Progressive Querying on Knowledge Graphs

Angela Bonifati
Lyon 1 University, CNRS Liris, IUF
Lyon, France
angela.bonifati@univ-lyon1.fr

Stefania Dumbrava
SAMOVAR/IP Paris, ENSIIE
Evry, France
stefania.dumbrava@ensiie.fr

Haridimos Kondylakis
FORTH-ICS
Heraklion, Greece
kondylak@ics.forth.gr

Georgia Troullinou
FORTH-ICS
Heraklion, Greece
troulin@ics.forth.gr

Giannis Vassiliou
FORTH-ICS
Heraklion, Greece
giannisvas@ics.forth.gr

ABSTRACT

The exact evaluation of queries over knowledge graphs encoded as RDF data has been extensively studied. However, in a wide array of applications, RDF queries do not even terminate, due to performance reasons. Notably, queries on public SPARQL endpoints are oftentimes timed out without returning any results. To address this, we propose a novel solution to the problem of progressive query answering and introduce the PING system that implements it on top of SPARK. In our approach, graph query answering leverages a hierarchical structure, which facilitates effective data partitioning, thus allowing us to reduce the sizes of intermediate results and return progressive answers. Moreover, it allows the RDF query evaluation algorithms to directly locate and access the different hierarchy levels required for query answering. Navigating through the hierarchy levels allows expanding or shrinking query results at different granularities. The extensive experimental study on real-world graph datasets, with varied query workloads, shows PING's effectiveness and efficiency, on both exact and progressive query answering, and its superiority to the most relevant baselines.

1 INTRODUCTION

Knowledge graphs are simple yet powerful abstractions for representing and analyzing semantic relationships between real-world objects. Key challenges that graph ecosystems [40] face are the heterogeneity of their data models and the efficiency of query processing on massive, highly interconnected datasets. While exact query processing on knowledge graphs represented as RDF data has received a lot of attention [50], performance problems are widespread, as shown by empirical analyses of SPARQL query logs, collected from publicly available SPARQL endpoints [6, 10]. Several queries of endpoints, such as Wikidata and DBPedia, are timed out because their evaluation on entire RDF graphs is computationally expensive. Nonetheless, distributed big data infrastructures, such as Spark, have emerged and offer improved performances. Indeed, Spark has been exploited for efficient query answering [2], using partitioning techniques, precomputing joins, and constructing indexes to reduce the amount of data needed for query answering.

The problem. Despite the success of these approaches, for big graphs, users still have to wait a considerable time before they see the first answer to their queries. One of the key reasons behind this is that query answering on interconnected data typically requires loading large chunks of the knowledge graph.

On the other hand, approaches have emerged trying to ensure the termination of queries by introducing restricted servers such as TPF [47], SAGE [32] and SmartKG [7]. However, these require a smart client to perform key operations, such as joins, and shipping intermediate results from the server to the client might require more time to finally evaluate the query. Although progressive query answering could be implemented to some extent by progressively returning the results of a pipelined execution plan, single-pass pipelining might not be always possible (e.g., duplicate elimination and aggregate caclulation).

To the best of our knowledge, no existing approach can return progressive query results to users.

Our solution. We propose a novel approach that uses hierarchical information to efficiently identify the data fragments required to return the first part of the answer and to progressively return the remaining ones, thus enabling **progressive query answering (PQA)**. While such hierarchies have been successfully used to represent RDF graphs as relations [30], *ours is the first work to exploit these to generate fine-grained graph partitions* for progressive query processing and further consumption in big data infrastructures.

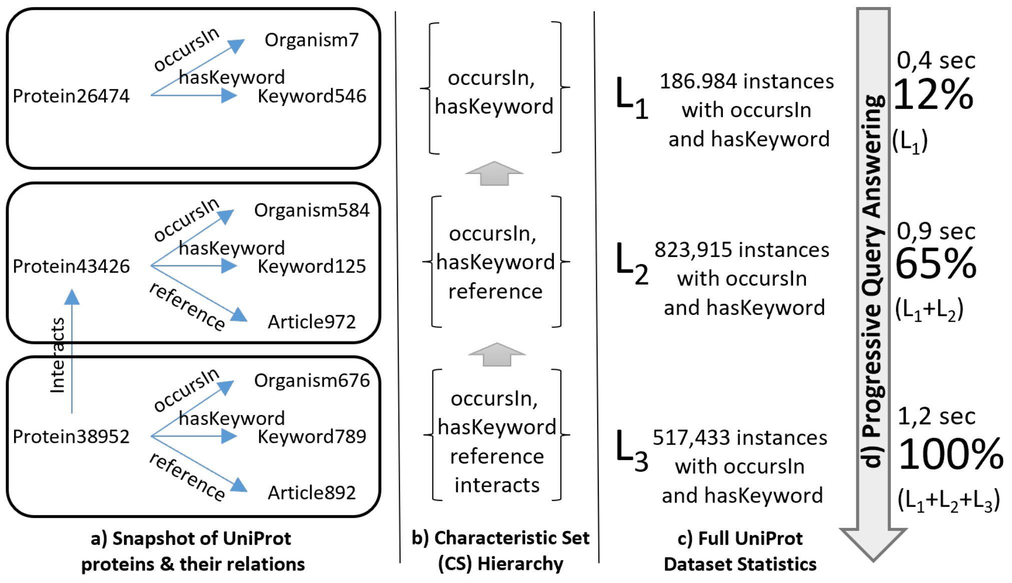
To illustrate the problem at hand, along with our solution, let us consider the following running example.

Example 1. *Fig. 1(a) depicts three example proteins, from the real-world Uniprot*[1] *dataset, together with their relations. Proteins are characterized by numerous properties. For ease of presentation, we focus on four of them: namely, $occursIn$, $hasKeyword$, $reference$, and $interacts$. The hierarchy of the proteins can be overly complex and challenging to navigate, and not all the properties are attached to each protein.*

However, a hierarchical structure (which we name characteristic set (CS) hierarchy as we will explain in Section 3.3) can be computed based on the existing properties (see Fig. 1(b)). Intuitively, this means that the $occursIn$ and $hasKeyword$ properties are specific to a protein, i.e., they always occur, while $reference$ and $interacts$ are supplementary properties, i.e., further refinements that can sometimes be missing. As such, the proteins can be split into partitions L_1, L_2, and L_3 (see Fig. 1(c)).

Assume that in the three levels, we also store other associated instances. Fig. 1(c) shows the number of instances, with both $occursIn$ and $hasKeyword$ properties, that can be progressively loaded - i.e., considering L_1; L_1 and L_2; and L_1, L_2, and L_3 - for a synthetically generated Uniprot dataset of 3GB. Consider the SPARQL query:

[1] https://www.uniprot.org/



Protein26474
occursIn
Organism7
hasKeyword
Keyword546
Protein43426
Organism584
Keyword125
reference
Article972
Interacts
Protein38952
Organism676
Keyword789
Article892
a) Snapshot of UniProt proteins & their relations
occursIn, hasKeyword
occursIn, hasKeyword reference
occursIn, hasKeyword reference interacts
b) Characteristic Set (CS) Hierarchy
L1 186.984 instances with occursIn and hasKeyword
L2 823,915 instances with occursIn and hasKeyword
L3 517,433 instances with occursIn and hasKeyword
c) Full UniProt Dataset Statistics
d) Progressive Query Answering
0,4 sec 12% (L1)
0,9 sec 65% (L1+L2)
1,2 sec 100% (L1+L2+L3)

Figure 1: Example of a `UniProt` instance and its hierarchical partitioning into L_1, L_2, and L_3.

```
SELECT * WHERE {
    ?x occursIn ?b.
    ?x hasKeyword ?d.
}
```

This matches the properties at each level of the partitions, from the top level L_1 to the following levels L_2 and L_3. For progressive query answering, we start returning answers first by only visiting L_1 in just 0.4 seconds. Then, in 0.5 additional seconds, we add more results from level L_2, and, after 0.3 extra seconds, we include the answers from L_3, completing querying for all levels in 1.2 seconds overall. The accuracy (i.e., the percentage of the results only from certain levels divided by the total number of results) of PQA ranges from 12%, at the highest level of abstraction (L_1), to 100%, when considering instances at levels L_2 and L_3. This example gives a high-level overview of the practical usage of PQA and of the trade-off between accuracy and runtime incurred when progressively considering more data.

We present the novel concept of progressive query answering over the computed levels of a characteristic set hierarchy, applicable to both typed and untyped RDF instances. This characteristic set hierarchy is mined, based on the properties of the various instances, to design a multi-level partitioning of the dataset. As such, we regroup, on the same level, all the instances that have the same set of outgoing properties. We then exploit the inclusion relationship of these sets of properties to partition further.

Finally, the multi-layered characteristic set hierarchy is exploited for progressive query answering. PQA, as opposed to exact query answering (EQA), allows for carrying out the query evaluation procedure gradually. We use the query symbols to navigate the characteristic set hierarchy and the induced multi-layered partitioning of the dataset. In exploring the partitioning, we only consider sets of levels that cover all the query symbols, which we call slices, and that can, thus, produce subsets of the total answers. As we visit all possible slices, we iteratively load more levels and return more answers, until the query evaluation is fully completed. Our system (PING) supports the following:

- **Progressive Query Answering.** Queries are evaluated on increasingly larger cumulative partitions, obtained by drilling down from the top (most abstract) level. The results are increasingly more refined and accurate.
- **Exact Query Answering.** The hierarchical partitioning scheme also allows exact query answering. PING can identify more precisely the portions of the data graph that should be loaded for query answering than competitors. As such, exact query answering is more efficient.

By being able to locate and navigate across the CS hierarchy levels, progressive query evaluation algorithms can *strike a balance between accuracy and performance.* To the best of our knowledge, PING is the *first approach that supports progressive graph query answering over CS hierarchies , without requiring an intelligent client, and that can also perform exact query evaluation.* It has been demonstrated in ISWC 2023 [9].

The paper is structured as follows. Section 2 outlines related work on flexible, exact, and approximate query answering. Section 3 presents our partitioning method, while Section 4 highlights its advantages for query answering guided by the CS hierarchy. PING's comparative performance is experimentally evaluated in Section 5. Section 6 concludes and outlines future work.

2 RELATED WORK

As efficient and effective query answering is key to many scientific problems, providing "flexibility" for query answering has been the focus of many works. In this paper, we focus on RDF graphs (for an overview of the domain, see [5]).

2.1 Flexible Query Answering

For semi-structured data, the RELAX [23] operator allows ontology-based relaxation of specified triple patterns, whereas in other approaches [13, 31] query relaxation is based on user preferences. Other works like [19] introduce the APPROX operator, enabling triple pattern replacement by other valid properties. However, we do not focus on flexible query approximation, i.e., on generating and evaluating variants of the original query, but on evaluating the query directly on the knowledge graph. These approaches are complementary to ours.

2.2 Exact Query Answering

For semantic graphs, several works have focused on exploiting extracted schemas and summaries for exact query answering, as surveyed in [12]. S+EPPs [18] exploits bisimulation quotient summaries for summary-based exploration and navigational query optimization. However, their approach focuses on SPARQL navigational extensions, which are beyond our scope. Other works focus on storage layouts [11, 22] and on structural indexes [38, 45] for SPARQL query optimization. Such methods are orthogonal to ours. ASSG [51] builds summaries of the part of an RDF graph that is concerned by a particular set of queries, however, without proper evaluation. Lately, hierarchies are proposed for querying big graphs [17], by identifying regular structures, collapsing these into supernodes, and building a hierarchy of contracted graphs. By contrast, our approach is based on discovering the CS hierarchy of RDF graphs. Also, the authors focus on exact query answering on a single machine, whereas PING can deliver progressive answers, focusing on big, parallel data infrastructures.

2.3 Approximate Query Answering

Approximate Query Processing (AQP) is a well-established area in relational and OLAP databases that focuses on enabling fast analytics on Big Data, by sacrificing some degree of accuracy [4, 27]. AQP techniques are typically based on sampling, data synopses, or a hybrid of the two [28, 34, 35]. For generic graphs, on the other hand, there have been works focusing on approximation algorithms, such as the shortest distance [20], nearest neighbors [49], ranking and binary classification [46], etc.

For semantic graphs, there is a limited amount of work in the field. Progressive query answering, as done by PING, is novel, albeit reminiscent of approximate query processing (AQP). For example, some approaches [3, 14, 15, 43] support answer approximation only for a limited set of analytical queries, returning intermediate approximate results at any time point. Compared to [43], PING's partitioning is not driven by fair-use policies, but by a CS hierarchy, and allows users to control query evaluation. SaGe [3, 32] relies on probabilistic data structures to approximate count-distinct queries in a *single pass*, with strong error guarantees. Unlike SaGe, PING supports evaluation in multiple passes, depending on the trade-off between accuracy and speed users desire. Crucially, unlike AQP approaches, PING guarantees the absence of false positives by construction and allows users to progressively refine answer accuracy.

2.4 Restricted SPARQL servers

Restricted SPARQL servers, e.g., SaGe, TPF [47], or SmartKG [7], ensure BGP queries terminate while preserving SPARQL endpoint responsiveness. However, they require an intelligent client that may introduce additional response time overhead.

The Triple Pattern Fragments (TPF) [47] paginates query results, to avoid server congestion. As such, a page of results can be obtained in bounded time, pushing query processing workload to the client side, but causing the unnecessary transfer of irrelevant data on complex queries with large intermediate results.

SmartKG [7] tries to share the load between servers and clients, while significantly reducing data transfer volume, by combining TPF with shipping compressed KG partitions. Still, this requires an intelligent client, and although compressed, shipping KG partitions introduces additional response time overhead.

Web preemption [3, 32] allows a Web server to suspend a running SPARQL query after a quantum of time and resume the next waiting query. Suspended queries are returned to users, who can re-submit them to continue the execution for another quantum of time. However, it still requires a smart client as the preemptable server only implements a SPARQL fragment; the smart Web client has to implement the missing operators such as joins and projections to recombine the results from the server.

Our approach, however, does not require a smart client. Also, even if the triple store processing is guaranteed to terminate, the size of the data transfer in the previous approaches is high and this might result in a time-consuming execution of the queries.

2.5 RDF Query Answering Using Spark

Some works perform exact query answering on partitions over big data infrastructures such as Spark. Popular ones are SPARQLGX, S2RDF, and WORQ.

SPARQLGX [21], vertically partitions the RDF datasets to increase query answering efficiency, keeping a file for each predicate in the dataset, which only includes domain and range entries.

S2RDF [42] exploits an extended version of the classic vertical partitioning. Each extended vertical partitioning table is a set of sub-tables corresponding to a vertical partition table. The sub-tables are generated by using right outer joins between vertical partitioning tables. For query processing, S2RDF transforms a SPARQL query to an algebra tree, and then it traverses this tree to produce a corresponding SQL query.

WORQ [29] uses a workload-driven partitioning of RDF triples. This tries to minimize the network shuffling overhead based on the query workload, using Bloom filters to determine if an entry in one partition can be joined with an entry in another.

However, all these works adopt simplistic partitioning schemes and fail to exploit multi-level hierarchical partitioning for exact query answering, as we show in the experimental evaluation. Moreover, none of these can perform progressive query answering. *Overall, no other available approach exploits multi-resolution, modular hierarchical structures, for progressive query answering.*

3 HIERARCHICAL PARTITIONING

3.1 High-level architecture

We depict the high-level architecture of our PING system in Figure 2. The framework comprises two main components, implemented on top of Spark: the *partitioner* and the *query processor*. The partitioner processes the initial dataset, extracts its CS hierarchy, and generates hierarchical partitions, as well as sub-partitions and the necessary indexes. For each one of the generated sub-partitions, indexes are created and stored in the Hadoop Distributed File System (HDFS). The query processor leverages these latter structures for PQA.

3.2 Preliminaries

We focus on RDF datasets, a widely-used standard for publishing and representing data on the Web, promoted by the W3C. An *RDF graph* $\mathcal{G}$ (in short a *graph*) is a set of *triples* of the form (s, p, o). A triple states that a *subject* s has the *property* p, whose value is the *object* o. We only consider triples that are well-formed according to the RDF specification [48]. These belong to $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$, where $\mathcal{U}$ is a set of Uniform Resource Identifiers (URIs), $\mathcal{L}$ is a set of typed or untyped literals (constants), and $\mathcal{B}$ is a set

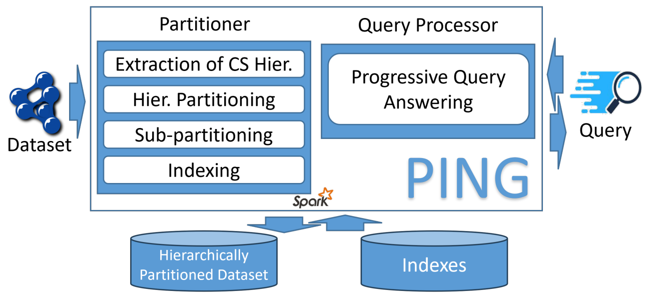


Figure 2: High-level architecture of the PING framework.

of blank nodes. We assume an infinite set $\mathcal{X}$ of variables, where $\mathcal{U}, \mathcal{B}, \mathcal{L}, \mathcal{X}$ are pairwise disjoint. Blank nodes are an essential feature of RDF and represent unknown URIs or literal tokens. The RDF standard also includes the `rdf:type` property, which allows specifying the type(s) of a resource. Each resource can have zero, one, or several types. We will henceforth denote (x, `rdf:type`, Z) as $\tau(x) = Z$.

For querying, we use SPARQL [1], the W3C standard query language for RDF datasets. A SPARQL query q defines a graph pattern P that is matched against an RDF graph $\mathcal{G}$. This is done by replacing the variables in P with elements of $\mathcal{G}$, such that the resulting graph is contained in $\mathcal{G}$. The basic building blocks of SPARQL are *triple patterns*, i.e., elements of $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{X}) \times (\mathcal{U} \cup \mathcal{X}) \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{X})$. A set of triple patterns forms a *basic graph pattern* (BGP). It is commonly acknowledged that the most important aspect for efficient SPARQL query answering is the efficient evaluation of the BGPs [42], on which we focus in this paper, leaving the remaining fragments for future work.

Common types of BGPs are *star* and *chain* queries. *Star* queries are characterized by triple patterns sharing the same variable on the subject position, whereas *chain* queries are formulated using triple patterns where the object variable in each triple pattern appears as a subject in the one immediately succeeding it. We henceforth refer to queries that combine star and chain patterns as *complex*. To define the semantics of SPARQL queries, let us consider the partial function μ that instantiates their variables, i.e., $\mu : \mathcal{X} \rightarrow \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. The evaluation of a BGP q over an RDF graph $\mathcal{G}$ is $q(\mathcal{G}) = \{\mu \mid dom(\mu) = var(q) \wedge \mu(q) \subseteq \mathcal{G}\}$, where $dom(\mu)$ is the subset of $\mathcal{X}$ defining μ and $var(q)$ is the set of variables in q. Finally, let $sym(q)$ be the $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ subset of symbols in q. We use s, p, and o to denote terms (variables or constant symbols) in the subject, property, and object position of triple patterns and t to denote a triple pattern.

3.3 Characteristic Sets

One of the benefits of RDF is its loose structure; one can extend and modify the schema at will, by adding or deleting new triples. Neumann and Moerkotte [33] introduced the notion of a *characteristic set* (CS) to capture the structure of an RDF dataset.

Definition 3.1 (Characteristic Set). For an RDF graph $\mathcal{G}$, the characteristic set of a node s is $CS(s) = \{p \mid \exists o \ : \ (s, p, o) \in \mathcal{G}\}$.

As such, the characteristic set of a node is the set of all (properties), i.e., outgoing edges, attached to it. Such characteristic sets exhibit *hierarchical relationships*, due to overlaps in their comprising sets of properties. *To the best of our knowledge, PING is the first to appropriately leverage the hierarchical structure induced by characteristic sets for semantic graph partitioning.*

We henceforth distinguish between the nodes in the dataset's graph that denote types, which we call *type nodes*, and the rest, which we call *instance nodes*.

Example 2. *The CSs for* `Protein` *instance nodes (Fig. 1) are:*

CS(`Protein26474`) $= \{occursIn, hasKeyword\}$

CS(`Protein43426`) $= \{occursIn, hasKeyword, reference\}$

CS(`Protein38952`) $= \{occursIn, hasKeyword, reference, interacts\}$

While these nodes are all of `Protein` *type, i.e.,* τ(`Protein26474`) $= \tau$(`Protein43426`) $= \tau$(`Protein38952`), *note that they all have different characteristic sets, as seen in the example.*

3.4 Extraction of the CS hierarchy

Characteristic sets help extract a CS hierarchy $\mathcal{H}$ from existing instance nodes.

Definition 3.2 (CS Subsumption). Given two instance nodes v_1 and v_2, $CS(v_1)$ *subsumes* $CS(v_2)$ when $CS(v_1) \subset CS(v_2)$.

Definition 3.3 (CS Hierarchy $\mathcal{H}$). CS subsumption creates a partial hierarchical ordering, such that if $CS(v_1) \subset CS(v_2)$, then $CS(v_1)$ is a parent of $CS(v_2)$. Formally, a CS hierarchy is a graph lattice $\mathcal{H} = \{V_{\mathcal{H}}, E_{\mathcal{H}}\}$, such that $V_{\mathcal{H}} \subseteq C$ and $E_{\mathcal{H}} \subseteq (V_{\mathcal{H}} \times V_{\mathcal{H}})$, where C is the set of all the CSs.

The key idea is that, based on the connectivity of instances, we construct a CS hierarchy and use it to index and partition the dataset. To do so, we visit all instance nodes of the input graph once, identifying their CSs.

Example 3. *Note that:* CS(`Protein26474`) $\subset$ CS(`Protein43426`) $\subset$ CS(`Protein38952`). *Hence,* $\mathcal{H}$ *will be enriched by those three CSs: the first in level one, the second, in level two, and the last, in level three. Another protein, e.g.,* `Protein67453`, *where we have that* CS(`Protein67453`) $= \{encodes, receivesSignal, reacts\}$, *would also be placed in level one, as there is not any other instance* x *with* $CS(x) \subset CS$(`Protein67453`).

3.5 Hierarchical Partitioning

Let us fix an arbitrary RDF graph $\mathcal{G}$. We also denote the extracted CS hierarchy with $\mathcal{H}$, the induced RDF graph partitioning with L, and with $\mathcal{H}_i$ and, respectively, L_i, their corresponding contents at level i. Based on $\mathcal{H}$, we construct a multi-level partitioning L of $\mathcal{G}$ comprising partitions L_i; these regroup all instances that have the same characteristic set, which belongs to $\mathcal{H}_i$. Note that the assignment to a partition for a dataset instance does not depend on it being typed, but solely on its CS, which always exists. We henceforth use the terms "partition" and "level" interchangeably.

In the resulting *hierarchical partitioning*, the highest (most abstract) level is a coarse-grain representation and the lower levels correspond to refinements of the initial graph. The partitions are computed once, by assigning instances to their respective level, based on their CS. Next, we state the *modularity* and *losslessness* properties of our partitioning, which hold by construction.

Theorem 3.4 (Modularity). *Given a graph $\mathcal{G}$ and the generated CS hierarchy $\mathcal{H}$, the result hierarchical partitioning scheme is modular, i.e., $L_i \cap L_j = \emptyset$, for all $i, j \leq |\mathcal{H}|$.*

Proof. The CS of a dataset instance is unique by construction. Hence, each instance will be assigned to a unique hierarchy level. This implies hierarchy levels cannot overlap and, thus, that the CS hierarchy itself is modular. □

Theorem 3.5 (Losslessness). *Given a graph $\mathcal{G}$ and a CS hierarchy $\mathcal{H}$, the result partitioning scheme is lossless, i.e., $L = \bigcup_{i \leq |\mathcal{H}|} L_i$.*

Proof. *Each* dataset instance has a CS and is assigned a specific partition based on it. Due to this, no information is lost by construction with our approach. □

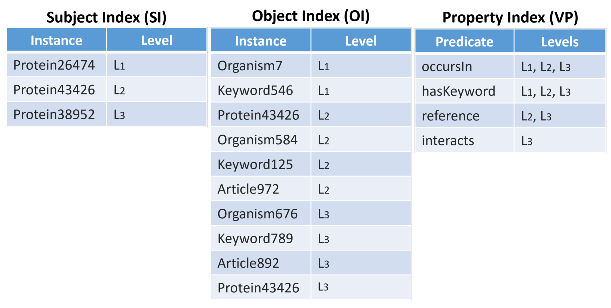
3.6 Sub-Partitioning

As described, the CS hierarchy is used to assign instances to a specific level, i.e., to a specific partition. On top of this, we also implement, for each partition, a vertical partition ($\mathcal{VP}$) step, called *sub-partitioning*, to further reduce the size of the data touched at query answering. For this, we split the triples of each partition L_i, into multiple vertical partitions $L_i[p]$, one file per property p. The vertical partitions are stored as parquet files in HDFS. Each vertical partition contains the subjects and the objects for a single property, enabling a more fine-grained selection of data at query time. Consequently, when looking for a specific property, we do not need to access the entire data of the level storing instances with this property, but only the specific sub-partition at that level with the related property. As shown in Section 5, this minimizes data access, leading to faster query execution times.

3.7 Indexing

To speed up query evaluation, we generate custom indexes, so that the necessary level sub-partitions can be directly identified during query execution. As such, we leverage the CS hierarchy to construct property, subject, and object indexes ($\mathcal{VP}$, $\mathcal{SI}$, and $\mathcal{OI}$, respectively). Specifically, as our partitioning approach is based on the hierarchy of CSs, which includes the corresponding sets of their properties, initially, we index for each property the partitions it is primarily assigned to ($\mathcal{VP}$). For each instance, we also index the partition in which it is located when it is in a subject position ($\mathcal{SI}$) and an object position ($\mathcal{OI}$) respectively. Thus, we can directly identify to which partitions each such instances belong. The aforementioned indexes are stored in HDFS and are loaded in the main memory of Spark as soon as the query processor is initialized.

Example 4. *In Fig. 3 we present the $\mathcal{SI}$, $\mathcal{OI}$, and $\mathcal{VP}$ indices constructed in Fig. 2. For example, we record, among others, that* `Protein26474` *is located on L_1, in the $\mathcal{SI}$ index, and on L_1 and L_2, in the $\mathcal{OI}$ index. Also, we can access the levels on which each property occurs with the $\mathcal{VP}$ index. For example, in the $\mathcal{VP}$ index, we record that occursIn appears on L_1, L_2, and L_3, whereas the* `Protein43426` *is located in L_2, as a subject, and* `Keyword789`*, in L_3, as an object.*



Subject Index (SI)

Instance	Level
Protein26474	L1
Protein43426	L2
Protein38952	L3

Object Index (OI)

Instance	Level
Organism7	L1
Keyword546	L1
Protein43426	L2
Organism584	L2
Keyword125	L2
Article972	L2
Organism676	L3
Keyword789	L3
Article892	L3
Protein43426	L3

Property Index (VP)

Predicate	Levels
occursIn	L1, L2, L3
hasKeyword	L1, L2, L3
reference	L2, L3
interacts	L3

Figure 3: Indexes available for our running example.

3.8 Partitioning Algorithm

Algorithm 1 presents the overall partitioning. Initially, we construct the CS hierarchy $\mathcal{H}$ (line 2). Then, for each triple (lines 3-4), we identify the hierarchy level it should be assigned to based on its subject's CS (line 5). Next, we build the layering of our dataset: we collect on the same partition all the instances with the same CS hierarchy level and update the computed partitioning (lines 6-7) adding the corresponding triples of instances located in the subject position. On individual levels, for each property of its instances, we add the corresponding triple parts (domain and range) into the proper sub-partitions, named after the property (lines 8-9). Finally, we add the location of the subject, object, and property to the three indexes (lines 10-12).

The algorithm needs to do one pass over all triples in order to calculate the CS hierarchy in line 2 and then another pass to assign the instances into the various partitions/sub-partitions and generate the necessary indexes. The two steps can be cleverly combined and be performed into one pass over the data by keeping appropriate pointers in the main memory and swapping them when needed. Hence, the complexity of the algorithm is linear, i.e., $O(n)$ where n is the number of triples in the graph.

Further as all triples are allocated into a single partition and subpartition, no space overhead is introduced for the raw dataset. In fact, as in the sub-partitioning phase, we remove the property names from the triples - the file storing that sup-partitioned is named after that property, and the overall required space is reduced. $\mathcal{VP}$, $\mathcal{SI}$, and $\mathcal{OI}$ indexes are common in triple stores and introduce minimal space overhead in addition –for each resource we need to keep its number and a few numbers with the partitions it exists.

Note that although an instance might have multiple types, it will always have only one CS and, hence, be uniquely assigned to a level/partition. We consider a typing relationship as yet another relationship, not differentiated from all others. As such when typing information exists for a specific instance, this information is also added to the partition where that instance is allocated. If no such information exists for the instance, no additional typing information is required.

This partitioning scheme has two key benefits. First, instances with the same CS are assigned to the same level and then subpartitioned according to their properties, highly minimizing the amount of data that the queries targeting it have to load. Second, it enables PQA, as volumes of data, spanning various CSs, are distributed in different partitions.

4 PROGRESSIVE QUERY ANSWERING

Next, we explain how PING performs progressive query answering. Let us fix a set of levels L that partitions $\mathcal{G}$, as presented in Section 3, and a query q. The main idea behind PING is that it

Algorithm 1 Partitioning($\mathcal{G}$)

Input: $\mathcal{G}$: a graph dataset;
Output: L: a set of levels; $\mathcal{VP}, \mathcal{SI}, \mathcal{OI}$: vertical partitioning, subject, and object indexes

```
1:  $L \leftarrow \emptyset, \mathcal{VP} \leftarrow \emptyset, \mathcal{SI} \leftarrow \emptyset, \mathcal{OI} \leftarrow \emptyset$
2:  $\mathcal{H} \leftarrow calculateCSHierarchy(\mathcal{G})$        ▷ Extraction of CS hierarchy
3:  for all $t \in \mathcal{G}$ do
4:      $(s, p, o) \leftarrow t$
5:      $i \leftarrow getHierarchyLevel(\mathcal{H}, CS(s))$
6:      $L_i \leftarrow \{(s, p, o) \in \mathcal{G} \mid CS(s) = i\}$
7:      $L \leftarrow \{L_i\} \cup L$                              ▷ Partitioning
8:      for all $s, o, p \in L_i$ do
9:          $L_i[p] \leftarrow L_i[p] \cup \{s, o\}$              ▷ Sub-Partitioning
10:         $\mathcal{VP}[p] \leftarrow \{i\} \cup \mathcal{VP}[p]$        ▷ $\mathcal{VP}$ Indexing
11:         $\mathcal{SI}[s] \leftarrow \{i\} \cup \mathcal{SI}[s]$        ▷ $\mathcal{SI}$ Indexing
12:         $\mathcal{OI}[o] \leftarrow \{i\} \cup \mathcal{OI}[o]$        ▷ $\mathcal{OI}$ Indexing
13:     end for
14: end for
15: return $L, \mathcal{SI}, \mathcal{OI}, \mathcal{VP}$
```

exploits precomputed indexes to identify, and gradually visit, the levels in L that should be accessed for query answering. As long as *all the symbols* in q appear on a given level, L_i, this can be used to partially answer q. This is reflected in the key definition of query *safety* given below.

Definition 4.1 (Safety). A symbol r is *safe* on a set of levels S from L if it occurs on at least one level. A triple t is *safe* on S if all its symbols are safe on S. A query q is *safe* on S if all its triple patterns are also safe.

Definition 4.2 (Slice). We call a set S of sub-partitions from L a *slice* for a query q, a symbol r, or a triple t, if these are safe on S. S is a *minimal* (respectively, a *maximal*) slice, if exists no slice S' exists, such that $S' \subset S$ (respectively, $S \subset S'$).

Leveraging slicing and, hence, respecting safety, we can produce partial answers, by only focusing on specific levels. Note that due to the safety property PING imposes, a query is only evaluated on a subset of its slices. This ensures that any partial results returned during PQA are still *exact*, i.e., correspond to subsets of the full result. In particular, each tuple outputted by PQA is not partial, but a valid answer to the original query.

We henceforth fix a query q and denote its slices S and S'.

Lemma 4.3 (PQA Monotonicity). *If $S' \subseteq S$, $q(S') \subseteq q(S)$.*

Proof. By monotonicity of the core SPARQL fragment we consider, i.e., covering select, project, join, and union. This, in turn, follows from the monotonicity of the corresponding relational algebra fragment [39]. □

Lemma 4.3 thus tells us that the evaluation of a query can be performed gradually, on a set of its slices, and that it leads to *increasingly more accurate results*, the more of these we consider. The *soundness* of PQA on every slice, i.e., the fact that we only obtain (subsets of) correct results, holds by the lemma below.

Lemma 4.4 (PQA Boundedness). *$q(S) \subseteq q(\mathcal{G})$.*

Proof. By construction and definition of query safety. Since S is a slice for q, it follows that q is safe on S and, hence, that the query can be evaluated on S since all its triples belong to it. As $S \subseteq L$ and L is a partition of $\mathcal{G}$, we have that $S \subseteq \mathcal{G}$. Given that q is monotonous, by Lemma 4.3, $q(S) \subseteq q(\mathcal{G})$. □

Considering the *entire* set of slices for the query, i.e., its maximal slice, we obtain its exact, lossless evaluation. As such, PING can also be used for EQA.

Theorem 4.5 (EQA Soundness and Completeness). *It holds that $q(S) = q(\mathcal{G})$, where S is the maximal slice for q.*

Proof. As we have showed the query evaluation in our setting is monotonous (Lemma 1) and bounded (Lemma 2), PQA admits a fixed point [44]. Following the fixed point semantics [37] of our SPARQL fragment, this is the unique minimal answer. □

Algorithm 2 captures the PQA of a query q over a hierarchical partitioning of a graph $\mathcal{G}$. We iterate over all the triple patterns in q and inspect all their symbols. Depending on whether they correspond to a predicate or to a subject or object constant, we inspect the corresponding index structures and collect the set of all sub-partitions where the instances are located.

We take the intersection of all such sub-partition sets and compute the minimal slice of the triple, i.e., the corresponding minimal set of duplicate-free partitions in the levels that cover its symbols (lines 2-3). Using this, we determine the set of all slices of q, i.e., the sets of sub-partitions that contain *all* of its symbols. Hence, we iterate over the cartesian product of the individual triple pattern slices (line 5). For every element, we take the union of its levels and build each query slice S (line 6). We then call EQA (i.e., Algorithm 3) and add S to the set of visited levels C (line 7).

Algorithm 2 PQA($\mathcal{G}, L, q, \mathcal{VP}, \mathcal{SI}, \mathcal{OI}$)

Input: $\mathcal{G}$: graph; L: $\mathcal{G}$ partitioning (set of levels); q: query; $\mathcal{VP}, \mathcal{SI}, \mathcal{OI}$: vertical partitioning, subject, and object indexes
Output: Φ – the answers to q

```
1:  $\Phi \leftarrow \emptyset, C \leftarrow \emptyset, \mathcal{I} \leftarrow \mathcal{VP} \cup \mathcal{SI} \cup \mathcal{OI}$        ▷ initialize
2:  for all $t \in q$ do                        ▷ compute triple pattern slices
3:      $HL(t) \leftarrow \bigcap_{r \in sym(t)} \{L_i[j] \mid i \leq |L|, j \in \mathcal{I}[r]\}$
4:  end for
5:  for all $S_t \in \bigtimes_{t \in q} HL(t)$ do
6:      $S \leftarrow \bigcup_{t \in q} \{l \mid l \in S_t\}$                    ▷ query slice
7:      $\Phi \leftarrow \Phi \cup \text{EQA}(S, q, C)$        ▷ accumulate query answers
8:      $C \leftarrow S \cup C$                        ▷ mark slice as visited
9:  end for
10: return $\Phi$
```

Example 5. *To illustrate PQA, consider the query q:*

```
SELECT * WHERE { ?x occursIn ?b.
    ?x hasKeyword <Keyword789>.
    ?x interacts ?y. }
```

To evaluate it, we inspect each triple pattern, identifying the corresponding levels/partitions and sub-partitions indicated by our indexes and properties. For the first triple, T_0, as the property occursIn *appears on L_1, L_2, L_3, we have $HL(T_0) = \{L_1[\text{occursIn}], L_2[\text{occursIn}], L_3[\text{occursIn}]\}$. For the second triple, T_1, we consider the property* hasKeyword *and the symbol* Keyword789*. We know that set of levels for* hasKeyword *is $\{L_1, L_2, L_3\}$, according to $\mathcal{VP}$ index, and that the one*

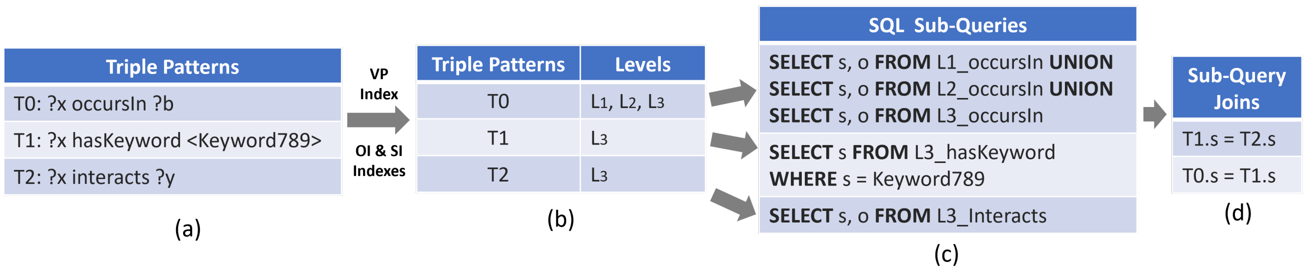


Figure 4: Computing the EQA of q on its maximal slice.

for `Keyword789` *is* $\{L_3\}$*, according to the* $\mathcal{OI}$ *index. As we choose the intersection of the set of sub-partitions, we have* $HL(T_1) = \{L_3[\texttt{hasKeyword}]\}$*. For* T_2*, we have that* $HL(T_2) = \{L_3[\texttt{interacts}]\}$*, since the set of levels for* `interacts` *is* $\{L_3\}$*. The set of all query slices HL is thus:* $HL(T_0) \times HL(T_1) \times HL(T_2)$*. For each of its elements, we take the union of their sub-partitions (Algorithm 2, line 6) and pass the resulting updated slice for EQA.*

Algorithm 3 implements EQA. For a slice S, it loads its unvisited levels (lines 2-3), calls query evaluation (line 4), and returns the result (line 5). We illustrate how PING progressively computes query answers by sequentially calling EQA.

Algorithm 3 EQA(S, q, C)

Input: S: slice (set of levels); q: query; C: visited set of levels
Output: Φ – the answers to q

```
1: $\Sigma \leftarrow \emptyset$                         ▷ initialize
2: for all $L \in S \setminus C$ do     ▷ iterate over unvisited slice levels
3:     $\Sigma \leftarrow \Sigma \cup L$                  ▷ build cumulative slices
4: end for
5: $\Phi \leftarrow q(\Sigma \cup C)$
6: return $\Phi$
```

Example 6. *Revisiting our running example, since our accumulator C is empty the first time EQA is called, we first compute a first partial answer considering only the slice* $L_1[\texttt{occursIn}] \cup L_3[\texttt{hasKeyword} \cup L_3[\texttt{interacts}]$ *and adding it to C. In the next iteration of PQA, the EQA algorithm is called on the slice additionally containing the unvisited* $L_2[\texttt{occursIn}]$*. We complete the procedure by adding the only unvisited sub-partition from the last slice,* $L_3[\texttt{occursIn}]$*, to the accumulated answer and evaluate q on the entire dataset returning all answers. Note that Algorithm 3 will terminate by being called on the* maximal *slice of q, as we iteratively accumulate (in C) all the slices on which q is safe. Note that we can directly compute the EQA of q on* $\mathcal{G}$ *by passing the maximal slice to the algorithm from the start. We illustrate this in Fig. 4. As such, Fig. 4(b) illustrates the vertical partitions and the levels that PING determined should be used. PING leverages these to formulate the SQL sub-queries shown in Fig. 4(c), joining their results (see Fig. 4(d)) and computing the final answers.*

The combined complexity of evaluating a query q on our hierarchical multi-level dataset partitioning L is $O(|P| \cdot (log|P| + \Sigma_{L_i \in L} log|L_i|))$, where $|P|$ is the number of triple patterns, following the complexity of evaluating BGP fragments of SPARQL [36].

Implications for aggregate queries and complex datasets. Notably, the CS hierarchy of a real dataset might be very complex. However, this is a rather positive fact, as the dataset can be split into more partitions, enabling fine-grained PQA. Further, although we focus on BGP queries, downstream aggregate query processing can also benefit from progressive querying, continuously refining the answer as time goes on, progressively improving its quality.

5 EVALUATION

We study the performance and coverage of PING on RDF graphs of various sizes, with diverse hierarchy levels. These highlight the effectiveness of PQA, which strikes a balance between query-answering efficiency and coverage. We also assess the performance of PING on EQA. Note that PING's PQA and EQA capabilities have been demonstrated live on dedicated scenarios [9]. Also, PING's codebase is open source and the datasets and queries used in our experiments are available online[2].

5.1 Setup

The experiments were conducted using a cluster of 4 physical machines running Apache Spark 3.0.0, a popular MapReduce framework. Each machine is equipped with 235GB of memory, 400GB of storage, and 38 cores running Ubuntu 20.04.2 LTS. In each machine, 10GB of memory was assigned to the memory driver, and 200GB was assigned to the Spark worker.

5.2 Datasets & Workloads

To evaluate PING, we used three synthetic datasets (*Uniprot*, *Shop*, and *Social*) of various sizes, obtained using the gMark [8] graph instance and query workload generator, the LUBM synthetic dataset and two large-scale real-world datasets (*DBpedia* and YAGO). Their characteristics are provided in Table 1.

Uniprot encodes the schema of the homonymous dataset, encoding protein sequences and their functional information.

Shop simulates the default schema of the Waterloo SPARQL Diversity Test Suite (WatDiv)[3], with purchased products and the customer information.

Social encodes the fixed schema of the LDBC Social Network Benchmark [16], representing a social network with people and the messages they post along with their likes.

LUBM is a benchmark focusing on university domain.

DBpedia includes version 3.8 of the homonymous dataset, and *YAGO* is a database with knowledge about the real world.

For the three synthetic datasets, we produce 60 queries (20 star, 20 chain, and 20 complex). We generated 2000 queries using

[2]https://github.com/giannisvassiliou/PING-EDBT-2025
[3]https://dsg.uwaterloo.ca/watdiv/

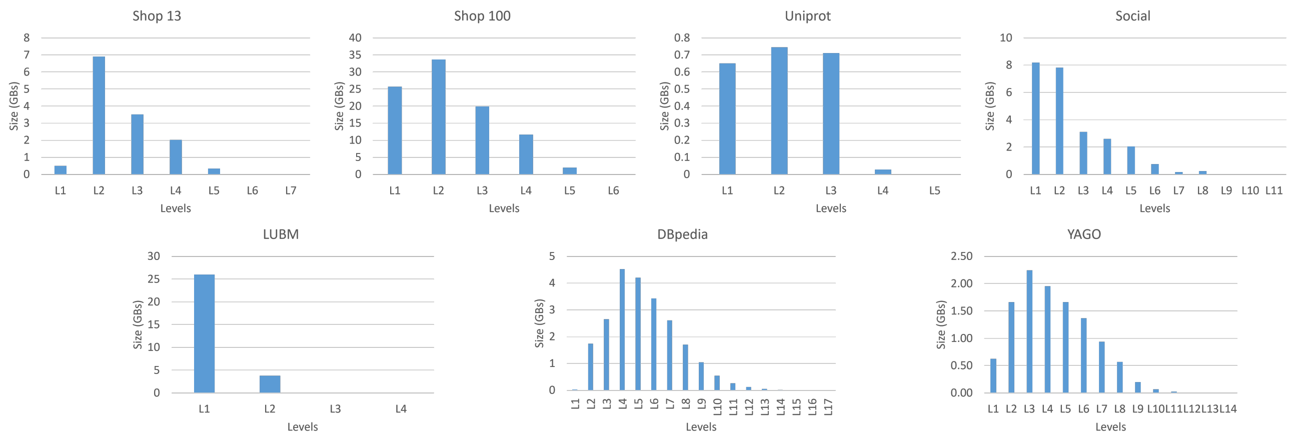


Figure 5: Data distribution across hierarchy partitioning levels for all datasets.

Dataset	Size	Triples	Star		Chain		Complex	
			Min	Max	Min	Max	Min	Max
Uniprot	3GB	2.1M	2	5	2	5	2	5
Shop	13GB	23M	2	5	2	5	3	5
	100GB	1B						
Social	18GB	50M	3	5	3	4	1	5
LUBM	30.1GB	173.5M	2	5	1	2	4	6
YAGO	12GB	82M	3	6	0	0	4	13
DBpedia	30GB	182M	1	5	1	4	4	5

Table 1: Dataset & Query workload characteristics

the gMark benchmark for each category and randomly picked the 20 first that returned an answer when issued to the corresponding endpoint. LUBM comes with 14 benchmark queries (6 star, 3 chain, 5 complex). For YAGO, we retrieved 15 queries (4 star, 11 complex) previously used for benchmarking in [29]. Although in those categories plain chain queries do not exist, large chains are evaluated within the complex queries. For DBpedia, similarly to gMark, we randomly selected 60 BGP (20 star, 20 chain, 20 complex) queries, using the FEASIBLE benchmark generator [41], based on real-world query logs. The query workload characteristics are shown in Table 1 in terms of their minimum and maximum number of triple patterns.

5.3 Competitors

As PING is the first to implement progressive, multi-resolution query answering for RDF datasets, we do not have direct competitors. However, in the extreme case that our system is used for exact query answering, we compare our approach with other representative systems focusing on exact query based on Spark, i.e., S2RDF v1.1 [42] and WORQ v0.1.0 [29]. We have chosen these since, in their respective papers, these **have been shown to greatly outperform other state-of-the-art competitors, i.e., SHARD, PigSPARQL, Sempala, and Virtuoso Open Source Edition v7** [42].

S2RDF uses Extended Vertical Partitioning, whereas WORQ uses Bloom filters on top of vertical partitioning to efficiently reduce data access for query answering. For a fair comparison, in both systems, we disabled caching of precomputed joins, as this is orthogonal to data partitioning and indexing, studied in this paper. All systems included in the comparison, i.e., PING, S2RDF, and WORQ only accept BGP queries for evaluation, whereas all of them use Parquet files for storing the data. Finally, a time-out of twenty-four hours was selected, i.e., after this time lapse without finishing the execution, each experiment was stopped.

5.4 Metrics

We use the following evaluation metrics.

Query execution time: We evaluate the efficiency of the various configurations of our algorithm.

Data access: We analyze the rows that should be accessed to perform query answering.

Coverage: As we partition the initial graph, when the loaded levels do not amount to the maximal slice of a query, we lose information when we evaluate it. We use the following formula to measure coverage: $\frac{|q(S)|}{|q(\mathcal{G})|}$, where $|q(S)|$ and $|q(\mathcal{G})|$ denote the number of answers obtained when evaluating q on a set of levels, up to and including, the slice S and, respectively, on $\mathcal{G}$.

Further, when our system is used for EQA, apart from query execution time and data access, we also use the following metrics to compare its performance.

Preprocessing time: We evaluate the efficiency of the algorithms for building the (sub)partitions and indexes.

Reduction factor: We evaluate the space that each system outputs and uses in terms of bytes for the compressed Parquet files. The reduction factor is equal to the size of the partition (S) produced by a system divided by the size of the initial dataset.

In calculating the aforementioned metrics in each case, we report an average of 10 executions.

5.5 Results on Progressive Query Answering

5.5.1 Results on Data Distribution. The distribution of the datasets in the various levels is shown in Fig. 5. As shown for the most synthetically generated datasets, the CS hierarchy has 5 − 7 levels, whereas we have 11 for *Social*, 17 for *DBpedia*, and 15 for *YAGO*. Regarding the spread of triples across levels, we notice a great variability, which is, however, dataset-specific. Note that gMark allows us to control the characteristics of the generated instances and query workloads. Hence, our benchmarks are structurally diverse and provide interesting use cases.

5.5.2 Results on Query Execution. Next, we present the results on progressive query answering for the various datasets we use.

Figure 6 shows the runtime, loaded data amount, and coverage of star, chain, and complex queries, varying the number of slices

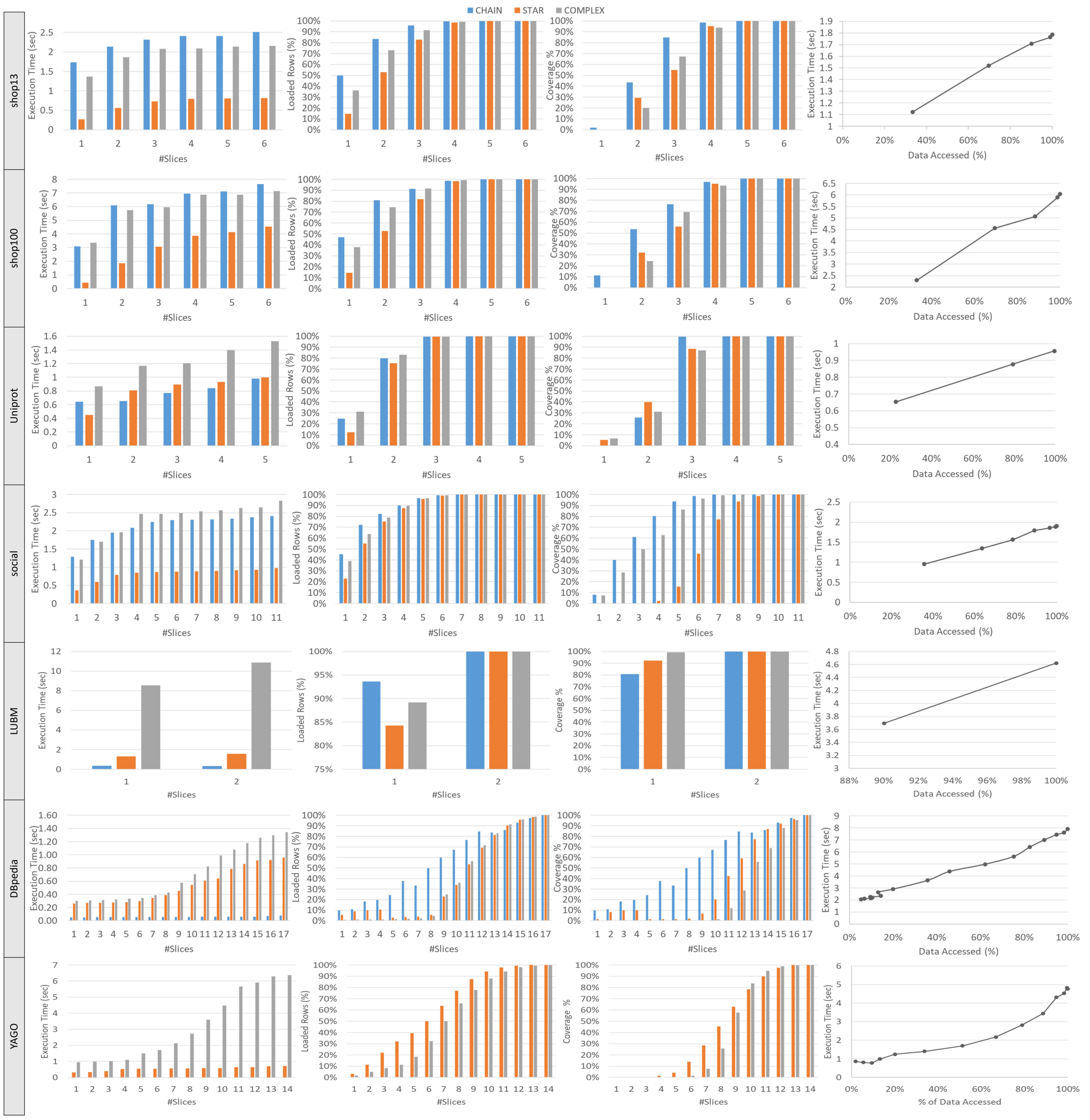


Figure 6: Results on PQA runtime, loaded rows and coverage of the various datasets.

used to answer them, as well as execution time as more data are loaded on our datasets. We discuss the results per dataset.

Shop. According to Fig. 6, the more slices we visit, the more the execution time increases. This is in line with the data access trends. Similarly, the more slices we visit, the more the coverage improves. However, at the fifth slice, we achieve 100% coverage, thus avoiding us visiting the last slice. We also observe different behaviors depending on the query type: chain queries are almost completely answered by visiting four slices, on which star queries are answered much faster than other query types, respectively chain and complex queries. These tendencies hold when scaling to 1 billion triples (shop100). The larger dataset requires more execution time; everything else is similar to the 13GB version.

Uniprot. For the UniProt dataset, we observe similar trends, despite now having only five slices. We record full coverage already on four slices and evaluate chain (and star) queries faster.

Social. For the Social dataset, we need to inspect 10 slices (out of 11) to reach 100% coverage, where execution time stabilizes after four slices. Although Social contains more levels, we can still reach full coverage after a few slices.

LUBM. LUBM generates instances consistently; as such, we only have two levels, which are both required in order to reach 100%

coverage for all queries. Nevertheless, we can retrieve the results from the first results faster, showing that even in highly structured datasets PQA has significant benefits.

DBpedia. DBpedia includes many instances without a predefined schema, as triples are introduced by various users. Hence, it exhibits includes numerous levels (more than double that in other datasets). We observe that chain queries reach higher coverage values and run faster than the other query types. Compared to other datasets, to achieve coverage close to 100%, PING needs to visit almost all the 17 slices, with a comparable increase in execution times and loaded rows. Also, chain queries in DBpedia are typically small (one to four triple patterns), as also confirmed by previous analysis of real-world logs [6]. Their execution time stays steady across PQA, even with many slices. This is not the case for star and complex queries on DBPedia, whose execution time increases with the number of levels. Conversely, chain queries quickly access a large percentage of the overall number of rows needed for EQA. This results in a significant increase in the percentage of data loaded at the early slices and in coverage, compared to star and complex queries.

YAGO. Similarly to DBpedia, YAGO is also another real dataset with many levels (14). The benchmark queries were significantly larger than other datasets (7 triple patterns on average), but we can observe a similar tendency as DBpedia queries. Star queries are really fast when compared to complex ones, despite the fact that they load a significant number of triples, where both categories require level 13 to be almost 100% answered.

Average execution time as returned data increases. In Fig. 6 we also plot the average execution time as the data accessed for query answering increases. In all datasets, we observe the benefits of our progressive approach, as it can guarantee that the query execution time increases linearly with the size of the data.

5.6 Results on Exact Query Answering

In the extreme case that PING is used for exact query answering, as already mentioned, we compare with WORQ and S2RDF.

5.6.1 Preprocessing Time. The times required for preprocessing the various datasets and systems are presented in Fig. 7.

For PING, the time scales based on the complexity and the size of the datasets and ranges between 8 minutes, for the smallest dataset (Uniprot), to 273 minutes, for the most complex one (DBpedia). For the same dataset (Shop), its largest version requires significantly more time, as more triples have to be examined and placed in the respective partitions. In real datasets (DBpedia and YAGO), high variations in the instances generate far more CSs than the synthetic ones, leading to large partitioning times.

Competitors also require a significant preprocessing time for partitioning and indexing the datasets. In most cases, PING is considerably faster than competitors, except for the smallest one (Uniprot) and the most structured one (LUBM). Both S2RDF and WORQ fail to process the most complex dataset (DBpedia), timing out after one week. However, partitioning is only executed once and offline before query answering, and, as such, is transparent to the users.

5.6.2 Reduction Factor. Fig. 7, at the bottom, presents the reduction factor for the various systems. WORQ adopts a dictionary compression policy for data storage and, as such, the resulting Parquet files occupy a small fragment only of the initial file with a reduction factor ranging between 0.27 and 0.42. S2RDF introduces additional extended vertical partitions and as such in most

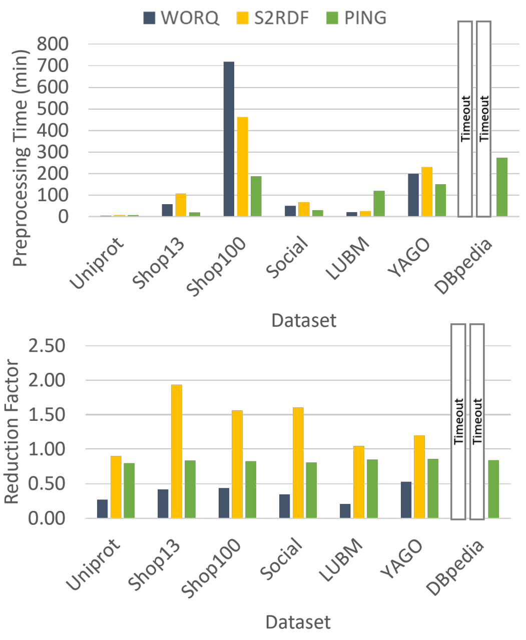


Figure 7: Preprocessing time and reduction factor.

cases requires additional storage, reaching a reduction factor of up to 1.94 of the initial dataset for Shop13. PING, however, adopts a sub-partitioning approach, (i.e., a vertical partitioning inside the partitions) that minimizes space, since predicates are omitted from the generated vertical partition tables (i.e. the predicates). Hence, the reduction factor is always smaller than 1, ranging from 0.79 to 0.83.

5.6.3 Query Execution Time. We report our results for exact query answering, comparing PING with S2RDF and WORQ. For space reasons, we only present results on the dataset with the larger queries (YAGO) and the largest one (Shop 100), as the trends are similar to the other ones. In Fig. 9, we report the runtimes of query execution and the number of triples that had to be accessed for both datasets.

YAGO. For YAGO we again use all 14 benchmark queries and report the average of five executions. Unfortunately, all queries are quite large in terms of the number of triple patterns. As such, they require all levels for query answering, making the benefits of accessing only a few partitions less apparent. Nevertheless, real user queries are usually smaller [10] and, as shown in Fig. 9, PING outperforms WORQ in all cases, and has a similar performance to S2RDF. Our system is able to load less triples than S2RDF in order to answer the complex queries, however, precomputing joins gives a small benefit over accessing less triples. Note that join precomputation is orthogonal to our method and could be further leveraged to optimize PING, whereas if we disable join precomputation PING significantly outperforms S2RDF in all queries.

Shop 100. In order to demonstrate the benefits of our system when we query target specific levels in the largest dataset, i.e. the Shop 100, we use the random query generator to select the first five queries targeting a specific number of levels from the 2-6 partitions (in total 25 queries). As long as the queries require accessing the entire set of levels where a specific resource exists,

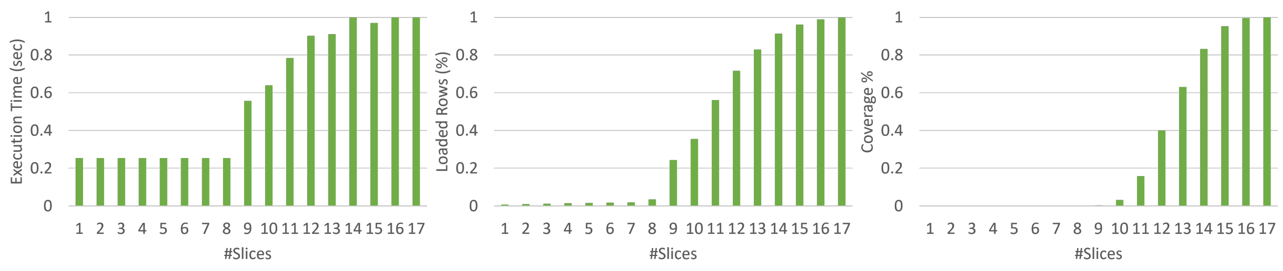


Figure 8: Execution time, loaded rows, and coverage for the visited Q55 slices.

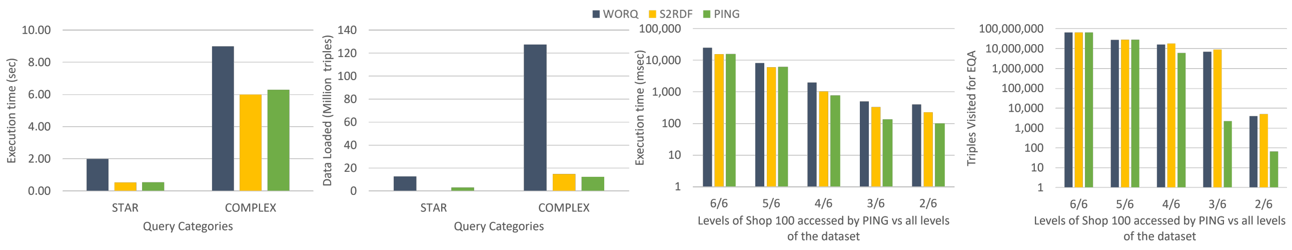


Figure 9: Execution time and triples visited for EQA on *Shop* 100 GB and YAGO.

our partitioning policy, in essence, is reduced to a vertical partitioning scheme - adopted by WORQ and S2RFD. Thus, execution times for queries that require access to the entire set of levels that include the symbols in those queries are similar. Note that the performance of WORQ is always worse than S2RDF, as Bloom filters unsuccessfully try to reduce the visited data. In essence, a minimal query optimization policy, as implemented by S2RDF, is enough to accelerate it (perform small joins first). However, if instances are available in the query, PING is able to focus on the specific levels that include this information and only accesses a subset of the entire vertical partition. This drastically improves query execution efficiency. For example, when PING only accesses two levels out of six, PING is **one order of magnitude faster than both S2RDF and WORQ, visiting two orders of magnitude fewer triples for EQA**.

5.7 Discussion of a real use case from DBpedia

To better illustrate PING's PQA method, we conducted a qualitative study on a real query from DBpedia, i.e., Q55. We have chosen this query since it is a complex query, with four triple patterns, requiring all DBpedia levels (as seen in Fig. 8) for its evaluation. As shown below, the query retrieves the types of companies founded in California and the products they produce:

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
Q55: SELECT * WHERE {
    ?company rdf:type ?company_type.
    ?company dbo:foundationPlace dbr:California.
    ?product dbo:developer ?company.
    ?product rdf:type ?product_type. }
```

To evaluate Q55, PING identifies the levels of its symbols using the available indexes, shown in Table 2. For the first triple pattern, we need to visit all levels. For the second, since dbo:foundationPlace is available in levels 2-13, and dbr:California is available as an object 2-17, we only need to visit levels 2-13. For the third triple pattern, we need to visit levels 2-11, and for the fourth, we need to visit all levels.

Symbol	Levels
rdf:type	1-17
dbo:foundationPlace	2-13
dbo:developer	2-11
dbr:California	2-17

Table 2: Symbol levels of DBpedia's Q55 query.

Next, the algorithm identifies the slices needed for evaluation. For the first slice, considering the first and the last triple pattern, PING loads the L_1[type] vertical sub-partition, for the second triple pattern, the L_2[foundationPlace] sub-partition, and for the third triple pattern, the L_2[developer] sub-partition. As seen in Fig. 8, the selected sub-partitions do not have rows that can be joined, hence PQA coverage on slice 1 is zero. For slice 2, we additionally load L_1[type] ∪ L_2[type], needed for the first and last triples. Again, there are no rows that can be joined, the slices are formulated one by one, and query evaluation is progressive. Slices that do not offer results are quickly skipped, and execution time increases as soon as results emerge, improving the coverage progressively.

Note that the partitioning is fixed and users do not have control over it. However, as outlined in Section 6, it leaves the possibility of returning first any subset of query slices. For example, if one knows that the slice corresponding to level 2 has the most triples, the user could choose it as the starting point for PQA.

Shifting our attention to the results in Fig. 8, the coverage is almost zero for the first 9 slices. This happens as the loaded rows are limited (as shown in the loaded rows diagram of Fig. 8) and they cannot be joined among the tables corresponding to the different predicates. However, after slice 9, more data that gives results is accumulated and the coverage gradually improves. This also requires more execution time.

The qualitative study highlights additional advantages of PQA over (bulk) EQA. Namely, the breakdown of query evaluation per level provides more insights and renders it user-controllable.

6 CONCLUSIONS

We have presented PING, the first system for progressive query answering over KGs that does not use an intelligent client. PING uses a CS hierarchy to partition KGs and progressively evaluate queries. As such, it offers minimal latency and allows trading query accuracy for efficiency. Experiments on synthetic and real-world datasets confirm the flexibility of our solution, which can transform KGs into partitions and progressively evaluate these. Moreover, we show that PING has the potential to dominate competitors, even when used for exact query answering, in several cases being orders of magnitude faster than competitors.

6.1 Limitations

Progressive query answering is only supported for monotonic queries that allow retaining the previously produced results when considering additional levels. Non-monotonic queries (e.g. with negation) are currently not allowed to be progressively evaluated.

Furthermore, PING currently considers that datasets are static and do not regularly update, which might not be true in practice [24–26]. Handling large dynamic graphs would require an incremental update for the existing partitioning scheme, which although trivial for instances that have a CS already in the CS hierarchy, becomes complicated when new levels should be introduced in the hierarchy.

Finally, for aggregate queries, PING would need to inspect all relevant slices to estimate the final accurate result, doing away with the benefits of PQA and requiring an approximate query-answering approach, which is beyond our scope.

6.2 Future Work

We plan to render PING capable of also handling navigational queries involving recursion. In this setting, the evaluation would require multiple iterations across the impacted levels, which might impact the overall benefit of the approach. To handle this efficiently, we will investigate the usage of dedicated optimization techniques. We also intend to explore how our algorithms should be adapted to progressively answer non-monotonic queries, make the partitioning scheme incremental, and render PING amenable to interactive, user-centered KG exploration. The idea is to provide answers based on specific efficiency and accuracy requirements. Exploring orthogonal techniques, such as Bloom filters (to identify levels with relevant answers) and pre-computation of joins (to boost efficiency) is also an interesting direction.

Also, PING currently considers that datasets are static and do not regularly update, which might not be true in practice. As such, we intend to develop an incremental update algorithm for the existing partitioning scheme. Finally, within PING we have explored PQA progressing sequentially, through the first hierarchy levels. However, we could optimize PING to return the largest/smallest partition first, before processing the remaining ones. Although PING does not require a smart client, experimentally comparing it with approaches that do, would further highlight the benefits of our approach.

ACKNOWLEDGMENTS

The work reported in this paper is implemented in the framework of H.F.R.I call “Basic research Financing (Horizontal support of all Sciences)” under the National Recovery and Resilience Plan “Greece 2.0” funded by the European Union – NextGenerationEU (H.F.R.I. Project Number: 16819).

REFERENCES

[1] [n.d.]. W3C Recommendation, SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/. Accessed: 2019-10-09.

[2] Giannis Agathangelos, Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. 2018. RDF Query Answering Using Apache Spark: Review and Assessment. In *34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16-20, 2018*. IEEE Computer Society, 54–59. https://doi.org/10.1109/ICDEW.2018.00016

[3] Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Arnaud Grall, and Thomas Minier. 2022. Online approximative SPARQL query processing for COUNT-DISTINCT queries with web preemption. *Semantic Web* 13, 4 (2022), 735–755. https://doi.org/10.3233/SW-222842

[4] Pritom Saha Akash, Wei-Cheng Lai, and Po-Wen Lin. 2022. Online Aggregation based Approximate Query Processing: A Literature Survey. *CoRR* abs/2204.07125 (2022). https://doi.org/10.48550/ARXIV.2204.07125 arXiv:2204.07125

[5] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *VLDB J.* 31, 3 (2022), 1–26. https://doi.org/10.1007/s00778-021-00711-3

[6] Wim Martens Angela Bonifati and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679.

[7] Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. 2020. SMART-KG: Hybrid Shipping for SPARQL Querying on the Web. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 984–994. https://doi.org/10.1145/3366423.3380177

[8] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869. https://doi.org/10.1109/TKDE.2016.2633993

[9] Angela Bonifati, Stefania Dumbrava, Haridimos Kondylakis, Georgia Troullinou, and Giannis Vassiliou. 2023. PING: Progressive Querying on RDF Graphs. In *ISWC (Posters/Demos/Industry) (CEUR Workshop Proceedings)*, Vol. 3632. CEUR-WS.org.

[10] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 127–138.

[11] Maxime Buron, François Goasdoué, Ioana Manolescu, Tayeb Merabti, and Marie-Laure Mugnier. 2020. Revisiting RDF storage layouts for efficient query answering. In *SSWS@ISWC (CEUR Workshop Proceedings)*, Vol. 2757. CEUR-WS.org, 17–32.

[12] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2019. Summarizing semantic graphs: a survey. *VLDB J.* 28, 3 (2019), 295–327.

[13] Peter Dolog, Heiner Stuckenschmidt, Holger Wache, and Jörg Diederich. 2009. Relaxing RDF queries based on user and domain preferences. *J. Intell. Inf. Syst.* 33, 3 (2009), 239–260. https://doi.org/10.1007/s10844-008-0070-7

[14] Stefania Dumbrava, Angela Bonifati, Amaia Nazabal Ruiz Diaz, and Romain Vuillemot. 2018. Approximate Evaluation of Label-Constrained Reachability Queries. *CoRR* abs/1811.11561 (2018).

[15] Stefania Dumbrava, Angela Bonifati, Amaia Nazabal Ruiz Diaz, and Romain Vuillemot. 2019. Approximate Querying on Property Graphs. In *SUM (Lecture Notes in Computer Science)*, Vol. 11940. Springer, 250–265.

[16] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD Conference*. ACM, 619–630.

[17] Wenfei Fan, Yuanhao Li, Muyang Liu, and Can Lu. 2022. A Hierarchical Contraction Scheme for Querying Big Graphs. In *SIGMOD Conference*. ACM, 1726–1740.

[18] Valeria Fionda, Giuseppe Pirrò, and Mariano P. Consens. 2019. Querying knowledge graphs with extended property paths. *Semantic Web* 10, 6 (2019), 1127–1168.

[19] Riccardo Frosini, Andrea Calì, Alexandra Poulovassilis, and Peter T. Wood. 2017. Flexible query processing for SPARQL. *Semantic Web* 8, 4 (2017), 533–563.

[20] Xinrui Ge, Jia Yu, Hanlin Zhang, Jianli Bai, Jianxi Fan, and Neal N. Xiong. 2022. SPPS: A Search Pattern Privacy System for Approximate Shortest Distance Query of Encrypted Graphs in IIoT. *IEEE Trans. Syst. Man Cybern. Syst.* 52, 1 (2022), 136–150. https://doi.org/10.1109/TSMC.2021.3073542

[21] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. 2016. SPARQLGX in Action: Efficient Distributed Evaluation of SPARQL with Apache

Spark. In *ISWC*.

[22] Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Chen, and Liang Chen. 2017. Stylus: A Strongly-Typed Store for Serving Massive RDF Data. *Proc. VLDB Endow.* 11, 2 (2017), 203–216. https://doi.org/10.14778/3149193.3149200

[23] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. 2008. Query Relaxation in RDF. *J. Data Semant.* 10 (2008), 31–61. https://doi.org/10.1007/978-3-540-77688-8_2

[24] Haridimos Kondylakis and Dimitris Plexousakis. 2011. Exelixis: evolving ontology-based data integration system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 1283–1286. https://doi.org/10.1145/1989323.1989477

[25] Haridimos Kondylakis and Dimitris Plexousakis. 2011. Ontology Evolution in Data Integration: Query Rewriting to the Rescue. In *Conceptual Modeling - ER 2011, 30th International Conference, ER 2011, Brussels, Belgium, October 31 - November 3, 2011. Proceedings (Lecture Notes in Computer Science)*, Manfred A. Jeusfeld, Lois M. L. Delcambre, and Tok Wang Ling (Eds.), Vol. 6998. Springer, 393–401. https://doi.org/10.1007/978-3-642-24606-7_29

[26] Haridimos Kondylakis and Dimitris Plexousakis. 2012. Ontology Evolution: Assisting Query Migration. In *Conceptual Modeling - 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings (Lecture Notes in Computer Science)*, Paolo Atzeni, David W. Cheung, and Sudha Ram (Eds.), Vol. 7532. Springer, 331–344. https://doi.org/10.1007/978-3-642-34002-4_26

[27] Kaiyu Li and Guoliang Li. 2018. Approximate Query Processing: What is New and Where to Go? - A Survey on Approximate Query Processing. *Data Sci. Eng.* 3, 4 (2018), 379–397. https://doi.org/10.1007/S41019-018-0074-4

[28] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. 2021. Combining Aggregation and Sampling (Nearly) Optimally for Approximate Query Processing. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1129–1141. https://doi.org/10.1145/3448016.3457277

[29] Amgad Madkour, Ahmed M. Aly, and Walid G. Aref. 2018. WORQ: Workload-Driven RDF Query Processing. In *ISWC*. 583–599.

[30] Marios Meimaris and George Papastefanatos. 2018. Hierarchical Characteristic Set Merging for Optimizing SPARQL Queries in Heterogeneous RDF. *CoRR* abs/1809.02345 (2018).

[31] Xiangfu Meng, Zong Min Ma, and Li Yan. 2008. Providing Flexible Queries over Web Databases. In *KES (2) (Lecture Notes in Computer Science)*, Vol. 5178. Springer, 601–606.

[32] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. 2019. SaGe: Web Preemption for Public SPARQL Query Services. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 1268–1278. https://doi.org/10.1145/3308558.3313652

[33] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*. IEEE Computer Society, 984–994.

[34] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1461–1476. https://doi.org/10.1145/3183713.3196905

[35] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. 2018. AQP++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In *SIGMOD Conference*. ACM, 1477–1492.

[36] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2006. Semantics and Complexity of SPARQL. In *ISWC (Lecture Notes in Computer Science)*, Vol. 4273. Springer, 30–43.

[37] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[38] François Picalausa, Yongming Luo, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. 2012. A Structural Approach to Indexing Triples. In *ESWC (Lecture Notes in Computer Science)*, Vol. 7295. Springer, 406–421.

[39] Yehoshua Sagiv and Mihalis Yannakakis. 1980. Equivalences Among Relational Expressions with the Union and Difference Operators. *J. ACM* 27, 4 (1980), 633–655.

[40] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, and et al. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[41] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. Feasible: A feature-based sparql benchmark generation framework. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14*. Springer, 52–69.

[42] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB* 9, 10 (2016), 804–815.

[43] Arnaud Soulet and Fabian M. Suchanek. 2019. Anytime Large-Scale Analytics of Linked Open Data. In *ISWC (1) (Lecture Notes in Computer Science)*, Vol. 11778. Springer, 576–592.

[44] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285 – 309.

[45] Thanh Tran, Günter Ladwig, and Sebastian Rudolph. 2013. Managing Structured and Semistructured RDF Data Using Structure Indexes. *IEEE Trans. Knowl. Data Eng.* 25, 9 (2013), 2076–2089.

[46] Ruud van Bakel, Teodor Aleksiev, Daniel Daza, Dimitrios Alivanistos, and Michael Cochez. 2021. Approximate Knowledge Graph Query Answering: From Ranking to Binary Classification. *CoRR* abs/2102.11389 (2021). arXiv:2102.11389 https://arxiv.org/abs/2102.11389

[47] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *J. Web Semant.* 37-38 (2016), 184–206. https://doi.org/10.1016/j.websem.2016.03.003

[48] W3C. [n.d.]. Resource Description Framework. http://www.w3.org/RDF/.

[49] Hongya Wang, Zeng Zhao, Kaixiang Yang, Hui Song, and Yingyuan Xiao. 2021. Approximate Nearest Neighbor Search Using Query-Directed Dense Graph. In *DASFAA (Workshops) (Lecture Notes in Computer Science)*, Vol. 12680. Springer, 429–444.

[50] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF Data Storage and Query Processing Schemes: A Survey. *ACM Comput. Surv.* 51, 4, Article 84 (sep 2018), 36 pages. https://doi.org/10.1145/3177850

[51] Haiwei Zhang, Yuanyuan Duan, Xiaojie Yuan, and Ying Zhang. 2014. ASSG: Adaptive structural summary for RDF graph data. In *ISWC (Posters & Demos) (CEUR Workshop Proceedings)*, Vol. 1272. CEUR-WS.org, 233–236.