On Distributed SPARQL Query Processing Using Triangles of RDF Triples

Hubert Naacke, Olivier Curé

Research online Publishing www.ronpub.com

Open Access

Open Journal of Semantic Web (OJSW)
Volume 7, Issue 1, 2020

http://www.ronpub.com/ojsw
ISSN 2199-336X

On Distributed SPARQL Query Processing Using Triangles of RDF Triples

Hubert Naacke[A], Olivier Curé[B]

[A] Sorbonne Universités, LIP6, CNRS, Paris, France, hubert.naacke@lip6.fr
[B] Université Paris Est, LIGM, Marne la Vallée, France, olivier.cure@u-pem.fr

Abstract

Knowledge Graphs are providing valuable functionalities, such as data integration and reasoning, to an increasing number of applications in all kinds of companies. These applications partly depend on the efficiency of a Knowledge Graph management system which is often based on the RDF data model and queried with SPARQL. In this context, query performance is preponderant and relies on an optimizer that usually makes an intensive usage of a large set of indexes. Generally, these indexes correspond to different re-orderings of the subject, predicate and object of a triple pattern. In this work, we present a novel approach that considers indexes formed by a frequently encountered basic graph pattern: triangle of triples. We propose dedicated data structures to store these triangles, provide distributed algorithms to discover and materialize them, including inferred triangles, and detail query optimization techniques, including a data partitioning approach for bias data. We provide an implementation that runs on top of Apache Spark and experiment on two real-world RDF data sets. This evaluation emphasizes the performance boost (up to 40x on query processing) that one can obtain by using our approach when facing triangles of triples.

Type of Paper and Keywords

Regular research paper: *indexing, RDF triangles, SPARQL, optimization, inference*

1 Introduction

Knowledge Graphs (KGs) are emerging as a new software tool that facilitates the development of innovative functionalities and services, *e.g.*, breaking data silos by supporting data integration and inferring implicit consequences from explicit knowledge. They have been adopted by many companies tackling a large set of domains, *e.g.*, Web search, retailing, product management, energy, healthcare. Applications served by these KGs require an efficient semantic data management approach. This is particularly evident when considering the size of the real world KGs, *i.e.*, in the range of millions to billions of graph nodes and edges. At such as scale, it is obvious that features such as horizontal scaling, data parallelism, fault-tolerance and efficient query processing must be present in the KG management system.

Considering the point of view of the end-user, an important performance metric associated to a database management system (DBMS) and query execution in particular is latency, *i.e.*, the time required to complete the execution of a given query. An adapted data indexing approach generally enables to reduce the latency to obtain a query answer set. Resource Description Framework (RDF) stores, *i.e.*, DBMSs that are using the RDF data model, are the most widely used KG management systems. Much work has been done to design and implement efficient indexing solutions [6], [15] and [19] for RDF stores. In fact, although most of the previously mentioned systems have been conceived at a time when data distribution was not needed (due

to rather small datasets), their indexing approaches have inspired distributed commercial, *e.g.*, Stardog[1], Amazon Neptune[2], GraphDB[3], and open source, *e.g.*, [20], systems. Most of these approaches concentrate on indexing a single triple at a time. For instance by indexing the subject (S), property (P), object (O), of a single triple in different orders, *e.g.*, SPO, SOP, PSO. Nevertheless, more complex triple structures are frequently occurring in RDF data sets and the Basic Graph Patterns (BGP) of SPARQL queries.

In this paper, we consider the indexing of triangles of triples, *i.e.*, a set of three triples such that each triple is connected with the two other triples. An analysis of the Yago2 [7] and Yago3 [11] real-world data sets highlights that approximately 16% of their triples are forming such a triangle structure. More precisely, in Yago2, 546 triangle types, *i.e.*, an ordered signature corresponding to the three properties of a triangle, cover over 1.2 million triangles. The cardinality of all these triangles are represented in Figure 1 for the 18 triangles containing two `rdf:type` properties (left side) and for the remaining 528 triangles containing no `rdf:type` property (right side). More generally, the Stanford Network Analysis Project (SNAP)[10] emphasizes that triangle structures are frequent in various types of graphs.

Our claim is that retrieving the variable bindings for a triple pattern representing a triangle should be performed more efficiently by using a single lookup over a triangle index structure than accessing each triple individually and performing join operations.

In order to address fault-tolerance and horizontal scalability issues, we present distributed algorithms based on the cluster-computing Apache Spark framework. We tackle reasoning aspects of triangles discovery and query processing by using the LiteMat encoding approach [3].

This paper is an extension of [14]. Its content has been enriched in order to be more self-contained (*e.g.*, more in-depth explanations on LiteMat's encoding scheme). Moreover, it provides results of new research conducted on strategies to efficiently discover triangles and process queries in a Spark distributed context and presents additional evaluations that demonstrate the adequacy of these new strategies. Intuitively, the strategies consider whether a parallel hash join is more efficient than a broadcast join; it devises a solution to combine these two join methods while taking into account data skew. This drastically impacts the performance of both the discovery and query processing in our system.

[1] https://www.stardog.com/
[2] https://aws.amazon.com/fr/neptune/
[3] https://www.ontotext.com/products/graphdb/

The paper is organized as follows. In Section 2, we provide some background knowledge on LiteMat and Apache Spark. Section 3 introduces the data structures used to cache triangles of RDF triples and details our discovery strategies for both cyclic and acyclic triangles. In Section 4, algorithms to optimize query processing are presented. This includes queries requiring some reasoning services to ensure exhaustive query answer sets. Some related work is highlighted in Section 5. An evaluation is proposed in Section 6. And finally, Section 7 concludes the paper and emphasizes on future work.

2 Background Knowledge

We cover two background knowledge in this section: LiteMat an encoding scheme for RDFS ontologies and Apache Spark on which our implementation is based on.

2.1 LiteMat

LiteMat is a semantic-aware encoding scheme that compresses RDF data sets and supports reasoning services associated to the RDF ontology language. In this work, we are focusing on the ρdf[12] subset of RDFS which considers inferences associated to the `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range` constructors. To address inferences drawn from these first two RDFS predicates, we attribute semantic-aware numerical identifiers to ontology terms, *i.e.*, concepts and predicates. More precisely, in LiteMat, concept encoding is based on an inferred graph, *i.e.*, the graph resulting from applying a concept classification over a concept hierarchy. Such an inference can be performed with an external reasoner such as HermiT [17]. This means that LiteMat is capturing the complete set of concept subsumptions, *i.e.*, up to the OWL DL ontology language, and not just the ones expressed by explicit `rdfs:subClassOf` triples.

Note that this paper also benefits from recent LiteMat extensions which are presented in [3]. These extensions consider OWL constructs such as `owl:sameAs`, `owl:inverseOf` and `owl:transitiveProperty`. In a nutshell, additional data structures and dedicated encoding approaches are supporting these new features. Up to this extension, LiteMat can be considered to tackle the RDFS++, *i.e.*, RDFS plus some OWL (popular) constructs, ontology language expressiveness. We are aiming to consider other extensions for LiteMat, *e.g.*, toward OWLRL or OWLQL.

This encoding scheme is performed by prefixing the identifier of a term with the identifier of its direct parent. This approach uses a binary representation of its entity

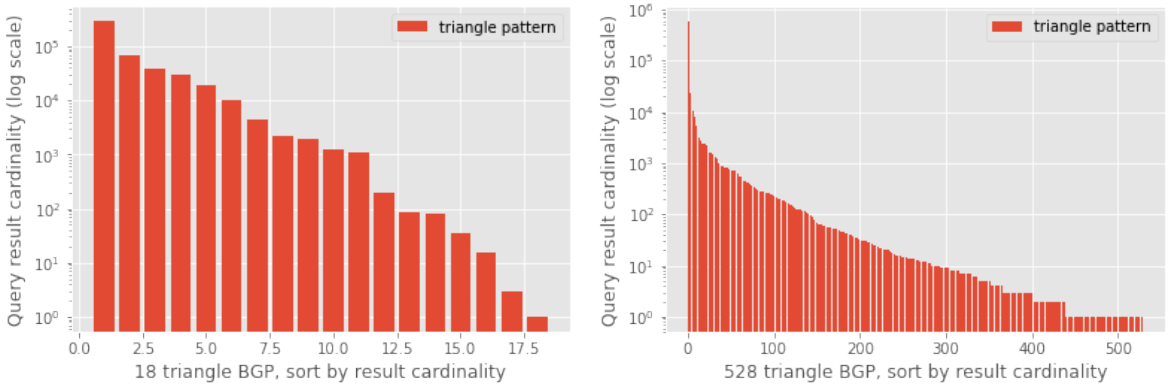

Figure 1: Cardinality of all BGP triangles in Yago2 (log scale)

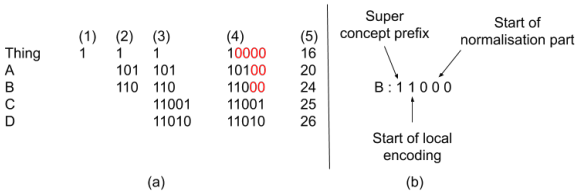

Figure 2: LiteMat encoding example

identifiers. In [3], a dedicated data structure is presented to address multiple inheritance cases. In such a situation, each concept can have several identifiers which are all stored in a hash table. A method then efficiently probes this structure to obtain all possible identifiers for a given concept.

The encoding is performed using a top-down approach, *e.g.*, starting from the most specific concept of the hierarchy (typically `owl:Thing` for the concept hierarchy and `TopObjectProperty` and `TopDataProperty` for property hierarchies), until all leaves are processed. A normalization is performed to guarantee that all encoding entries have the same length on a given hierarchy, *i.e.*, concept and property. This operation is performed by setting a set of right-most bits to 0.

In Figure 2, we consider a small ontology extract containing the following axioms: $A \sqsubseteq owl : Thing$, $B \sqsubseteq owl : Thing$, $C \sqsubseteq B$ and $D \sqsubseteq B$. Figure 2a highlight the top-down encoding approach with (1) setting the local identifier of $owl : Thing$, (2) its direct subconcepts (A and B) and B's subconcepts in (3). Then in (4) the normalization step is performed, *i.e.*, added right-most bits are written in red. Column (5) provides the integer value attributed to each concept.

The mapping between URIs and their identifiers are stored in dictionaries, two for the concepts and two for the properties to support a bidirectional retrieval, *i.e.*, from a URI to its identifier and from an identifier to its URI. Moreover, in the former dictionaries, additional identifier metadata are stored. For instance, the local length (binary length before the normalization phase) of each dictionary entry is stored along the final identifier entry. Figure 2b emphasizes the different metadata of the LiteMat encoding for the B concept: super concept identifier part, start of local encoding and start of the normalization part.

The semantic encoding of concepts and predicates supports reasoning services usually required at query processing time. For instance, consider a query asking for the pressure value of sensors of type S1. This would be expressed as the following two triple patterns: `?x pressureValue ?v. ?x type S1.` In the case sensor concept $S1$ has n sub-concepts, then a naive query reformulation requires to run the union of n+1 queries. With LiteMat's semantic-aware encoding, we are able, using two bit-shift operations and an addition, to compute the identifier interval, *i.e.*, [lowerBound, upperBound), of all direct and indirect sub-concepts of $S1$. We can thus compute the query with a simple reformulation: replacing the concept S1 with a new variable : `?x type ?newVar` and introducing a filter clause constraining values of this new variable: `FILTER (?newVar>=lowerBound && ?newVar<upperBound).`

2.2 Apache Spark

Apache Spark [22] is a cluster computing engine which can be considered as a main-memory extension of the MapReduce model enabling parallel computations on unreliable machines and automatic locality-aware scheduling, fault-tolerance and load balancing. While

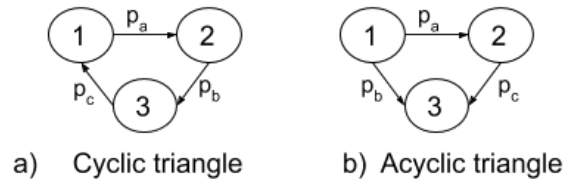

Figure 3: Cyclic and acyclic triple triangles

both, Spark and Hadoop[4], are based on a data flow computation model, Spark is more efficient than Hadoop for applications requiring frequent reuse of working data sets across multiple parallel operations.

This efficiency is mainly due to two complementary distributed main-memory data abstractions: (i) Resilient Distributed data sets (RDD) [21], a distributed, lineage supported fault tolerant data abstraction for in-memory computations and (ii) DataFrames (DF), a compressed and schema-enabled data abstraction. Both data abstractions ease the programming task by natively supporting a subset of relational operators like $project$ and $join$. DF is even more user-friendly by proposing both a Domain Specific Language (DSL) and a SQL interface. These operators enable the translation and processing of high-level query expressions (*e.g.*, SPARQL).

3 Discovering and Indexing Triangles of Triples

In this section, we present the two forms of triangles that can be observed in RDF graphs. Then, we propose a compact representation for these triangles. Finally, we introduce methods to discover both explicit and implicit triangles.

3.1 Cyclic and Acyclic Triangles

Triangles are composed of a set of three connected triples. Two forms are distinguished: cyclic (Figure 3a) and acyclic (Figure 3b). In a cyclic triangle, the out-going and in-going degrees of each node has a value of 1. In an acyclic triangle, one node has an out-going degree of 2 hence one of the two remaining nodes has an in-going degree of 2 and out-going degree of 0. The last node has out-going and in-going degrees of 1.

3.2 Triangle Representation

The objective of our indexing structure is to retrieve all the URIs and blank nodes identifying nodes of a given triangle in a deterministic manner as efficiently as possible and to store them using a compact approach. To satisfy these prerequisites, we consider that properties of a triangle are first class citizens. A direct impact is that BGPs forming a triangle with at least one property variable will not be handled efficiently with the triangle index. For instance, the following BGP: ($?x$ $?p$ y . $?x$ `rdf:type` c . y `rdf:type` c) is not considered in our triangle caching approach since our caching approach can not handle its processing efficiently. Thus, the variable bindings of this BGP, *i.e.*, for variables $?x$ and $?p$, will be retrieved using a one triple pattern at a time execution model. Due to the low frequency of variables at the property position in real-world SPARQL queries, we do not consider that this is an important limitation of our approach.

Our selected triangle caching structure takes the form of a distributed hash table (DHT) where the key is the set of the three properties involved in a triangle and the value corresponds to the set of node labels instantiating this triangle. We consider two triangle indexes, one for each triangle form, *i.e.*, cyclic and acyclic.

To address the deterministic aspect of our indexing structure, *i.e.*, every triangle instance is indexed exactly once, we apply a certain order on the set of properties forming our DHT key. Respecting this order is preponderant in the context of directed graphs. This order satisfies the following rules: (i) for an acyclic triangle, the first property of the DHT key relates to the node with two out-going edges and its object node has one out-going edge. The second property is the other property of the node with two out-going edges and the third property is the remaining one (*i.e.*, its subject node has one out-going edge and its object node has one in-going edge). (ii) for a cyclic triangle, the order of the properties is based on the integer-based identifiers provided by the LiteMat encoding. The first property of the key corresponds to the property with the minimal identifier among the properties of the triangle. Given this property, the order of the next two properties follow the direction imposed by the triangle.

The value component of our DHT structure contains a non-empty set of three node labels (URIs or blank nodes) that are instantiating the associated triangle key. Let us consider the cyclic triangle of Figure 3a and assume that the key is (p_a, p_b, p_c) then the first (respectively second and third) node label of a value instance is the subject of the triple with property p_a (respectively p_b and p_c). It hence yields the structure $(p_a, p_b, p_c) \rightarrow \{(1, 2, 3)\}$.

In cases where the three properties are equal, *i.e.*, $p_a = p_b = p_c$, then the node labels $(1, 2, 3)$ are ordered by increasing value. For Figure 3b's acyclic triangle and a (p_a, p_b, p_c) key, the first, second and third node labels are respectively the subject of the triple with property p_a, the

[4] https://hadoop.apache.org/

Algorithm 1: Compute/Index acyclic and cyclic triangles

Input: a set of RDF triples $D[s, p, o]$
Output: triangle index T_A and T_C

1 **for** $i \in [1, 3]$ **do**
2 $\quad D_i \leftarrow D.\texttt{rename}(s \text{ as } s_i, p \text{ as } p_i, o \text{ as } o_i)$
3 $J1_{Acyclic} \leftarrow D_1.\texttt{join}(D_2, s_1 = s_2)$
4 $Acyclic \leftarrow J1_{Acyclic}.\texttt{join}(D_3, o_1 = s_3 \text{ and } o_2 = o_3)$
5 $J1_{Cyclic} \leftarrow D_1.\texttt{join}(D_2, o_1 = s_2)$
6 $Cyclic \leftarrow J1_{Cyclic}.\texttt{join}(D_3, o_2 = s_3 \text{ and } o_3 = s_1).\texttt{map}(x => \texttt{order}(x)).\texttt{distinct}$
7 **for** $(p_a, p_b, p_c) \in Acyclic.\texttt{select}(p_1, p_2, p_3).\texttt{distinct}$ **do**
8 $\quad T_A[p] \leftarrow Acyclic.\texttt{where}(p_1 = p_a \text{ and } p_2 = p_b \text{ and } p_3 = p_c).\texttt{select}(s_1, o_2, s_3)$
9 **for** $(p_a, p_b, p_c) \in Cyclic.\texttt{select}(p_1, p_2, p_3).\texttt{distinct}$ **do**
10 $\quad T_C[p] \leftarrow Cyclic.\texttt{where}(p_1 = p_a \text{ and } p_2 = p_b \text{ and } p_3 = p_c).\texttt{select}(s_1, s_2, s_3)$
11 **return** T_A, T_C

object of the triple with property p_b and the remaining node label.

Given this deterministic index creation, the system can easily re-build triples involved in the triangle indexing structure. However, we will show in Section 4 that this reconstruction is never necessary.

3.3 Discovery of Explicit Triangles

The discovery of triangles explicitly stored in the KG are performed using Spark DF's DSL. Algorithm1 provides the pseudo code for the discovery and indexing of cyclic and acyclic triangles.

In lines 3-6, *Acyclic* and *Cyclic* triangles are discovered. Each triangle instance in a query result conforms to the canonical representation defined in Section 3.2. More precisely, the *order* method (line 6) implements the above-defined reordering rules. The *distinct* method removes the duplicates among resulting triangle instances that eventually have the same canonical representation. In lines 7-10, the indexes are built. Let T_A (respectively T_C) denote the index for acyclic (respectively cyclic) triangles. An index entry associates a pattern $p = (p_a, p_b, p_c)$ with a DF $T_A[p]$ that contains a set of triangle instances. Every $T_A[p]$ is distributed across the Spark cluster and cached in main memory to enable efficient index access: $T_A[p]$ is an in-memory distributed data structure which can be efficiently read and filtered in parallel during query processing.

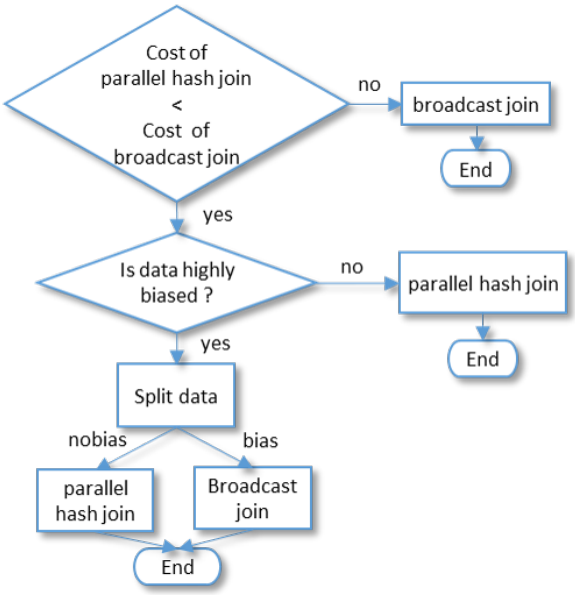

Figure 4: Physical join operation selection

In a distributed environment, efficiently processing multi-join queries such as *Acyclic* and *Cyclic* requires two optimization steps as illustrated on Figure 4 : (i) to decide, for each join operation, which physical join operator to use as suggested in [13]; and (ii) to handle highly biased data that may degrade parallel join computation.

3.3.1 Distributed Join in Presence of Biased Data

The query processing engine provides two join algorithms: either a parallel hash-join or a broadcast nested loop join. We compare the two algorithms performance applied to the case of triangle computation, *i.e.*, when joining the entire knowledge graph with itself.

Parallel hash-join runs faster than broadcast nested loop join for joining two large data sets since its communication cost is lower [13]. Parallel hash-join consists in distributing the data sets according to the join attribute values, then performing a local join for each value in parallel. However, in the presence of highly biased data, when few values are far more frequent, *e.g.*, the United_States URI in the Yago data sets, than the others, such join method will fail to fully execute in parallel as one can expect. Indeed, it starts by quickly computing in parallel the join for almost all the values, then it ends up spending most of its time in joining the few biased values. The fewer biased values exist in the data sets and the higher frequency they have, the longer

is the execution of the join. This is because the degree of parallelism is decreasing during the join computation so that only one processor, *i.e.*, a straggler, remains busy computing the highest frequent value.

To avoid this drawback, one has to use the broadcast nested loop join which is not sensitive to biased data. Indeed, triples with the same frequent value are distributed across the machines, which allows processing them in parallel. However, broadcast nested loop join has a high communication cost because it requires to transmit the data to be joined to every machine.

In summary, we face the problem where none of the two join algorithms brings satisfactory performance on its own to process the discovery of triangle queries. As far as we know, current query optimizers do not handle such a case of highly biased data, which gives us the opportunity to propose the following method.

Let D be a collection of (S, P, O) triples. The domain of the object o is composed of a list of values $Dom_0 = (o_1, ..., o_n)$ ordered by decreasing frequency, with the frequency f_o defined as $f_o(o_i) = |\sigma_{o=o_i}(D)|$, where σ is the standard selection operation of the relation algebra. We have:

$$\forall i < j, f_o(o_i) \geq f_o(o_j)$$

Similarly, the domain of the *subject* s is composed of a list of values $Dom_s = (s_1, ..., s_n)$ ordered by decreasing frequency $f_s(s_i) = |\sigma_{s=s_i}(D)|$. With these two functions, we can determine the biased values that cause the highest overhead on the join computation, *i.e.*, values with the highest frequency in the join result. In the general case of a subject-object join, we have $Dom_{so} = Dom_s \cap Dom_o$ with associated frequency $f_{so}(b) = f_s(b) \times f_o(b)$. Let $B = (b_1, ..., b_k)$ contain the K most frequent values of Dom_{so} which are considered as biased. Relying on these definitions, we propose to join the knowledge graph D with itself as follows:

- Split the data set D in two parts: D_{bias} which contains the triples with biased values for the join attributes and D_{nobias} which contains the remaining triples. Thus

  $$D_{bias} = D \ltimes B$$

  and

  $$D_{nobias} = D \setminus D_{bias}$$

- Process D_{nobias} using parallel hash join which ensures that the entire computation is performed with the expected degree of parallelism.

- Process D_{bias} using broadcast nested loop join: the communication cost remains low since the size of D_{bias} is relatively small.

- the final result is the union of the results on D_{bias} and D_{nobias}.

3.3.2 Optimizing $Acyclic$ and $Cyclic$ Queries

We apply the above method to process $Acyclic$ and $Cyclic$ queries. We assume a main memory query processing environment, the data set D is persisted in distributed main memory after being hash-partitioned on its triples' subject.

$Acyclic$ **query.** It is composed of two joins as detailed on Algorithm 1. On line 3 the first join $J1_{Acyclic}$ is a subject-subject join. The data bias is low on the $subject$ attribute, therefore we choose to compute it using a parallel hash-join on all the D triples. On line 4, the second join is on a compound key defined by (o_1, o_2) for $J1_{Acyclic}$ and respectively by (s_3, o_3) for D_3. We observe a low bias for such compound key. This is due to the correlation between two $object$ values is generally low.

Thus the second join is also computed using a parallel hash-join that implies to distribute $J1_{Acyclic}$ as well as D_3 on the compound keys (o_1, o_2) and (s_3, o_3) respectively. Although the size of $J1_{Acyclic}$ is L times the size of J_3, this join method outperforms the broadcast nested loops join method in our experiments. This is due to a small L value (*e.g.*, $L = 11$ for the Yago3 data set). Indeed, the later join algorithm needs to index the broadcasted data set on the fly which time may become predominant in comparison with the former join algorithm.

$Cyclic$ **query.** The data set is highly biased on the $object$ values. Therefore, we apply the proposed bias-aware join method. We split D into D_{bias} that contains the triples which $object$ values is among the k most frequent ones, and D_{nobias} that contains the remaining triples. The optimal number K of biased values that yield the best performance is determined through an empirical exploration of a series of K values, as shown in our experiments (Section 6). On line 5, $J1_{cyclic}$ is computed as:

$D1_{bias}.join(broadcast(D2_{bias}), o_1 = s_2)$
$.union(D1_{nobias}.join(D2_{nobias}, o_1 = s_2))$

Finally, on line 6, J_{cyclic} is computed via a broadcast nested loops join to prevent from an overhead of re-partitioning the result of $J1_{cyclic}$ which size is L' times larger than D_3 (*e.g.*, $L' = 155$ for the Yago3 data set). Thus, the communication cost only depends on D_3 which is temporary broadcasted to every cluster machine.

3.4 Discovery of Implicit Triangles

The full benefit of triangle-aware query processing of KGs is reached when both explicit and implicit triangles

are cached. In this section, we present a strategy that tackles inferences using the RDFS ontology language. The triangles considered here are necessarily acyclic and have the following signature $(p, rdf:type, rdf:type)$.

We aim to improve the processing of triangles contained in the BGP of a query by analyzing the underlying TBox, in particular the concept hierarchy as well as properties' domain and range.

We first generalize axioms of the TBox into triangles. Given a property p and the following Description Logic axioms: $\top \sqsubseteq \forall p C_r$, $\top \sqsubseteq \forall p^- C_d$, we can infer additional triangles when (i) $C_r \equiv C_d$, (ii) $C_r \sqsubseteq C_d$ or $C_d \sqsubseteq C_r$. Considering case (i), such a triangle may not have been discovered by our acyclic method if the ABox is incomplete, *i.e.*, some `rdf:type` triples are missing. Case (ii) is more interesting and considers the situation where the domain of p is a sub-concept of p's range or vice-versa.

These new triangles can be discovered using the following R_1^{TBox} SPARQL query:

```
select ?p1, ?c2 where {
{?p1 rdfs:domain ?c1.
?p1 rdfs:range ?c2.
?c1 rdfs:subClassOf ?c2 }
union {
?p1 rdfs:domain ?c2.
?p1 rdfs:range ?c1.
?c1 rdfs:subClassOf ?c2}
union {
?p1 rdfs:domain ?c2.
?p1 rdfs:range ?c2}}
```

Hence, for each result (p_1, c_2) of R_1^{TBox}, we know that for each triple ($x\ p_1\ y$) in the ABox and each c equals to either c_2 or a super-concept of c_2, there must exist a triangle ($x\ p_1\ y$. x `rdf:type` c . y `rdf:type` c). We store the (p_1, c_2) tuple in an additional property/concept hashMap, denoted pcCatalog, where the key is a tuple (p,1), the value 1 denotes the computation with R_1^{TBox}, and the value of the hashMap is the concept identifier, *i.e.*, $c2$. This hashMap supports some query processing optimization (see Section 4.4).

Example 1: Let us consider the LUBM ontology[5] and its $advisor$ property whose domain and range are respectively $Person$ and $Professor$. The LUBM ontology states that $Professor \sqsubseteq Faculty \sqsubseteq Employee \sqsubseteq Person$, hence the entry $(advisor, Person)$ has been added to pcCatalog after the execution of R_1^{TBox}. Note that whenever the domain or range of a property p is not specified in the ontology, we attribute `owl:Thing` to them. In that case, we consider that the generalization is not specific enough and we do not add entries to the pcCatalog for these properties.

The entries of pcCatalog computed with R_1^{TBox} can be used to infer novel triangle instances from the KG's ABox. These triangles are specialization of the triangles discovered using R_1^{TBox} and will be stored in the triangle index structure. Let $minClass(c, c')$ be a function that takes two ontology concepts belonging to the same hierarchy branch as parameters, *i.e.*, either c is a super-concept of c' or the other way around, and that returns the most specific super concept of c and c'. Let also assume that $(\rho, 1) \rightarrow \{C\}$ is an entry in pcCatalog tuple and that for each triple of the form (s,ρ,o) in the ABox where s (respectively o) is asserted the type c (respectively c'), $minClass(c, c') \sqsubseteq C$, then a new value $(s, minClass(c, c'), o)$ entry can be added to the triangle index for key $(\rho, type, type)$.

The following R_1^{ABox} SPARQL query retrieves value entries for our triangle structure:

```
select ?x, ?y, minClass(?t1,?t2) where {
?x p1 ?y.
?x rdf:type ?t1.
?y rdf:type ?t2.
minClass(?t1,?t2) rdfs:subClassOf C.
filter(minClass(?t1,?t2) != C)}
```

Example 2: We consider the context of Example 1 and an ABox containing the following triples: i_1 `advisor` i_2. i_2 `rdf:type FullProfessor`. i_1 `rdf:type Faculty` with i_1 and i_2 two individuals and $FullProfessor \sqsubseteq Professor$. Then R_1^{ABox} would retrieve a new value $(i_1, Faculty, i_2)$ entry for the $(advidor, rdf:type, rdf:type)$ triangle key since the most specific super concept of $Faculty$ and $FullProfessor$ is $Faculty$. Together with the LiteMat encoding query processing approach, this will improve the retrieval of variable bindings for queries containing this triangle pattern.

4 Query Processing with a Triangles Index

We present several optimization strategies for SPARQL query processing which are based on our triangle abstraction. These strategies either depend on our triangle caching approach or on inferences conducted over the SPARQL query's BGP. In the former, the system first parses the BGP of a SPARQL query to detect triangles. Intuitively, the approach is similar to the explicit method previously described (see Section 3.3) for RDF graphs and generates triangle signatures following the method described in Section 3.2. Given such a signature, a standard lookup is performed to retrieve all variable bindings of a triangle.

4.1 Inference-Free Triangle Query Processing

This method applies to arbitrary triangles found in SPARQL query's BGP that (i) do not contain any `rdf:type` properties and (ii) where all properties correspond to URIs, *i.e.*, are not variables. Triangles not satisfying (i) are handled by our inference-based approach which is described in Section 4.2. Triangles not satisfying (ii) would yield a partial 3-tuple property signature, *i.e.*, containing at least one variable. It would be inefficient, due to its non determinism, to search for such a signature in the set of keys of our triangle DHT.

For all triangles that follow constraints (i) and (ii), a simple lookup in the index retrieves all instances of variables bindings.

Example 3: Let us consider the following SPARQL query BGP: `?x isLocatedIn ?y. ?y isLocatedIn ?z. ?x isLocatedIn ?z`. The triangle key is `(isLocatedIn, isLocatedIn, isLocatedIn)` which retrieves over 574 800 variables binding for `?x, ?y` and `?z` for the Yago2 data set.

If a triangle contains some URIs at the subject or object positions, a literal at the object position, *e.g.*, `?x=<Berkeley_Marina>` in the previous query, then a simple filter over the proper variable binding, *e.g.*, the first one in this running example, would yield the correct bindings for the other query variables.

4.2 Inference-Based Triangle Query Processing

We now propose another triangle generalization approach, as R_1^{TBox}, but when the concepts of the domain and range of a property are not in the same hierarchy branch, *i.e.*, not related by a subsumption relationship. Like R_1^{TBox}, this reasoning case solely depends on the TBox and will generate new entries in the pcCatalog hashMap.

In order to perform this inference, we need to define a function denoted $lca(c, c')$ (lowest common ancestor - LCA) which takes as parameters two ontology concepts and returns the most specific super concept common to c and c'.

The reasoning service is performed by the following SPARQL query, denoted R_2^{TBox}:

`select ?`p_1`, lca(?`c_1`,?`c_2`) as` lca `where {`
`?`p_1 `rdfs:domain ?`c_1`.`
`?`p_1 `rdfs:range ?`c_2
`filter (lca(?`c_1`,?`c_2`)<>?`c_1 `and`
`lca(?`c_1`,?`c_2`)<>?`c_2`) }`

Thus, for each R_2^{TBox} answer result (p_1, lca), the system infers that each triple (x p_1 y) in the ABox and for all c equals to either lca or a super-concept of lca then there exists a triangle (x p_1 y . x rdf:type c . y rdf:type c). Note that the methods proposed in [3] can be easily applied to support multiple inheritance of concepts. The result of this query is persisted in pcCatalog but with a key made of a tuple (p,2), the value 2 denoting this entry has been computed from R_2^{TBox}. In Section 4.4, we present how this data structure is used to optimize some SPARQL queries.

4.3 Efficient Computation of R1 and R2 Using LiteMat Encoding

Using LiteMat's encoding approach provides two main advantages: (i) a compact representation of the (RDFS) ontology semantics and RDF data sets, (ii) an efficient query rewriting service that supports RDFS inferences. These benefits come at low storage and computational costs since they only require a set of dictionaries for the concept and property hierarchies, the domain and range of ontology properties; two bit shift operations and the addition of an integer to obtain sub-concepts (respectively sub-property) of a given concept (respectively property). These two advantages guarantee that we can perform RDFS inferences without accessing any Semantic Web compliant API, *e.g.*, Apache Jena[5]. Hence, it provides an important performance gain for the execution of queries containing triangles.

All queries, *i.e.*, R_1^{TBox}, R_2^{TBox} and R_1^{ABox}, presented up to now heavily benefit from LiteMat's encoding facility. The TBox queries use the LiteMat dictionaries providing to lookup for the concept identifiers of the domain (respectively range) of a property p and are denoted $dom(p)$ (respectively $ran(p)$). R_1^{TBox} uses the $upper(c)$ function which provides the upper limit of the interval surrounding all the sub-concepts of a given concept c (performed with two bit shifts and an addition). The following Algorithm2 computes R_1^{TBox} efficiently.

Considering the R_1^{ABox} query, since it is performed over the ABox, it can be processed directly over the encoding ABox via a translation to Spark's DF. It only remains to compute the $minClass$ function. Again, LiteMat TBox's encoding makes it quite easy and efficient, *i.e.*, without accessing the original TBox, an expensive API and any external reasoner. In fact, due to LiteMat's encoding properties, $minclass(C, D) = C \wedge D$ where C and D are two LiteMat identifiers and $\wedge$ corresponds to the logical conjunction operation.

Finally, the R_2^{TBox} query is computed with Algorithm 3 which uses the dictionaries of LiteMat. This algorithm integrates the computation of the lca function (from line 3 to 7) over two concepts present in two different branches of the concept hierarchy. Recall that

[5] https://jena.apache.org/

Algorithm 2: Compute query R_1^{TBox} with LiteMat

Input: a property p, LiteMat's range and domain dictionaries
Output: a LiteMat concept identifier or null

```
1  r ← range(p);
2  d ← domain(p)
3  if r==d then return d;
4  else if r>=d and d<upper(r) then return r;
5
6  else if d>=r and r<upper(d) then return d;
7
8  return null
```

Algorithm 3: Compute query R_2^{TBox} with LiteMat

Input: a property p, LiteMat's dictionaries
Output: a LiteMat concept identifier or null

```
1  r ← range(p);
2  d ← domain(p)
3  led ← locaEncoding(d);
4  ler ← locaEncoding(r)
5  shift ←
     encodingLength − minimum(led, ler)
6  lca = ((r ≫ shift)and(d ≫ shift)) ≪ shift
7  if lca!=d and lca !=r then return lca;
8
9  return null
```

some metadata are stored in the concept and property dictionaries. Here, we are using the index of the start of the local encoding of a concept. Intuitively, line 3 retrieves the start of the local encoding of the domain and range of the property. The minimum of these two values represents the encoding level at which these two concepts have a common ancestor. Line 7 is responsible for computing the LCA of these two concepts. In fact, it performs a binary AND operation over the results of the right bit shift over the two concepts and finally performs a left bit shift over the result value.

4.4 Property/Concept Pair Catalog Optimization

In this section, we consider that the following BGP of a SPARQL Q has been submitted to our system: `?x` ρ `?y. ?x rdf:type D. ?y rdf:type D`. We will also consider that we may have a pcCatalog entry containing with key ρ and a triangle index entry key (ρ, `rdf:type`,`rdf:type`).

Lines 1 to 7 of the procedure described in Algorithm

Algorithm 4: Processing a BGP with a triangle

Input: pcCatolog pc

```
1  if lookup (p,1)==ρ then
2      if C ≡ D or C ⊑ D then
3          // triangle index-free execution of query
             ?x ρ ?y
4
5      else if D ⊑ C then
6          // use triangle index and LiteMat rewrite
             the query
7          else
8              //answer set is empty
9  else
10     if lookup (p,2)==ρ then
11         if C ≡ D or C ⊑ D then
12             // triangle index-free execution query
                 ?x ρ ?y
13
14         else
15             // answer set is empty
16     else
17         // use the triangle index
```

4 handle the optimization provided by the R_1^{TBox} query. They enable to remove the two `rdf:type` triples of the BGP in Lines 2 and 3, and permit a LiteMat rewriting in Line 5. Otherwise, it is not possible to retrieve any variable bindings for $?x$ and $?y$ and the result is thus empty (Line 7). Lines 8 to 13 manage pcCatalog entries computed from R_2^{TBox}. In that case, the optimization also permits to remove the `rdf:type` triples and even to directly return an empty answer set for that BGP part. Line 15 considers the standard triangle index lookup (for acyclic triangles containing two `rdf:type` properties) when no entries are present for p in pcCatalog.

5 Related Work

A large body of research has been conducted toward improving the query evaluation of SPARQL queries. A common aspect lies in the multiple indexing approach. For instance, YARS2[6] is a distributed system that supports six indexes over quads where the fourth element is denoted as the triple context (*i.e.*, C). The six indexes thus correspond to SPOC, POC, OCS, CSP, CP and OS. Hexastore[19] has six indexes over RDF triples, namely SPO, SOP, PSO, POS, OSP and OPS. RDF-3X[15] adds six indexes for pairs of elements (*i.e.*, SP, PS, SO, OS, OP, PO) and three indexes for single elements (*i.e.*, S,

P and O). Nevertheless, these influential systems do not consider complex triple pattern forms as indexing entries, *e.g.*, triangles of RDF triples.

CliqueSquare [4] aims at maximizing local joins, but replicates the whole data set 3 times which is not applicable to a main-memory approach. The S2RDF[16] SPARQL processor is the system that certainly shares the most features with our framework. In fact, both systems are based on the Spark cluster computing framework and they propose a data layout that mimics indexing. Its modeling approach extends the vertical partitioning approach proposed in [1]. In fact, the pre-processing of S2RDF's extended Vertical Partitioning (ExtVP) is much more involved, in terms of required memory footprint and computational cost, than our triangle indexing. Moreover, S2RDF does not consider reasoning during pre-processing and query evaluation. Hence, we consider that our triangle structure approach is orthogonal to S2RDF's ExtVP, *i.e.*, it could be integrated into its indexing panoply of solutions.

The AdPart system [2] implements a main memory SPARQL engine using MPI (Message Passing Interface) data transfer and lacks fault tolerance (contrary to Spark). AdPart uses a distributed semi-join operator to limit data transfer for selective joins over large sub-queries by combining adapted partitioned and broadcast join variants. It could be interesting to study this new operator within our framework.

Many papers consider efficient data placement to speed up the query processing. [8] proposes a survey of such partitioning approaches for the RDF data model and SPARQL querying. Recently, the Molecule Hash Cover method [9] has been proposed and presents an interesting trade-off between scalability and query execution performance. Our triangle partitioning approach in the presence of bias can be considered as a special case of data placement strategy where the molecule represents a triangle and where the partitioning is performed up a to certain threshold.

A recent work on indexing DFs [18] did not fit our needs when we were designing our triangle index. The main reason is that in [18] the set of rows associated with a given key is not distributed across several partitions. While this is an efficient design for highly selective index entries, this is not appropriate in our case where some entries are not selective enough, *i.e.*, a triangle pattern that can match millions of triangle instances. Using [18], the instances associated with a given index entry are grouped together in one partition, whereas it is more efficient to distribute them across several partitions: allowing to access them in parallel, as our triangle index enables.

6 Evaluation

We now detail the triangle index implementation on top of the Apache Spark parallel computing platform, evaluate the overhead of building such index, and the benefit of using it for SPARQL query processing. Note that building the pcCatalog has no overhead since the TBox size is generally relatively small compared to the ABox size.

6.1 Experiment Setup and Index Implementation

We conducted an evaluation on two real data sets Yago2 [7] and Yago3 [11] that respectively contain 4.5M and 12.4M fact triples, and 5.4 (respectively 24M) `rdf:type` triples. We only consider fact triples having object properties. The other triples having datatype properties can not be involved in triangles since they can not have out-going edges.

We implemented the triangle index such that it allows for parallel query processing conforming to the Apache Spark execution model. Index creation first consists of processing the $Acyclic$ and $Cyclic$ triangle queries, *cf.* Algorithm 1 which depends on the data bias. We report with Figure 5 (respectively Figure 6) on the distribution of the join keys in the result of the first join $J1_{Acyclic}$ (respectively $J1_{Cyclic}$).

We can see that for $J1_{Acyclic}$ the most frequent URI is <United_States>. That URI is the subject of 6.3M bindings out of 137M bindings in the query result *i.e.*, it represents 4.5% of the entire result set. The bias is even stronger for $J1_{Cyclic}$ where the most frequent key is associated with 1.4B bindings which represents 37% of the entire result set.

Such bias will slow down the parallel join evaluation because the processor in charge of the most frequent keys has to produce a large part of the whole result, considering that each join key is handled by one processor. In other words, the workload of computing the join result is not balanced if the result size per processor is lower than the result size associated with of the most frequent key. For example, suppose we want to process $J1_{Cyclic}$ using 50 processors; one processor would produce 37% of the result instead of 2% in a balanced situation free of bias. We say that the bias ratio is $37/2 = 18.5$ in this case.

In summary, to quantify the impact of the bias on $J1$ queries we define the bias ratio as follows. Let C be the number of cores allocated to process a join (*i.e.*, the degree of parallelism is C). In a perfectly balanced case, all the cores would produce the same amount of result bindings. Thus the number of bindings per core is $|J_1|/C$. In case of bias, the core dedicated to the

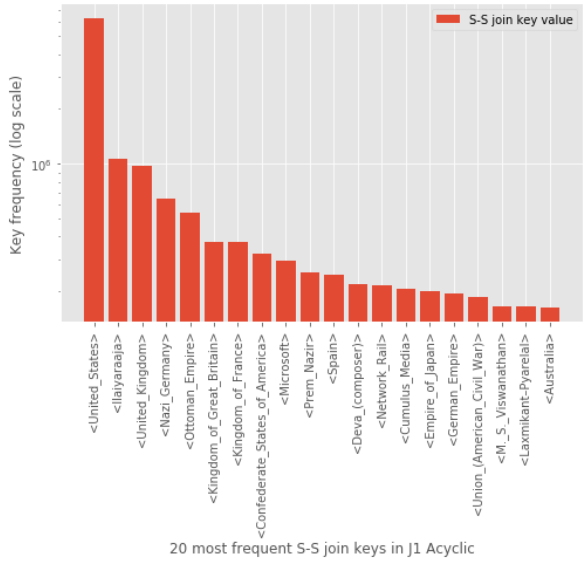

Figure 5: Data bias in $J1_{Acyclic}$

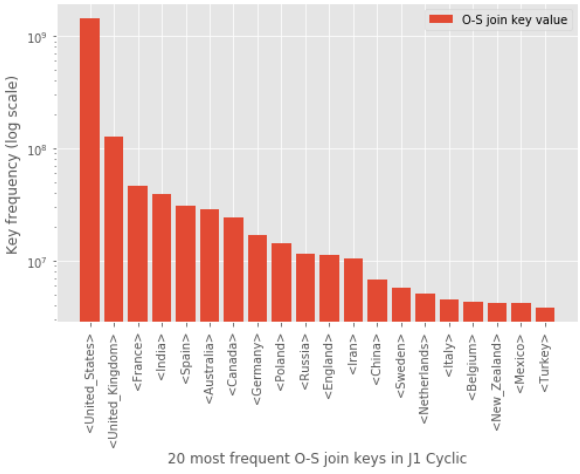

Figure 6: Data bias in $J1_{Cyclic}$

most frequent key is producing f bindings (as defined in Section 3.3.1). So, the bias ratio is

$$biasRatio = \frac{max(f, \frac{|J_1|}{C})}{\frac{|J_1|}{C}}$$

We report the bias ratio on Figure 7 for several subsets D^K_{bias} of the Yago3 data set. D^K_{bias} is the original Yago3 data set except the triples which join key is in the k most frequent values of $J1_{Cyclic}$ (in red, dotted) and $J1_{Acyclic}$ (in blue, crossed). The number of cores is set to $C = 96$ as in further experiments.

Figure 7 clearly shows that for $J1_{Cyclic}$ the most frequent value may cause significant performance degradation because a straggling task has to process 35 times more results than what is expected in a balanced case. Not only the top-1 value has a great impact. Also the other successive top-K most frequent values are concerned: removing the top-12 most frequent values still causes a straggling task with 5 times more results than in a balanced case. The blue plot shows that for $J1_{Acyclic}$ which is less biased, only the top-1 most frequent value may slow the join processing down. Finally, Figure 7 highlights a performance trade-off: choosing a high K value allows for removing stragglers, but on the other hand, it leaves aside a larger set of triples having their join key among the top-K values and thus it requires a longer additional time to process them. We investigate this trade-off in the next Section.

6.2 Impact of Bias on T_A and T_C Performance

The goal of this section is to empirically tune the optimal k parameter for T_A and T_C queries over the large Yago3 dataset. We conducted this experimentation from 1 to 8 machines, each node having 12 cores and 20GB memory.

For T_A, we vary k from 0 to 4. For each k, we process the query either in a centralized setting *i.e.*, 12 cores on a single node, or on distributed settings, *i.e.*, 48 and 96 cores on 4 and 8 nodes respectively. We report the T_A performance on Figure 8. For $k = 0$ the bias is not taken into account whereas for $k > 0$ the k most frequent values are processed separately.

The performance benefit from $k = 0$ to $k = 1$ is 25% for 8 nodes, 16% for 4 nodes, and 8% for a single node. The performance results are consistent with the preliminary analysis of subject frequencies shown on Figure 7: for $k > 0$ the bias ratio is almost constant and so are the query response times.

On Figure 9, we detail response times of the T_A query processed on both the biased (blue) and the unbiased (red) parts. We can see that the predominant cost is due to the large unbiased part of the dataset. Indeed, processing the small biased part (*i.e.*, 6.3M triples for $k = 0$ to 13M for $k = 4$) out of 143M triples is less than 10% of the complete $J1_{Acyclic}$ result set.

The impact of bias is even higher for T_C as shown on Figure 10. For $k = 0$ which means ignoring bias, the response times were 52 minutes with 8 nodes (96 cores) and above 1 hour with less nodes. Therefore they have not been reported because they were considered as intractable. Indeed, referring to Figure 6, the most frequent value, *i.e.*, <United_States>, generates 1.4B bindings which is about one third of the entire $J1_{Cyclic}$ result set being processed by only one core. This clearly

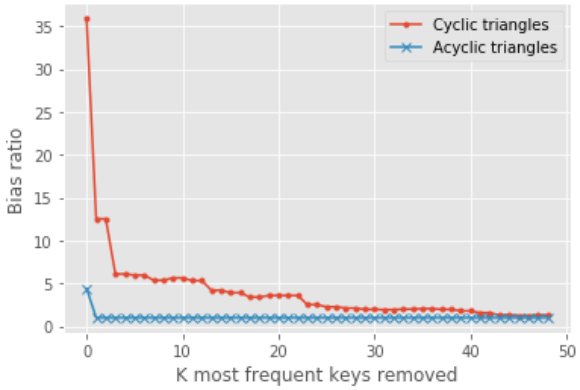

Figure 7: Bias ratio in $J1_{Cyclic}$ for increasing number of most frequent keys removed

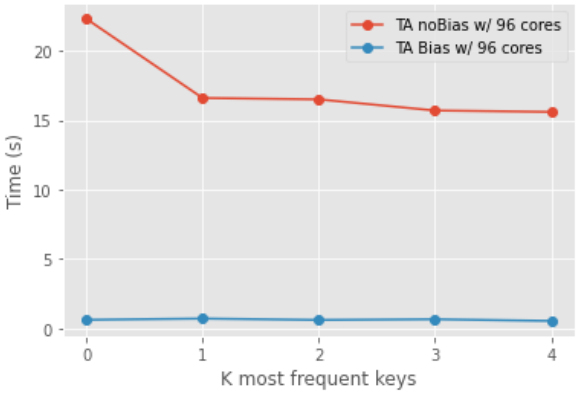

Figure 9: Detailed TA response times for the biased/unbiased parts wrt. k

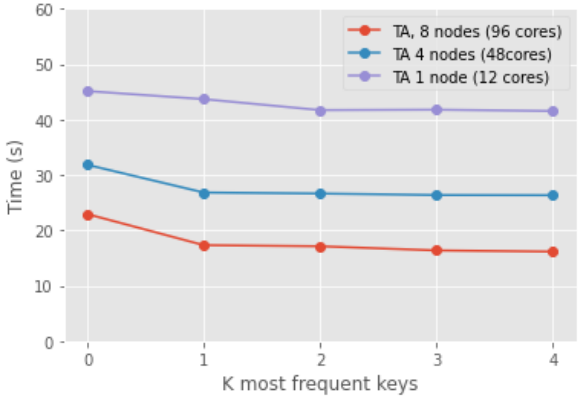

Figure 8: TA response time when splitting the dataset at k

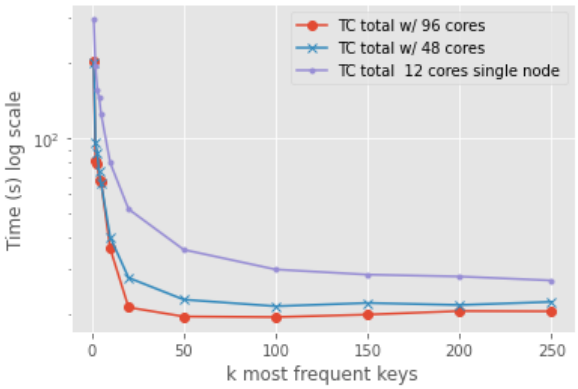

Figure 10: TC response time (log scale) when spliting the dataset at k

illustrates an unbalanced computation problem.

From $k = 1$ to $k = 50$, the response time is decreasing rather fast and yields a convincing performance improvement of 10x for 8 nodes and 8x for 4 nodes. For higher k values, the response time is decreasing more smoothly and tends to stabilize for $k > 200$. Figure 10 also shows the benefit of using a cluster engine to compute triangles in comparison with a centralized setting: for $k = 100$ the performance gain is 28% and 35% when using 4 and 8 nodes respectively. Still, this benefit is far from a linear speed-up, one reason is that the network communication dominates the join computation, with little demand for CPU resources.

We bring more details about response times on Figures 11 and 12 for the two distributed settings with 4 nodes (48 cores) and 8 nodes (96 cores) respectively.

On each figure, we report the times to process the bias and unbiased parts of the dataset separately. This highlights the bias/nobias trade-off: on one hand processing a truncated and more uniform dataset, and on the other hand facing the overhead of completing the computation with a larger biased part. Searching for an optimal k value, one can conclude that a range of k values in $[150 - 250]$ yields near optimal performance. Note that larger k values may cause a failure to process the biased part because this corresponds to larger broadcasted parts of the dataset that may exceed the available memory at each node.

Finally, these figures demonstrate that biased data occurring in knowledge graphs need careful attention. These experiments also motivate the need for more transparent query processing methods to handle biased data in SPARQL query engines. Based on these extensive empirical experiments, we set $k = 1$ for T_A and $k = 100$ for T_C in Section 6.3.

6.3 Index Creation Time and Size

The experimentation goal is to assess the index creation overhead in terms of time and size, and the scalability of

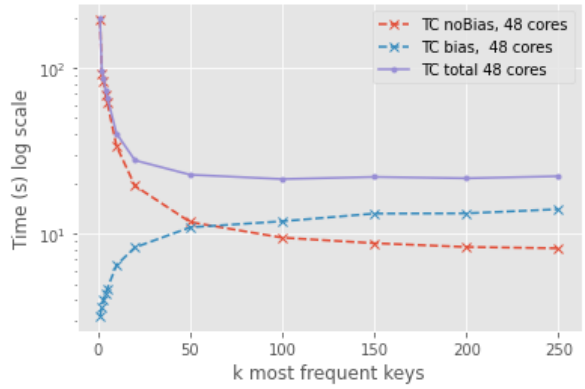

Figure 11: Detailed TC response time for 4 nodes

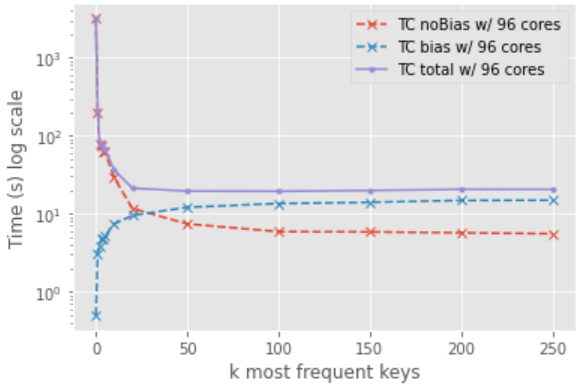

Figure 12: Detailed TC response time for 8 nodes

the approach when using the same cluster engine from 1 to 8 nodes as in the previous section.

T_A and T_C are hashMaps in the application driver, with values $T_A[p]$ and $T_C[p]$ pointing to DFs persisted in the cluster memory.

In the left part of Figure 13, we report on the index creation time including the computation of $Cyclic$ and $Acyclic$ results mentioned in Algorithm 1 for Yago2 and Yago3.

Using a single node, the overall index creation time is 79 seconds for Yago2 and 130 seconds for Yago3.

The detailed processing times for T_A, T_C and *inferred TA* resulting from R_1^{ABox} (Section 3.3) are also reported. In the distributed case, the creation time drops down to 50 seconds for 8 nodes (96 cores). From 4 to 8 nodes, the speedup is 1.6x for T_A and 1.9x for *inferred TA*. The T_C speedup is only 1.2x because for the distributed cases (4 and 8 nodes) most of the time is spent in shuffling the result of the first join to the nodes, which implies a predominant network cost.

The size of indexes created for Yago2 and Yago3 are presented in right part of Figure 13. The index T_A and T_C contain 1.2M triangle instances and the total number of different patterns, *i.e.*, index entries, is 546. Their overall memory footprint is 47MB . This represents only 16% of the KG restricted to triples with object properties whose size is 296MB, and an even smaller ratio compared to the entire KG including triples with datatype property. Similar proportions have been observed for Yago3.

6.4 Query Processing Performance

This evaluation also assesses the relative performance benefit of accessing the index compared to accessing the KG for triangle queries. We ran all the possible triangle BGP queries where every node is a variable and with non empty result sets, *e.g.*, the two following BGP were executed:

Q_1: {?x isLocatedIn ?y. ?y isLocatedIn ?z. ?x isLocatedIn ?z}

Q_2: {?x created ?y. ?y isLocatedIn ?z. ?x isLocatedIn ?z}

Using our approach, each query Q_i is transformed into an index probe $I_i = T_A[p_i]$ with p_i being bound with the three properties mentioned in Q_i. For instance, $p_2 = (created, isLocatedIn, isLocatedIn)$ for Q_2. Then all the partitions of the DF I_i are scanned in parallel to get the result of Q_i. In comparison, we also run the same queries without the index but still using an optimized query plan that performs the joins efficiently. Experiments are run on 1 machine with 16 cores and 20GB memory allocated to Spark engine.

Figure 14 reports the query response times of triangle queries for 546 distinct acyclic patterns on Yago2. The query response time includes the time to access every binding of a result set. In the context of the Spark lazy execution model, we use a Spark *action* that not only counts the number of bindings but really reads the content of every binding. The index performance gain is ranging from 3x to 40x.

We now emphasize the relative benefit of the index for selective queries that may already perform efficiently without an index. To this end, we experiment selective triangle BGP such that one variable out of three is bound to a value. More precisely, we consider the worst case for our index and target on a query Q' that must benefit the less from an index. Q' is defined as the most selective query issued from the less selective triangle BGP. We analyzed the BGP cardinalities and choose the one that matches the highest number of instances: it is the BGP with $p1 = p2 = p3 = locatedIn$. Then, we bind the subject of p_1 and p_2 with a value such that it maximizes the query selectivity: among the instances, we choose a node such that only one triangle contains this node value. That condition holds for the node *2015_NBA*. We get the

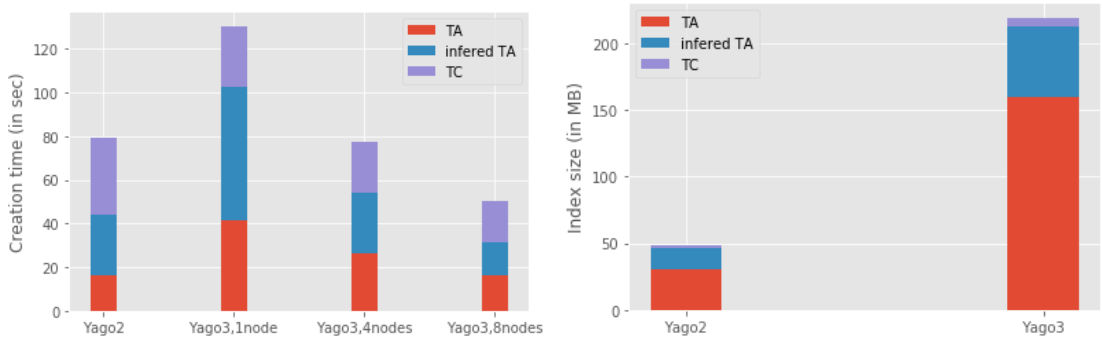

Figure 13: Creation time (left) and size (right) of T_A and T_C indexes

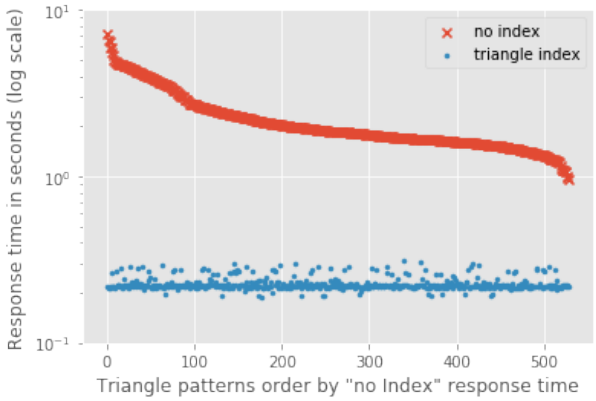

Figure 14: Query response time for Yago2 triangle patterns

following Q' BGP:

```
<2015_NBA> isLocatedIn ?y.
?y isLocatedIn ?z.
<2015_NBA> isLocatedIn ?z.
```

The response time of Q' is 351ms using the T_A index and 1853ms when accessing the KG directly without any index. We conclude that our index brings a speedup of at least 5x making it useful in practice.

7 Conclusion

We introduced a set of solutions for the efficient management of RDF data sets where triangle patterns are frequently occurring. Our data structures and algorithms tackle important management issues such as indexing, query processing and reasoning. This approach is orthogonal to existing solutions which solely consider indexing single triples instead of groups of them. Our evaluation was conducted in a scalable, parallel computing setting over two large real-world data sets and emphasized improvement of query processing of up to two orders of magnitude with a limited memory footprint overhead and relatively fast index creation. As future work, we are considering other data management issues where triangles of RDF triples can provide some benefits, *e.g.*, data cleansing and curation, KG enrichment and recommendation services.

References

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007, pp. 411–422.

[2] R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning," *VLDB J.*, vol. 25, no. 3, pp. 355–380, 2016.

[3] O. Curé, W. Xu, H. Naacke, and P. Calvez, "Litemat, an encoding scheme with RDFS++ and multiple inheritance support," in *The Semantic Web: ESWC 2019 Satellite Events - ESWC 2019 Satellite Events, Portorož, Slovenia, June 2-6, 2019, Revised Selected Papers*, 2019, pp. 269–284.

[4] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis, "Cliquesquare: Flat plans for massively parallel RDF queries," in *IEEE ICDE*, 2015, pp. 771–782.

[5] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *J. Web Sem.*, vol. 3, no. 2-3, pp. 158–182, 2005.

[6] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A federated repository for querying graph structured data from the web," in *The Semantic Web, 6th International Semantic Web Conference ISWC 2007*, 2007, pp. 211–224. [Online]. Available: http://iswc2007.semanticweb.org/papers/211.pdf

[7] J. Hoffart, F. Suchanek, and G. Weikum, "Yago2: A spatially and temporally enhanced knowledge base from wikipedia," *Artif. Intell.*, vol. 194, pp. 28–61, 2013.

[8] D. Janke and S. Staab, *Storing and Querying Semantic Data in the Cloud*. Cham: Springer International Publishing, 2018, pp. 173–222.

[9] D. Janke, S. Staab, and M. Leinberger, "Data placement strategies that speed-up distributed graph query processing," in *Proceedings of The International Workshop on Semantic Big Data*, ser. SBD '20. New York, NY, USA: Association for Computing Machinery, 2020.

[10] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[11] F. Mahdisoltani, J. Biega, and F. M. Suchanek, "YAGO3: A knowledge base from multilingual wikipedias," in *CIDR*, 2015. [Online]. Available: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper1.pdf

[12] S. Muñoz, J. Pérez, and C. Gutierrez, "Simple and efficient minimal rdfs," *Web Semant.*, vol. 7, no. 3, pp. 220–234, Sep. 2009.

[13] H. Naacke, B. Amann, and O. Curé, "SPARQL graph pattern processing with apache spark," in *Proceedings of Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017*, 2017, pp. 1:1–1:7.

[14] H. Naacke and O. Curé, "Triag, a framework based on triangles of RDF triples," in *Proceedings of The International Workshop on Semantic Big Data, SBD@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, 2020, pp. 3:1–3:6.

[15] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[16] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2RDF: RDF querying with SPARQL on spark," *PVLDB*, vol. 9, no. 10, pp. 804–815, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf

[17] R. Shearer, B. Motik, and I. Horrocks, "HermiT: A Highly-Efficient OWL Reasoner," in *Proc. of the 5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008 EU)*, A. Ruttenberg, U. Sattler, and C. Dolbear, Eds., Karlsruhe, Germany, October 26–27 2008.

[18] A. Uta, B. Ghit, A. Dave, and P. A. Boncz, "[demo] low-latency spark queries on updatable data," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 2009–2012.

[19] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[20] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "TripleBit: a fast and compact system for large scale RDF data," *PVLDB*, vol. 6, no. 7, pp. 517–528, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p517-yuan.pdf

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, 2010. [Online]. Available: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

AUTHOR BIOGRAPHIES

Hubert Naacke is an assistant professor at Sorbonne Université, Paris 6, France. He is a member of the LIP6 CNRS research Lab. He has published 4 journal papers, over 40 papers in international, peer-reviewed conferences on databases, big data and the semantic web.

Olivier Curé is an assistant professor at the Université Paris-Est, France. He is a member of the LIGM CNRS research Lab. He has published 1 book, 5 book chapters, 9 journal papers and over 70 papers in international, peer-reviewed conferences on databases, big data and the semantic web.