Parallel Quotient Summarization of RDF Graphs

Pawel Guzewicz, Ioana Manolescu

Parallel Quotient Summarization of RDF Graphs

Paweł Guzewicz
Ioana Manolescu
{pawel.guzewicz,ioana.manolescu}@inria.fr
Inria and LIX (UMR 7161, CNRS and Ecole Polytechnique)
France

ABSTRACT

Discovering the structure and content of an RDF graph is hard for human users, due to its heterogeneity, complexity, and possibly large size. One class of tools for this task are *structural RDF graph summaries*, which allow users to grasp the different connections between RDF graph nodes. RDFQuotient graph summaries are a brand of structural summaries we developed. They are usually very compact, making them good for first-sight visual discovery. Existing algorithms for building these summaries are centralized, and require the graph to fit in memory.

Going beyond, in this work we present novel algorithms for building RDFQuotient summaries in a parallel, shared-nothing architecture. We instantiate our algorithms to Apache Spark platform; our experiments demonstrate the merit of our approach.

CCS CONCEPTS

• **Information systems** → **Database performance evaluation**; **Resource Description Framework (RDF)**.

KEYWORDS

RDF graphs, summarization, parallel computations, Spark

ACM Reference Format:
Paweł Guzewicz and Ioana Manolescu. 2019. Parallel Quotient Summarization of RDF Graphs. In *The International Workshop on Semantic Big Data (SBD'19), July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3323878.3325809

1 INTRODUCTION

The structural heterogeneity inherent to RDF graphs makes it hard for the casual users to get acquainted with a graph's structure. To help solve this problem, many *RDF summarization* techniques have been proposed [CGK+18]. Each of them builds out of a given RDF graph, a compact structure which conveys the essential information of the graph, all that while being much more compact. Among these techniques, we have studied in recent years *quotient structural summarization of RDF graphs* [ČGM15b, ČGM17, GGM19]. A quotient RDF graph summary is a graph obtained by grouping RDF nodes in *equivalence classes*, each class being represented by a summary node; quotient summary edges similarly "represent" potentially many RDF edges. While quotient summarization has been studied for many equivalence relations, in particular for bisimilarity [HHK95] equivalence relations [MS99, CLO03, KBNK02, CMRV10, SNLPZ13, TLR13], the interest of the equivalence relations we defined in [GGM19, ČGGM18] is to be very *tolerant of heterogeneity*, thus leading to very *compact* RDF graph summaries, which are particularly suited for human users seeking to get an idea of an RDF graph structure. This is illustrated for many popular RDF graphs in our online gallery[1].

On the algorithmic side, we have previously proposed scalable algorithms for *centralized* summary construction, including incremental ones, capable of reflecting a newly added triple in a summary without the need to re-summarize the graph [GGM19]. A limitation of these algorithms is that their memory needs grow linearly with the size of the graph, which is problematic for very large graphs in memory-constrained environments.

This work presents novel, *parallel* algorithms for building our summaries out of an input RDF graph. We have implemented our algorithms using Spark; our experiments confirm that parallelism allows to distribute work and speed up execution.

Below, Section 2 recalls RDFQuotient summaries (most examples are borrowed from [GGM19]). Section 3 presents our contribution, namely our parallel summarization algorithms. Section 4 describes our experiments. We then discuss related work and conclude.

2 RDFQUOTIENT SUMMARIES

Let U be a set of URIs, L be a set of literals and B be a set of blank nodes as per the RDF specification. An RDF graph G is a set of *triples* of the form (s, p, o) where $s \in U \cup B \cup L$, $p \in U$ and $o \in U \cup B \cup L$. The special URI *type*, part of the RDF standard, is used to attach types to nodes. An RDF graph may contain *ontology (schema)* triples; while there are interesting interactions between summarization and ontologies [ČGGM18], below we only focus on summarizing the non-schema triples, which make up the vast majority of all RDF graphs we encountered. Thus, we consider G consists exclusively of *type* triples and/or *data* triples (all those whose property is not *type*; we call these *data properties*).

An *RDF equivalence relation*, denoted $\equiv$, is a binary relation over the nodes of an RDF graph that is reflexive, symmetric and transitive. Given an equivalence relation $\equiv$, an *RDF graph quotient* is an RDF graph having (i) one node for each equivalence class of nodes; (ii) for each edge $n_1 \xrightarrow{a} n_2$, a summary edge $n_1^{\equiv} \xrightarrow{a} n_2^{\equiv}$, where $n_i^{\equiv}$, $i \in \{1, 2\}$, is the summary node corresponding to the equivalence class of n_i, also called *representative of* n_i. We call *representation function* a function that maps each graph node n_i to its representative

[1] https://project.inria.fr/rdfquotient

SBD'19, July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6766-0/19/07…$15.00
https://doi.org/10.1145/3323878.3325809

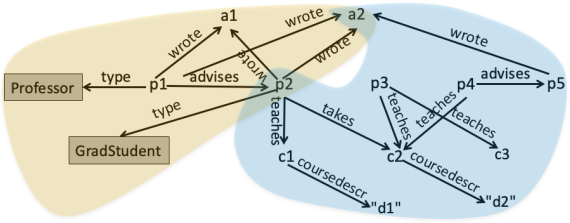

Figure 1: Sample RDF graph.

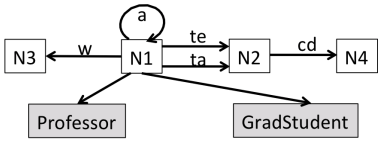

Figure 2: Weak summary of the graph in Figure 1.

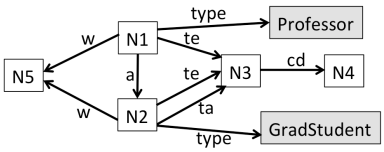

Figure 3: Strong summary of the graph in Figure 1.

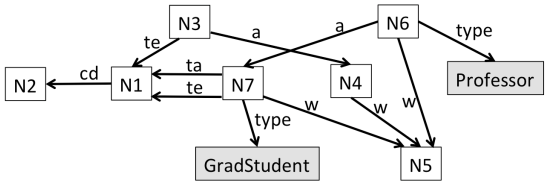

Figure 4: Typed weak summary of the graph in Figure 1.

($n_i^{\equiv}$). As stated above, the literature comprises many quotient graph summaries, which differ by their equivalence relations.

The equivalence relations we use are based on the concept of *property cliques*, which encodes a *transitive* relation of *edge label (property) co-occurrence on graph nodes*. Given an RDF graph G, two data properties *a*,*b* are in the same **source clique** iff: (*i*) there exists a G node *n* which is the source of *a* and *b* (i.e., $(n, a, x) \in$ G and $(n, b, y) \in$ G for some x and y), or (*iii*) there exists a data property *c* such that *c* is in the same source clique as *a*, and *c* is in the same source clique as *b*. Symmetrically, *a* and *b* are in the same **target clique** if there exists a G node which is the target of *a* and *b*, or a data property *c* which is in the same target clique as *a* and *b*. In Figure 1 (disregard the colored areas for now, their role will be explained later), the properties *advises* and *teaches* are in the same source clique due to p_4. The same holds for *advises* and *wrote* due to p_1; consequently, *advises* and *wrote* are also in the same source clique. Further, the graduate student p_2 teaches a course and takes another, thus *teaches*, *advises*, *wrote* and *takes* are all part of the same source clique. In this example, p_1, p_2, p_3, p_4, p_5 have the source clique SC_1 = {*advises*, *takes*, *teaches*, *wrote*}, c_1, c_2, c_3 have the source clique SC_2 = {*coursedescr*} and a_1, a_2 have the empty source clique $SC_3 = \varnothing$. Similarly, the target cliques are, respectively; TC_1 = {*advises*} for p_2, p_5, TC_2 = {*teaches*, *takes*} for c_1, c_2, c_3, TC_3 = {*coursedescr*} for d_1, d_2, TC_4 = {*wrote*} for a_1, a_2 and $TC_5 = \varnothing$ for p_1, p_3, p_4.

It is easy to see that the set of non-empty source (or target) cliques is a *partition over the data properties of an RDF graph* G. Moreover, if a G node *n* is source of some data properties, they are all in the same source clique; similarly, all the properties of which *n* is a target are in the same target clique. Based on these cliques, for any nodes n_1, n_2 of G, we define:

- n_1 is **weakly equivalent** to n_2, denoted $n_1 \equiv_W n_2$, iff n_1, n_2 have the same source clique *or* the same target clique;
- n_1 is **strongly equivalent** to n_2, denoted $n_1 \equiv_S n_2$, iff n_1, n_2 have the same source clique *and* the same target clique.

Furthermore, we decide that in any RDF equivalence relation, *any class node*, i.e., a URI *c* appearing in a triple of the form $(n, type, c)$, and *any property node*, i.e., a URI *p* appearing as a subject or object of *subproperty* triple, is (*i*) only equivalent to itself and (*ii*) represented by itself in any RDFQuotient summary. This ensures that RDF types (classes), which (when present) denote an important information that data producers added to help understand their RDF graphs, are preserved in the summary.

The equivalence relations $\equiv_W$ and $\equiv_S$ lead to the **weak**, respectively **strong** summaries, defined as *quotients of* G *through* $\equiv_W$, denoted $G_{/W}$, respectively, *through* $\equiv_S$, denoted $G_{/S}$. Figures 2 and 3 illustrate these on the sample graph in Figure 1. For brevity, in the figures we use *a*, *w*, *te*, *ta*, *cd* to denote respectively the properties *advises*, *writes*, *teaches*, and *coursedescr*.

A different flavor of RDFQuotient summaries can be defined to take into account in the first place the types attached to different nodes. We term *type-then-data* RDFQuotient summary, a summary that first groups the nodes that have the same set of types, and then applies an equivalence relation, such as weak or strong, to classify the remaining, untyped nodes. Formally:

- n_1 is **typed weakly equivalent** to n_2, denoted $n_1 \equiv_{TW} n_2$, iff n_1, n_2 are typed nodes and they have the same set of types or n_1, n_2 are untyped nodes and they are weakly equivalent;
- n_1 is **typed strongly equivalent** to n_2, denoted $n_1 \equiv_{TS} n_2$, iff n_1, n_2 are typed nodes and they have the same set of types or n_1, n_2 are untyped nodes and they are strongly equivalent.

Analogously, **typed weak** and **typed strong** summaries are defined as quotients through $\equiv_{TW}$, denoted $G_{/TW}$, and $\equiv_{TS}$, denoted $G_{/TS}$, respectively. Figure 4 shows the typed weak summary of sample graph in Figure 1. In our example, the $G_{/TW}$ and $G_{/TS}$ summaries coincide.

3 PARALLEL ALGORITHMS

A first idea is to simply partition G among the available nodes (machines), summarize each slice of the graph on its node, and then summarize again the union of these partial summaries to get the summary of G. Unfortunately, this may be incorrect. Assume that G is the graph in Figure 1 that is partitioned on two machines into G_1 and G_2, as shown with orange and blue regions, respectively. Notice that all *teaches* triples are in G_2. Summarizing the two subgraphs separately, $(G_1)_{/W}$ has a *wrote* edge, $(G_2)_{/W}$ has a *teaches* edge. However, the information that *wrote* and *teaches* had common sources in G is lost: using only $(G_1)_{/W}$ and $(G_2)_{/W}$, one cannot compute $G_{/W}$. A similar reasoning holds for $G_{/S}$, $G_{/TW}$ and $G_{/TS}$.

Therefore, we devised new parallel algorithms for building our summaries (Sections 3.1 to 3.3); they all assume a distributed storage and a MapReduce-like framework. Section 3.4 shows how to tailor our algorithms to the Spark framework used in our implementation. We assume the graph holds |G| triples and we have *M* machines at

our disposal. All the algorithms perform two preprocessing steps: (*i*) build the sets of class and property nodes of G, which must be preserved by summarization; (*ii*) **dictionary-encode** the RDF URIs and literals into integers, to manipulate less voluminous data. (This is quite standard in RDF data management works.)

3.1 Parallel computation of the strong summary

We compute the strong summary through a sequence of parallel processing jobs as follows.

(S1) We distribute *all (data and type)* triples of input graph equally among all the machines, e.g. using round robin approach, so that each m_i, $1 \leq i \leq M$ holds at most $\left\lceil \frac{|G|}{M} \right\rceil$ triples.

(S2) In a Map job, each machine m_i for a given data triple t = s p o **emits** two pairs: (s, (source, p, o)) and (o, (target, p, s)), where source and target are two constant tokens (labels). *The data and type triples initially distributed to each machine $m_1, \ldots, m_M$ are kept (persisted) on that machine throughout the computation*. All other partial results produced are discarded after they are processed, unless otherwise specified.

(S3) In the corresponding Reduce job, for each resource $r \in$ G, all the data triples whose subject or object is r are on a same machine m_i.
For each such r, m_i can infer the relationships (same source clique, same target clique) that hold between the data properties of G appearing on incoming and outgoing edges of r. Formally, a *property relation information (or PRI, in short)* between two properties a, b of G states that they are in the same source clique, or that they are in the same target clique. For instance, if m_i hosts the blue triples from Figure 1, where a = *teaches* and b = *takes*, the triples $p2$ *teaches* $c1$ and $p2$ *takes* $c2$ lead to a PRI of the form (*teaches*, *takes*, source).
We also emit PRIs for each property with itself, in order to prepare the necessary information so that all cliques are correctly computed in the steps below (even those consisting of a single data property).
The PRIs resulting from all the data triples hosted on m_i are de-duplicated locally at m_i.

(S4) Each machine **broadcasts** its PRIs to all other machines *while also keeping its own PRIs*. Observe that for k properties having, for instance, the same source, $\frac{k(k-1)}{2}$ PRIs can be produced. However, it suffices to broadcast $k-1$ among them (the others will be inferred by transitivity in step (S6)).

(S5) Based on this broadcast, each machine has the necessary information to compute the source and target cliques of G *locally*, and actually computes them[2].
At the end of this stage, the cliques are known *and will persist on each machine until the end of the algorithm*, but we still need to compute: (*i*) all the (source clique, target clique) pairs which actually occur in G nodes, and (*ii*) the representation function and (*iii*) the summary edges.

(S6) The representation function can now be locally computed on each machine as follows:

- For a given pair of source and target cliques (SC, TC), let N_{SC}^{TC} be an URI uniquely determined by SC and TC, such that a different URI is assigned to each distinct clique pairs: N_{SC}^{TC} will be the URI of the $G_{/S}$ node corresponding these source and target cliques.
- For each resource r stored on m_i, the machine identifies the source clique SC_r and target clique TC_r of r, and creates (or retrieves, if already created) the URI $N_{SC_r}^{TC_r}$ of the node representing r in $G_{/S}$.

(S7) Finally, we need to build the edges of $G_{/S}$.

(a) To summarize *data triples*, for each resource r whose representative N_r is known by m_i, and each triple (hosted on m_i) of the form $r\ p\ o$, m_i emits $(o, (p, N_r))$. This triple arrives on the machine m_j which hosts o and thus already knows N_o. The machine outputs the $G_{/S}$ triple N_r p N_o.

(b) To summarize *type triples*, for each resource r represented by N_r such that the type triple r *type* c is on m_i, the machine outputs the summary triple N_r *type* c.

The above process may generate the same summary triple more than once (and at most M times). Thus, a final duplicate-elimination step may be needed.

Algorithm correctness. The following observations ensure the correctness of the above algorithm's stages.

- Steps (S1) to (S4) ensure that each machine has the complete information concerning data properties being source- or target-related. Thus, each machine correctly computes the source and target cliques.
- Step (S3) ensures that each machine can correctly identify the source and target clique of the resources r which end up on that machine.
- The split of the triples in Step (S1) and the broadcast of source and target clique ensure that the last steps (computation of representation function and of the summary triples) yield the expected results.

3.2 Parallel computation of the weak summary

The algorithm for weak summarization starts with the steps (S1) - (S3) as above, then continues as shown below. In a nutshell, this algorithm exploits the observation that by definition of the weak summary, each data property occurs only once.

(W4) Instead of PRIs, the machines emit *Unification Decisions*. A unification decision between two data properties a, b, is of one of the following forms: (*i*) a,b have the same source node in $G_{/W}$; (*ii*) a,b have the same target node in $G_{/W}$; (*iii*) the source of a is the same as the target of b. For instance, in the blue region of Figure 1, two triples $p2$ *takes* $c2$, $c2$ *coursedescr* $d1$ lead to the UD "the target of *takes* is the same as the source of *coursedescr*"; similarly, $p4$ *teaches* $c2$, $p4$ *advises* $p5$ lead to the UD "the source of *teaches* is the same as the source of *advises*" etc. In the above, just like for the PRIs, a and b can be the same or they can be different.

(W5) Each machine broadcasts its unique set of UDs while also keeping its own. The number of UDs is bound by the number of properties pairs in G. Not all of the combinations are sent; a transitive closure is applied in step (W6).

[2] In practice: (*i*) this can be implemented e.g., using Union-Find; (*ii*) this is redundant as only one of them could have done it and broadcast the result.

(W6) Each machine has the necessary information to compute the nodes and edges of $G_{/W}$ as follows:
- Assume that for two sets IP, OP of incoming, respectively, outgoing data properties, we are able to compute a unique URI W_{IP}^{OP}, which is different for each distinct pair of sets.
- Build $G_{/W}$ with an edge for each distinct data property p in G; the source of this edge for property is $W_{\emptyset}^{\{p\}}$, while its target is $W_{\{p\}}^{\emptyset}$. All edges are initially disconnected, that is, there are initially $2 \times P$ nodes in $G_{/W}$.
- Apply each UD on the graph thus obtained, gradually fusing the nodes which are the source(s) and target(s) of the various data properties. This entails replacing each W node with one reflecting all its incoming and outgoing data properties known so far.

At the end of this process, each node has $G_{/W}$. We still need to compute the representation function.

(W7) On each machine holding a triple $r_1\ p\ r_2$, we identify the W nodes W_p, W^p in $G_{/W}$ which contain p in their outgoing, respectively, incoming property set. We output the $G_{/W}$ triple $W_p\ p\ W^p$.

(W8) The type summary triples are built exactly as in step (S7b).

3.3 Parallel computation of the typed strong and typed weak summaries

We now present the changes needed by the above algorithms to compute the typed counterparts of weak and strong summaries. The changes are needed in order to reflect the different treatment of the type triples. In particular, we introduce a new constant token `type` to be sent in step (S2). We emit pairs corresponding to type triples only in the forward direction, e.g., we send $(s, (\texttt{type}, p, o))$ but not $(o, (\texttt{type}, p, s))$. We do not emit pairs with tokens `source` nor `target` for type triples, as typed nodes *do contribute to property cliques* (recall Section 2). The type triples are then cached (kept) at each machine, and not used until the step (S6) in the strong summarization algorithm, respectively (W6) in the weak one.

To determine the representative of a typed node, each machine that received some type triples groups them by the subject and creates a temporary class set IDs based on the types it knows for each subject. Then, a step similar to (S7a) is needed to disseminate information about each such typed nodes, say n at machine m. Any machine m' which has received some triples of whom n is a subject/object, but has no type triples about n, needs to know it is typed, thus omit it from the clique computation.

3.4 Apache Spark implementation specifics

We used Spark 2.3.0 with Hadoop's YARN scheduler 2.9.0. We implemented our algorithms in Scala 2.11.

A Spark cluster has a set of *worker nodes* and a *driver*, which coordinates the run of an application and communicates with cluster manager (e.g. YARN scheduler). A Spark *executor* is a process working on a piece of data in a worker node. Each Spark application consists of *jobs* that are divided into *stages*, each divided into *tasks*.

Basic terminology. Spark relies on *Resilient Distributed Datasets (RDDs)*, which are fault-tolerant and immutable collections of data distributed among the cluster nodes. Spark also supports *broadcast variables*, i.e., collections of the data that are first gathered at the driver and then are broadcast (copies shipped over network) to all the cluster nodes. These collections are immutable and can only be used for local lookups.

Adapting our algorithms to Spark. We adapt our distributed algorithms to Spark's RDD-based computation model as follows. Each step of an algorithm consumes an RDD and builds another one. First of all, we load the input graph into an RDD called `graph`. Then, we preprocess it in order to create the RDDs: `dictionary`, `reverseDictionary`, `nonDataNodesBlacklist` and `encodedTriples`. `dictionary` maps G nodes to their integer encodings, `reverseDictionary` is its reverse map and `encodedTriples` are the iG triples encoded into integers. We collect the class and property nodes in `schema-Nodes` RDD; this (very small) collection is broadcast to all nodes.

Then, we create an RDD called `nodesGrouped` that is a map from G nodes to the set of their incoming and outgoing edges. Next, in the weak algorithm we create `unificationDecisions` RDD that is a collection of unification decisions, in the strong algorithm we create respectively a `propertyRelationInformation` RDD. For those two RDDs we exclude (pre-filter) class and property nodes, and in case of typed summaries we exclude typed nodes too.

The RDDs `unificationDecisions` and `propertyRelationInformation` are gathered at the driver, which computes the source and target cliques of G. The driver then broadcasts the source and target clique IDs of each property to all the nodes. This allows to create the summary nodes and to create the summary edges. The algorithms that build $G_{/S}$, $G_{/TW}$ and $G_{/TS}$ also use an RDD called `representationFunction`, which stores a mapping between the input graph node and its summary representative. This map is filled by each algorithm, and then used on each machine to emit, for each triple of the form $n\ p\ m$ that it stores, the corresponding summary triple $n_{rep}\ p\ m_{rep}$, where n_{rep}, m_{rep} are the representatives of n and m respectively. The weak summarization algorithm does not need this map; as explained in Section 3.2 (step (W6) and below), it can create the summary triple representing $n\ p\ m$ directly based on p.

Finally, in all the algorithms we need to decode the properties (edge labels). We do it by joining the encoded summary triples with the `reverseDictionary` in order to replace each encoded property with its full value from G.

4 EXPERIMENTAL EVALUATION

Cluster setup. We are using a cluster of 6 machines, each of which is equipped with an Intel Xeon CPU E5-2640 v4 @2.40GHz and 124GB RAM. Each machine has 20 physical CPU cores. However, we use the cluster with a hyper-threading option enabled, which gives an operating system effectively 40 cores for resource allocation. All machines in this cluster are connected to a switch using 10 Gigabit Ethernet. We give to Spark and YARN 100GB of RAM and 36 cores at each machine. We leave some fraction of the memory (remaining 24GB) and 4 CPU cores for the operating system.

RDF graphs. To compare algorithm behavior for different data sizes, we used synthetic benchmark graphs from the BSBM [BS09] benchmark graph, of 1M, 10M, respectively 100M triples. We found

that in these graphs, **61% to 68%** of the nodes were **untyped**, while the others have at least one type. On the one hand, this justifies the need for summaries (such as $G_{/W}$ and $G_{/S}$) which do not require all nodes to be typed; on the other hand, there is also a sizable share of typed nodes, which makes the computation of $G_{/TW}$ and $G_{/TW}$ significantly different, on these graphs, than that of $G_{/W}$ and $G_{/S}$. For the BSBM graphs, as for many other RDF graphs we experimented with and whose summaries are depicted online[1], the RDFQuotient summaries are quite small: from 179 triples ($G_{/W}$, BSBM1M) to 3325 triples ($G_{/TS}$, BSBM100M)[3]. Thus, the information (PRIs and UDs) broadcast by our algorithms is also quite compact.

Configuration. We recall here that M denotes the number of machines (Spark workers). Spark parameters relevant for our performance analysis were set as follows:

- The number of cores per machine C_M: we picked $C_M = 36$ (all available cluster resources). We pick all the available resources in order to maximize the performance.
- The number of cores per executor C_E. Following Spark guidelines, unless otherwise specified, we pick $C_E = 4$.
- The available memory per machine R_M: we set it to 100 GB.
- The amount of memory per executor $R_E \geq 2.78$ GB. There can be at most C_M executors per machine, so they will use at least $\frac{R_M}{C_M}$, in general $R_E = \frac{R_M \cdot C_E}{C_M}$. We set the lower bound for R_E as a minimal memory of the YARN container, within the memory limit for executor we need to hold out around 1GB for a memory overhead (memory for JVM in YARN).
- The number of executors per machine E_M, typically 9.
- The total number of executors E is computed as $E_M \cdot M$.
- The number of partitions $P = \alpha E$. Following existing recommendations[4], we set $\alpha = 4$.

Speed up through parallelism. We fix the following parameter values $M = 5$, $C_E = 4$, $E_M = 9$, $R_E = 11$GB, $P = 180$ and we vary E, the number of executors, by using more or less machines.

Figure 5 shows the computation time (in minutes) of the parallel algorithms with respect to the number of executors, on a BSBM graph of 10M triples. The time here includes loading, preprocessing, summarization and saving the output file containing the graph summary. We can see that parallelization helps decrease the running time: there is a big gain from 9 to 18 executors, and smaller gains as the parallelism increases (in our setting, benefits basically disappear/amortize going from 36 to 45).

Figure 6 zooms to show *only the summarization time* (in seconds) for the same set of computations. We see that the algorithm is overall efficient, i.e., summarization itself is quite fast (seconds as opposed to minutes for the overall computation). We investigated and found large parts of the computation time in Figure 6 are spent: (*i*) encoding the RDF triples into triples of integers; this is by far the dominant-cost operation. To do this, we need to identify the (duplicate-free) set of node labels from G, operation which we implemented using Spark's `distinct()` function, which eliminates

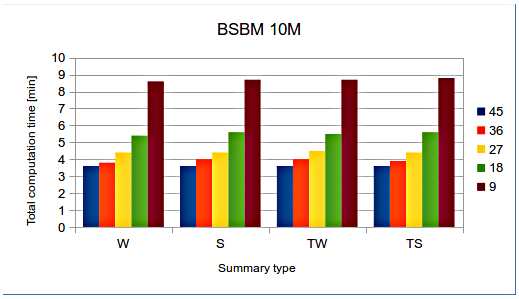

Figure 5: Total computation time for various number of executors.

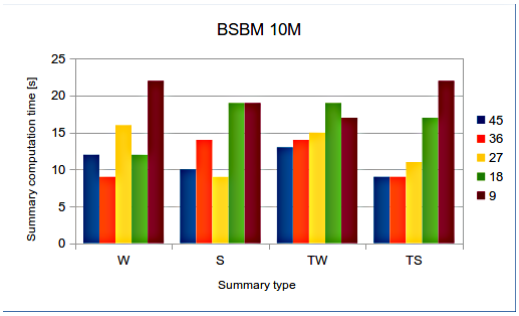

Figure 6: Summarization time for various number of executors.

duplicates from an RDD that is (as usual) spread across the nodes. It involves communication between nodes, thus its high latency. We note, however, that our initial implementation, which lacked the encoding and worked directly with URIs and literals from G, was way slower (and encountered memory issues). Thus, we believe the encoding cost is worth paying; (*ii*) the pre-computation of the RDF class and property nodes from a given graph G. This is also implemented by each machine adding its schema nodes to an RDD and then calling `distinct()`.

Coming back to Figure 6, we see that despite the need for a global synchronization step, parallelism (increasing the number of executors) clearly shortens the summarization time. A few data points are counter-intuitive, e.g., using 45 executors takes slightly more than using 36 for weak summarization, or using 36 takes more than using 27 for strong summarization. We believe these points are due to the (random) way in which the data is distributed among the executors, which in turn determines when the broadcast operation (steps (S4) and (W5)) is finished. While this distribution is simply made in round-robin fashion, we find that overall parallelism clearly helps, which can be seen e.g. by comparing the times for the lowest parallelism degrees (9, 18) and the highest (36, 45); this holds even more when put back in the perspective of the overall time (Figure 5).

Scaling in the dataset size. Our next experiment studies the impact of the RDF graph size on the summarization time. We have repeated the above experiments for BSBM graphs of 1M and 100M triples, leading to time measures for 1M, 10M, and 100M triples, that we analyze together to determine how the algorithms scale. Figure 7 shows the total computation time (in minutes) while Figure 8 zooms in just on the time to compute the summary. Note the logarithmic scale of the y axis in both graphs.

The total time (Figure 7) which is dominated by the data preprocessing and thus I/O-bound, grows linearly with the data size.

[3] Our online summary visualizations[1] also apply two post-processing steps to make summaries even smaller: group nodes by their most general type (à la [GM18]) and draw leaf nodes as attributes of their parents. This is how the typed strong summary of the BSBM 100M graph is depicted with only 10 edges!

[4] http://spark.apache.org/docs/latest/tuning.html#level-of-parallelism, https://stackoverflow.com/questions/31359219/relationship-between-rdd-partitions-and-nodes

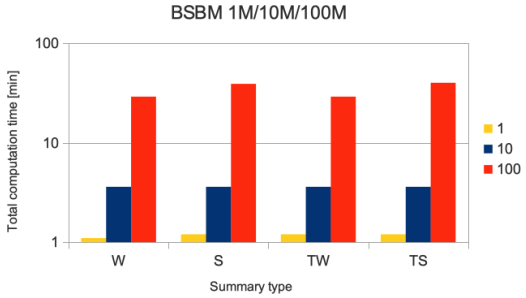

Figure 7: Total computation time for datasets of different size.

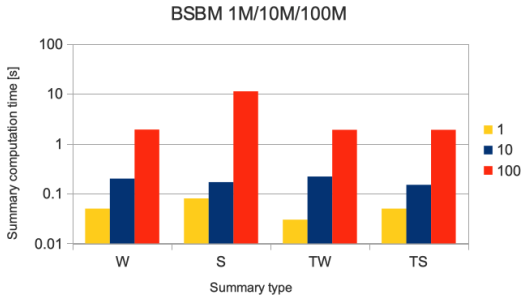

Figure 8: Summarization time for datasets of different size.

The summarization time alone (Figure 8) grows almost linearly with the data size for weak, typed weak and typed strong summarization, whereas the growth of the summarization time is super-linear for the strong summary. We have not been able to determine precisely the cause. Recalling also Figure 6, we believe there is some variability in Spark in-memory execution performance, that we were not able to control precisely. However, considering (also) the fact that summarization itself takes a relatively small part of the total time, we can conclude that overall our parallel algorithms scale up well (basically linearly) with the data size.

5 RELATED WORK AND CONCLUSION

Graph summarization has a long history and many applications; a recent survey dedicated to RDF summarization is [CGK+18]. Closest to us are existing quotient summaries of RDF graphs, notably those based on *bisimulation* [MS99, CLO03, KBNK02, CMRV10, SNLPZ13, TLR13], possibly bounded to some distance k; [CDT13, KC15, CFKP15] parallelize the computation of bisimulation summaries. In [CDT13], a summary groups RDF nodes in a distinct group: by their type set or by their outgoing properties (these two are not quotients, since a node may belong to multiple groups); or by types *and* properties. Grouping by the type set (which collapses all untyped nodes) leads to the smallest summaries (most of which still have from 30 to 10^5 edges), but it can lead to loss of information. All the other summaries in [CDT13] have thousands of edges. Characteristic sets, proposed as support for RDF cardinality estimation [NM11, GN14], can also be cast as a form of quotient summary, albeit much less compact than ours. Even though [NM11, GN14, CDT13] ignore incoming properties (useful structural information about the nodes), they lead to summaries impractical for visualization due to the lack of the *transitive* aspect built into our property cliques (Section 2). Dataguides [GW97] are non-quotient structural summaries, which may be larger than the original graph and may take exponential time to build.

We had demonstrated [ČGM15b] and (informally) presented $G_{/W}$ and $G_{/TW}$ in a short "work in progress" paper [ČGM15a], with procedural definitions (not as quotients). [ČGM17] discusses the interplay between summarization and RDF graph saturation. In [GM18], we define a type-then-data summarization variant based on type generalization; it is orthogonal (and complementary) to this work. Our centralized algorithms for incrementally summarizing RDF graphs has been demonstrated recently [GGM19].

In this work we presented novel parallel algorithms for computing RDFQuotient graph summaries that let the computations scale up to large graphs. We studied the design of the algorithms dedicated to a distributed setting, and we implemented them in the Spark framework. We assessed the performance and showed benefits of the parallel algorithms by experimenting with datasets of size different by orders of magnitude, and by varying the degree of parallelism.

REFERENCES

[BS09] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2), 2009.
[CDT13] Stéphane Campinas, Renaud Delbru, and Giovanni Tummarello. Efficiency and precision trade-offs in graph summary algorithms. In *IDEAS*, 2013.
[CFKP15] Mariano P. Consens, Valeria Fionda, Shahan Khatchadourian, and Giuseppe Pirrò. S+EPPs: Construct and explore bisimulation summaries + optimize navigational queries; all on existing SPARQL systems (demonstration). *PVLDB*, 8(12), 2015.
[ČGGM18] Šejla Čebirić, François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. Compact Summaries of Rich Heterogeneous Graphs. Inria Research Report, available at https://hal.inria.fr/hal-01325900, July 2018.
[CGK+18] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. Summarizing Semantic Graphs: A Survey. *The VLDB Journal*, 2018.
[ČGM15a] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. Query-oriented summarization of RDF graphs. In *BICOD*, 2015.
[ČGM15b] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. Query-oriented summarization of RDF graphs (demonstration). *PVLDB*, 8(12), 2015.
[ČGM17] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. A framework for efficient representative summarization of RDF graphs. In *ISWC (poster)*, 2017.
[CLO03] Qun Chen, Andrew Lim, and Kian Win Ong. $D(K)$-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.
[CMRV10] Mariano P. Consens, Renée J. Miller, Flavio Rizzolo, and Alejandro A. Vaisman. Exploring XML web collections with DescribeX. *TWEB*, 4(3), 2010.
[GGM19] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. Incremental structural summarization of RDF graphs (demo). In *EDBT 2019*, Lisbon, Portugal, March 2019.
[GM18] Pawel Guzewicz and Ioana Manolescu. Quotient RDF Summaries Based on Type Hierarchies. In *DESWeb (Data Engineering meets the Semantic Web) Workshop*, Paris, France, April 2018.
[GN14] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, 2014.
[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
[HHK95] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.
[KC15] Shahan Khatchadourian and Mariano P. Consens. Constructing bisimulation summaries on a multi-core graph processing framework. In *GRADES*, 2015.
[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, 1999.
[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.
[SNLPZ13] Alexander Schätzle, Antony Neu, Georg Lausen, and Martin Przyjaciel-Zablocki. Large-scale bisimulation of RDF graphs. In *SWIM*, 2013.
[TLR13] Thanh Tran, Günter Ladwig, and Sebastian Rudolph. Managing structured and semistructured RDF data using structure indexes. *IEEE TKDE*, 25(9), 2013.