsible for avoiding race conditions.

B.2.6 `CLUnorderedMultimap_TS<K, V, Hash, KeyEqual>`

`CLUnorderedMultimap_TS<K, V, Hash, KeyEqual>` implements an unordered associative container, associating a list of values of type `V` with equivalent keys of type `K`. In particular, the list of values is implemented by an object of class `CLList_TS<V>`, outlined in Appendix B.2.2.

The data structure carries out the same concept of bucketing as in `CLSet_TS<T, Hash, KeyEqual>` (ref. Appendix B.2.4), i.e. abstractions on *small page groups*. The only difference consists in the elements being inserted into these buckets, which are actually of type `std::pair<K, CLList_TS<V>*>` (hereinafter `key_list_t`). To further delve into the differences in implementation with `CLSet_TS<T, Hash, KeyEqual>`, we might also state that a small page group is divided into an array of b buckets such that, for a given key k of type `K`, we can find its bucket by `Hash(`k`)` mod b. So, if a small page group has size sz_{small_PG}, each bucket will have size $sz_{bucket} = \frac{sz_{small\_PG}}{b}$ and contain $max\_elem_{bucket} = \frac{sz_{bucket}}{\texttt{sizeof(key\_list\_t)}}$ elements.

In detail, given an object of class `CLUnorderedMultimap_TS<K, V, Hash, KeyEqual>` with b buckets, suppose that we want to append a value val to the list of values associated with a given key k. Thus, we compute bucket $i =$ `Hash(k)` mod b, and we atomically read the index $last_i$ of the last element

inserted in that bucket. Therefore, we check without any kind of lock whether k is present in bucket i up to $last_i$, by applying `KeyEqual` on the attribute `first` of objects of type `key_list_t`. If k has not been found then we acquire a lock in mutual exclusion with any possible concurrent writers. Then, we perform similar checks again from $last_i$ to the index of the last element inserted in bucket i, which might have been changed. If we don't find any duplicate key, we finally insert a new element *elem* of type `key_list_t`, according to the policies for bucket expansion outlined in Appendix B.1.3; here, we also instantiate an object *list* of class `CLList_TS<V>`. So, we store k as `first` of *elem* and a pointer to *list* as `second` of *elem*. Lastly, we append v to *list*, then atomically update the index of the last element contained in bucket i and release the lock. Clearly, if the search for equivalent keys turns out to be successful, we only append the value v to the list associated with the key k already in place.

The interface of the class also exhibits a method allowing to get the list of values associated with a given key. However, this is performed without involving any lock, similar to the aforementioned lock-free checks. Thus, no information is given to the user about which state of the map is being observed. Eventually, if a list of values associated with the key is found, a pointer to an object of class `CLList_TS<V>` is returned, which handles concurrency on its own as described in Appendix B.2.2.