Approximate Querying on Property Graphs

Stefania Dumbrava[1], Angela Bonifati[2], Amaia Nazabal Ruiz Diaz[2], and Romain Vuillemot[3]

[1] ENSIIE Évry & CNRS Samovar
stefania.dumbrava@ensiie.fr
[2] University of Lyon 1 & CNRS LIRIS
{angela.bonifati,amaia.nazabal-ruiz-diaz}@univ-lyon1.fr
[3] École Centrale Lyon & CNRS LIRIS
romain.vuillemot@ec-lyon.fr

Abstract. Property graphs are becoming widespread when modeling data with complex structural characteristics and enriching edges and nodes with a list of properties. In this paper, we focus on the approximate evaluation of counting queries involving recursive paths on property graphs. As such queries are already difficult to evaluate over pure RDF graphs, they require an ad-hoc graph summary for their approximate evaluation on property graphs. We prove the intractability of the optimal graph summarization problem, under our algorithm's conditions. We design and implement a novel property graph summary suitable for the above queries, along with an approximate query evaluation module. Finally, we show the compactness of the obtained summaries as well as the accuracy of answering counting recursive queries on them.

1 Introduction

A tremendous amount of information stored in the LOD can be inspected, by leveraging the already mature query capabilities of SPARQL, relational, and graph databases [14]. However, arbitrarily complex queries [2, 3, 7], entailing rather intricate, possibly recursive, graph patterns prove difficult to evaluate, even on small-sized graph datasets [4, 5]. On the other hand, the usage of these queries has radically increased in real-world query logs, as shown by recent empirical studies on SPARQL queries from large-scale Wikidata and DBPedia corpuses [8, 17]. As a tangible example of this growth, the percentage of SPARQL property paths has increased from 15% to 40%, from 2017 to beginning 2018 [17], for user-specified Wikidata queries. In this paper, we focus on regular path queries (RPQs) that identify paths labeled with regular expressions and aim to offer an approximate query evaluation solution. In particular, we consider counting queries with regular paths, which are a notable fragment of graph analytical queries. The exact evaluation of counting queries on graphs is $\#P-$complete [21] and is based on another result on enumeration of simple graph paths. Due to this intractability, an *efficient and highly-accurate approximation* of these queries is desirable, which we address in this paper.

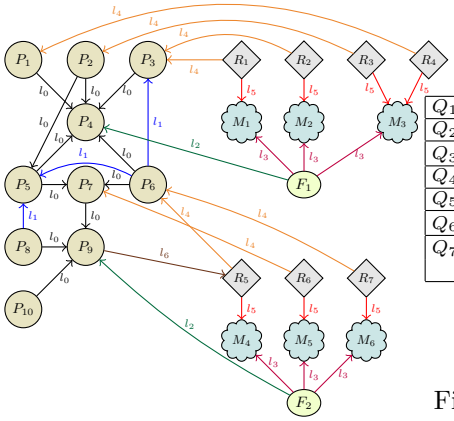


Fig. 1: Example Social Graph $\mathcal{G}_{SN}$

$Q_1(l_5)$	$Ans(count(\_)) \leftarrow l_5(\_,\_)$
$Q_2(l_2)$	$Ans(count(\_)) \leftarrow l_2?(\_,\_)$
$Q_3(l_0)$	$Ans(count(\_)) \leftarrow l_0^+(\_,\_)$
$Q_4(l_0)$	$Ans(count(\_)) \leftarrow l_0^*(\_,\_)$
$Q_5(l_4, l_1)$	$Ans(count(\_)) \leftarrow l_4 + l_1(\_,\_)$
$Q_6(l_4, l_5)$	$Ans(count(\_)) \leftarrow l_4^- \cdot l_5(\_,\_)$
$Q_7(l_4, l_5)$	$Ans(count(x)) \leftarrow l_4^- \cdot l_5(x,\_), \geq (x.age, 18), \leq (x.age, 24).$

Forum · Message · Reply · Person
– knows (l_0) – follows (l_1) – moderates (l_2) – contains (l_3)
– authors (l_4) – replies (l_5) – reshares (l_6)

Fig. 2: Targeted Advertising Queries

Approximate query processing on relational data and the related sampling methods are not applicable to graphs, since the adopted techniques are based on the linearity assumption [15], i.e., the existence of a linear relationship between the sample size and execution time, typical of relational query processing. As such, we design a novel query-driven graph summarization approach tailored for *property graphs*. These significantly differ from RDF and relational data models, as they attach data values to property lists on both nodes and edges [7].

To the best of our knowledge, ours is the first work on approximate property graph analytics addressing counting estimation on top of navigational graph queries. We illustrate our query fragment with the running example below.

Example 1 (Social Network Advertising). Let $\mathcal{G}_{SN}$ (see Fig. 1) be a property graph (see Sec. 2) encoding a social network, whose schema is inspired by the LDBC benchmark [12] [4]. Entities are people (type `Person`, P_i) that *know* (l_0) and/or *follow* (l_1) either each other or certain forums (type `Forum`, F_i). These are *moderated* (l_2) by specific persons and can *contain* (l_3) messages/ads (type `Message`, M_i), to which persons can *author* (l_4) other messages in *reply* (l_5).

We focus on a RPQ [3, 23] dialect with counting, capturing following query types ($Q_1 - Q_7$) (see Fig. 2): *(1) Simple/Optional Label.* The number of pairs satisfying Q_1, i.e., $()\xrightarrow{l_5}()$, counts the ad *reactions*, while that for Q_2, i.e., $()\xrightarrow{l_2?}()$, indicates the number of *potential moderators*. *(2) Kleene Plus/Kleene Star.* The number of the *connected/potentially connected acquaintances* is the count of node pairs satisfying Q_3, i.e., $() \leftarrow l_0^+()$, respectively, Q_4, i.e., $() \leftarrow l_0^*()$. *(3) Disjunction.* The number of the *targeted subscribers* is the sum of counting all node pairs satisfying Q_5, i.e., $()\xleftarrow{l_4}()$ or $()\xleftarrow{l_1}()$. *(4) Conjunction.* The *direct reach* of a company via its page ads is the count of node pairs satisfying Q_6, i.e., $()\xleftarrow{l_4}()\xrightarrow{l_5}()$. *(5) Conjunction with Property Filters.* Recommendation systems can further refine the Q_6 estimates. Thus, one can compute the *direct demographic reach* and target people within an age group, e.g., 18-24, by counting all node pairs that satisfy Q_7, i.e. $(x)\xleftarrow{l_4}()\xrightarrow{l_5}()$, s.t $x.age \geq 18$ and $x.age \leq 24$.

[4] One of the few benchmarks currently available for generating property graphs.

Contributions. Our paper provides the following main contributions:

- We design a property graph summarization algorithm for approximately evaluating counting regular path queries (Sec. 3).
- We prove the intractability of the optimal graph summarization problem under the conditions of our summarization algorithm (Sec. 3).
- We define a query translation module, ensuring that queries on the initial and summary property graphs are expressible in the same fragment (Sec. 4).
- Based on this, we experimentally exhibit the small relative errors of various workloads, in the expressive query fragment from Example 1. We measure the relative response time between estimating counting recursive queries on summaries and on the original graphs. For non-recursive queries, we compare with SumRDF [19], a baseline graph summary for RDF datasets (Sec. 5).

In Section 2, we revisit the property graph model and query language. We present related work in Section 6 and conclude the paper in Section 7.

2 Preliminaries

Graph Model. We take the *property graph model* (PGM) [7] as our foundation. Graph instances are multi-edge digraphs; its objects are represented by typed, data vertices and their relationships, by typed, labeled edges. Vertices and edges can have any number of *properties* (key/value pairs). Let L_V and L_E be disjoint sets of vertex (edge) labels and $\mathcal{G} = (V, E)$, with $E \subseteq V \times L_E \times V$, a *graph instance*. Vertices $v \in V$ have an id label, l_v, and a set of property labels (attributes, l_i), each with a (potentially undefined) term value. For $e \in E$, we use the binary notation $e = l_e(v_1, v_2)$ and abbreviate v_1, as $e.1$, and v_2, as $e.2$. We denote the number of occurrences of l_e, as $\#l_e$, and the set of all edge labels in $\mathcal{G}$, as $\Lambda(\mathcal{G})$. Other key notations henceforth used are given in Table 1.

$\mathcal{G}, \Phi, v, V, e, E$	$\triangleq$ Graph, graph partitioning, vertex (set), edge (set)
$\mathcal{G}^*, v^*, V^*, e^*, E^*$	$\triangleq$ S-graph, s-node (set), s-edge (set)
$\hat{\mathcal{G}}, \hat{v}, \hat{V}, \hat{e}, \hat{E}$	$\triangleq$ H-graph, h-node (set), h-edge (set)
$\lambda(\mathcal{G})$	$\triangleq$ label on which a graph $\mathcal{G}$ is maximally l-connected
$\Lambda_d(v^*), d \in \{1, 2\}$	$\triangleq$ set of edge labels with direction d w.r.t v^* (1-incoming, 2-outgoing)

Table 1: Notation Table

Graph Query Language. To query the above property graph model, we rely on an RPQ [10,11] fragment with aggregate operators (see Fig. 3). RPQs correspond to SPARQL 1.1 property paths and are a well-studied query class tailored to express *graph patterns* of one or more *label-constrained reachability paths*. For labels l_e^i and vertices v_i, the *labeled path* π, corresponding to $v_1 \xrightarrow{l_e^1} v_2 \ldots v_{k-1} \xrightarrow{l_e^k} v_k$, is the concatenation $l_e^1 \cdot \ldots \cdot l_e^k$. In their full generality, RPQs allow one to select vertices connected via such labeled paths in a *regular language* over

Clauses	C	::=	$A \leftarrow A_1, \ldots, A_n \mid Q \leftarrow A_1, \ldots, A_n$
Queries	Q	::=	$Ans(count(\_)) \mid Ans(count(l_v)) \mid Ans(count(l_{v_1}, l_{v_2}))$
Atoms	A	::=	$\pi(l_{v_1}, l_{v_2}) \mid op(l_{v_1}.l_i, l_{v_2}.l_j) \mid op(l_{v_1}.l_i, k), op \in \{<, \leq, >, \geq\}, k \in \mathbb{R}$
Paths	π	::=	$\epsilon \mid l_e \mid l_e? \mid l_e^{-1} \mid l_e^* \mid l_{e_1} \cdot l_{e_2} \mid \pi + \pi$

Fig. 3: Graph Query Language

L_E. We restrict RPQs to handle *atomic paths* – bi-directional, optional, single-labeled (l_e, $l_e?$, and l_e^-) and transitive single-labeled (l_e^*) – and *composite paths* – conjunctive and disjunctive composition of atomic paths ($l_e \cdot l_e$ and $\pi + \pi$). While not as general as SPARQL, our fragment already captures more than 60% of the property paths found in practice in SPARQL query logs [8]. Moreover, it captures property path queries, as found in the large Wikidata corpus studied in [9]. Indeed, almost all the property paths in the considered logs contain Kleene-star expressions over *single* labels. In our work, we enrich the above query classes with the *count* operator and support basic graph reachability estimates.

3 Graph Summarization

We introduce a novel algorithm that summarizes any property graph into one tailored for approximately counting reachability queries. The key idea is that, as nodes and edges are compressed, informative properties are iteratively added to the corresponding newly formed structures, to enable accurate estimations.

The **grouping phase** (Sec. 3.1) computes Φ, a label-driven $\mathcal{G}$-partitioning into *subgroupings*, following the connectivity on the most frequent labels in $\mathcal{G}$. A first **s**ummarization collapses the vertices and inner-edges of each subgrouping into ***s**-nodes* and the edges connecting s-nodes, into ***s**-edges*. The **merge phase** (Sec. 3.2), based on further label-reachability conditions, specified by a **h**euristic mode m, collapses s-nodes into ***h**-nodes* and *s-edges* into ***h**-edges*.

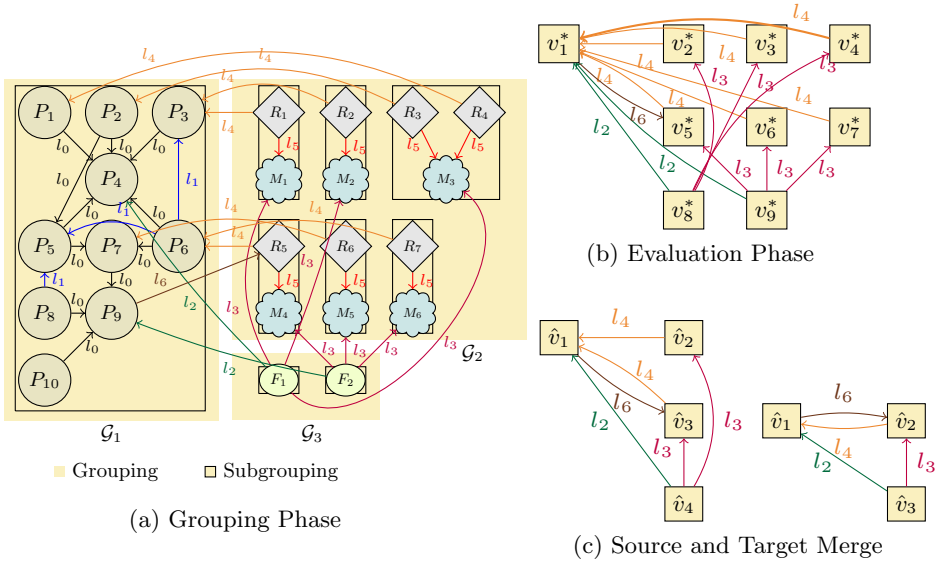
3.1 Grouping Phase

For each frequently occurring label l in $\mathcal{G}$, in descending order, we iteratively partition $\mathcal{G}$ into Φ, containing components that are connected on l, as below.

Definition 1 (Maximal L-Connectivity). *A $\mathcal{G}$-subgraph[5], $\mathcal{G}' = (V', E')$, is* maximally l-connected*, i.e., $\lambda(\mathcal{G}') = l$, iff 1) $\mathcal{G}'$ is weakly-connected, 2) removing any l-labeled edge from E', there exists a V' node pair not connected by a l^+-labeled undirected path, 3) no l-labeled edge connects a V' node to $V \setminus V'$.*

Example 2. In Fig. 1, $\mathcal{G}_1$ is maximally l_0-connected, since it is weakly-connected, not connected by an l_0-labeled edge to the rest of $\mathcal{G}$, and such that, by removing $P_8 \xrightarrow{l_0} P_9$, no undirected, l_0^+-labeled path unites P_8 and P_9.

[5] $\mathcal{G}'$ is a *$\mathcal{G}$-subgraph* iff $V' \subseteq V$ and $E' \subseteq E$ and is *weakly connected* iff there exists an undirected path between any pair of vertices.



(a) Grouping Phase

(b) Evaluation Phase

(c) Source and Target Merge

Fig. 4: Summarization Phases for $\mathcal{G}_{SN}$

We call each such component a *subgrouping*. The procedure (see Alg.1) computes, as the first *grouping*, all the subgroupings for the most frequent label, l_1, and then identifies those corresponding to the rest of the graph and to l_2. At the end, all remaining nodes are collected into a final subgrouping. We illustrate this in Fig. 4, on the running example below.

Example 3 (Grouping). In Fig. 1, $\#l_0 = 11$, $\#l_1 = 3$, $\#l_2 = 2$, $\#l_3 = 6$, $\#l_4 = \#l_5 = 7$, $\#l_6 = 1$, and $\overrightarrow{\Lambda(\mathcal{G})} = [l_0, l_5, l_4, l_3, l_1, l_2, l_6]$, as $\#l_4 = \#l_5$ allows arbitrary ordering. We add the maximal l_0-connected subgraph, $\mathcal{G}_1$, to Φ. Hence, $V = \{R_{i\in\overline{1,7}}, M_{i\in\overline{1,6}}, F_1, F_2\}$. Next, we add $\mathcal{G}_2$, regrouping the maximal l_5-connected subgraph. Hence, $V = \{F_1, F_2\}$; we add $\mathcal{G}_3$ and output $\Phi = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3\}$.

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

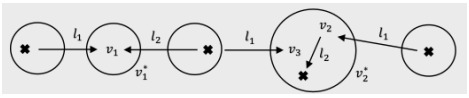
Input: $\mathcal{G}$ – a graph; **Output:** Φ – a graph partitioning

1: $n \leftarrow |\Lambda(\mathcal{G})|$, $\overrightarrow{\Lambda(\mathcal{G})} \leftarrow [l_1, \ldots, l_n]$, $\Phi \leftarrow \emptyset$, $i \leftarrow 1$ ▷*Descending frequency label list* $\overrightarrow{\Lambda(\mathcal{G})}$
2: **for all** $l_i \in \overrightarrow{\Lambda(\mathcal{G})}$ **do** ▷*Label-driven partitioning computation*
3: $\quad \Phi \leftarrow \Phi \cup \{\mathcal{G}^*_k = (V^*_k, E^*_k) \subseteq \mathcal{G} \mid \lambda(\mathcal{G}^*_k) = l_i\}$ ▷*Maximally* l_i*-Connected Subgraphs*
4: $\quad V \leftarrow V \setminus \{v \in V^*_k \mid k \in \mathbb{N}\}$ ▷*Discard Already Considered Nodes*
5: $\quad i \leftarrow i + 1$
6: $\Phi \leftarrow \Phi \cup \{\mathcal{G}_i = (V^*_i, E^*_i) \subseteq \mathcal{G} \mid V^*_i = V \setminus V^*\}$ ▷*Collect Remains in Final Subgroup*
7: **return** Φ

A $\mathcal{G}$-partitioning Φ (see Fig. 4a) is transformed into a *s-graph* $\mathcal{G}^* = (V^*, E^*)$ (see Fig. 4b). As such, each s-node gathers all the nodes and inner edges of a Φ-subgrouping, $\mathcal{G}^*_j$, and each *s-edge*, all same-labeled *cross-edges* (edges between pairwise distinct s-nodes). During this phase, we compute analytics concerning the regrouped entities. We leverage PGM's expressivity to internalize these as

properties, e.g., Fig. 5 (right)[6]. Hence, to every s-edge, e^*, we attach $EWeight$, its number of compressed edges, e.g., in Fig. 4b, all s-edges have weight 1, except $e^*(v_4^*, v_1^*)$, with weight 2. To every s-node, v^*, we attach properties concerning: *(1) Compression.* $\mathcal{V}Weight$ and $EWeight$ store its number of *inner vertices/edges*. *(2) Inner-Connectivity.* The percentage of its l-labeled inner edges is $LPercent$ and the number of its vertex pairs, connected with an l-labeled edge, is $LReach$. These first two types of properties will be useful in Sec.4, for estimating Kleene paths, as the labels of inner-edges in s-nodes are not unique, e.g., both l_0 and l_1 appear in v_1^*. *(3) Outer-Connectivity.* For pairs of labels and direction indices with respect to v^* ($d = 1$, for incoming edges, and $d = 2$, for outgoing ones), we compute *cross-connectivity*, $CReach$, as the number of binary cross-edge paths that start/end in v^*. Analogously, we record that of binary *traversal paths*, i.e., formed of an inner v^* edge and of a cross-edge, as $TReach$. Also, for a label l and given direction, we store, as V_F, the number of *frontier vertices* on l, i.e., that of v^* nodes at either endpoint of a l-labeled s-edge.

We can thus record *traversal connectivity* information, $LPart$, dividing the number of traversal paths by that of the frontier vertices on the cross-edge label. Intuitively, this is due to the fact that, traversal connectivity, as opposed to cross connectivity, also needs to account for the "dispersion" of the inner-edge label of the path, within the s-node it belongs to. For example, for a traversal path $l_c \cdot l_i$, formed of a cross-edge, l_c, and an inner one, l_i, not all frontier nodes l_c are endpoints of l_i labeled inner-edges, as we will see in the example below.



$\mathcal{V}Weight$	$v_1^* \mapsto 10, v^*_{\{2,3,5,6,7\}} \mapsto 2,$ $v_4^* \mapsto 3, v^*_{\{8,9\}} \mapsto 1$
$EWeight$	$v_1^* \mapsto 14, v^*_{\{2,3,5,6,7\}} \mapsto 1,$ $v_4^* \mapsto 3, v^*_{\{8,9\}} \mapsto 0$
$LReach$	$(v_1^*, l_0) \mapsto 11, (v_1^*, l_1) \mapsto 3$
$LPercent$	$(v_1^*, l_0) \to 79, (v_1^*, l_1) \to 21$

Fig. 5: Selected Properties for Fig.4b (right); Frontier Vertices (left);

Example 4 (Outer-Connectivity). Fig.5 (left) depicts a stand-alone example, such that circles denote s-nodes, labeled arrows denote the s-edges relating them, and crosses represent nameless vertices, as we only label relevant ones, for simplicity. We use this configuration to illustrate analytics regarding cross and traversal connectivity on labels l_1 and l_2. For instance, as we will see in Sec.4, when counting $l_1 \cdot l_2^-$ *cross-edge paths*, we will look at the $CReach$ s-node properties mentioning these labels and note that there is a single such one, i.e., that corresponding to l_1 and l_2 appearing on edges incoming v_1^*, i.e., $CReach(v_1^*, l_1, l_2, 1, 1) = 1$. When counting $l_1 \cdot l_2$ *traversal paths*, for the case when l_1 appears on the cross-edge, we will look at the properties of s-nodes containing l_2 inner-edges. Hence, for v_2^*, we note that there is a single such path, formed by an outgoing l_2 edge and incoming l_1 edge, as $TReach(v_2^*, l_1, l_2, 1, 1) = 1$. To estimate the *traversal connectivity* we will divide this by the number of frontier vertices on incoming l_1 edges. As, $V_F(v_2^*, l_1, 1) = \{v_2, v_3\}$, we have that $LPart(v_2^*, l_1, l_2, 1, 1) = 0.5$.

[6] All corresponding formulas are provided in the additional material.

3.2 Merge Phase

We take as input the graph computed by Alg.1, and a label set and output a compressed graph, $\hat{\mathcal{G}} = (\hat{V}, \hat{E})$. During this phase, sets of *h-nodes*, $\hat{V}$, and *h-edges*, $\hat{E}$, are created. At each step, as previously, $\hat{\mathcal{G}}$ is enriched with approximation-relevant precomputed properties (see Sec. 4).

Each *h-node*, $\hat{v}$, merges all s-nodes, $v_i^*, v_j^* \in V^*$, that are maximally label connected on the same label, i.e., $\lambda(v_i^*) = \lambda(v_j^*)$, and that have either the same set of incoming (*source-merge*) or outgoing (*target-merge*) edge labels, i.e., $\Lambda_d(v_i^*) = \Lambda_d(v_j^*)$, $d \in \{1, 2\}$ (see Alg. 2). Each *h-edge*, $\hat{e}$, merges all s-edges in E^* with the same label and orientation, i.e., $e_i^*.d = e_j^*.d$, for $d \in \{1, 2\}$.

Algorithm 2 MERGE(V^*, Λ, m)

Input: V^* – s-nodes; Λ – labels; m – heuristic mode; **Output:** $\hat{V}$ – h-nodes

1: **for all** $v^* \in V^*$ **do**
2: $\quad \Lambda_d(v^*) \leftarrow \{l \in \Lambda \mid \exists e^* = l(\_, \_) \in E^* \wedge e.d = v^*\}$ ▷*Labels Incoming/Outgoing* v^*
3: **for all** $v_1^*, v_2^* \in V^*$ **do** ▷*Pair-wise S-node Inspection*
4: $\quad b_\lambda \leftarrow \lambda(v_1^*) \stackrel{?}{=} \lambda(v_2^*)$, $b_d \leftarrow \Lambda_d(v_1^*) \stackrel{?}{=} \Lambda_d(v_2^*)$, $d \in \{1, 2\}$ ▷*Boolean Conditions*
5: $\quad$ **if** $m =$ `true` **then** $\hat{v} \leftarrow \{v_1^*, v_2^* \mid b_\lambda \wedge b_1 =$ `true`$\}$ ▷***Target-Merge***
6: $\quad$ **else** $\hat{v} \leftarrow \{v_1^*, v_2^* \mid b_\lambda \wedge b_2 =$ `true`$\}$ ▷***Source-Merge***
7: $\hat{V} \leftarrow \{\hat{v}_k \mid k \in [1, |V^*|]\}$ ▷*H-node Computation*
8: **return** $\hat{V}$

To each h-node, we attach properties, whose values, except $LPercent$, are the sum of those corresponding to each of its *s-nodes*. For the label percentage, these values record the weighted percentage mean. Next, we merge *s-edges* into *h-edges*, if they have the same label and endpoints, and attach to each h-edge, its number of compressed s-edges, $EWeight$. We also record the avg. s-node weight, $\mathcal{V}^*Weight$, to estimate how many nodes a h-node compresses.

To formally characterize the graph transformation corresponding to our summarization technique, we first define the following function.

Definition 2 (Valid Summarization). *For $\mathcal{G} = (\mathcal{V}, E)$, a* valid summarization function $\chi_\Lambda : \mathcal{V} \to \mathbb{N}$ *assigns vertex identifiers, s.t., any vertices with the same identifier are either in the same maximally l-connected $\mathcal{G}$-subgraph, or in different ones, not connected by an l-labeled edge.*

A *valid summary* is thus obtained from $\mathcal{G}$, by collapsing vertices with the same χ_Λ into h-nodes and edges with the same (depending on the heuristic, ingoing/outgoing) label into h-edges. We illustrate this below.

Example 5 (Graph Compression). The graphs in Fig. 4c are obtained from $\mathcal{G}^* = (V^*, E^*)$, after the **merge phase**. Each h-node contains the s-nodes (see Fig. 4b) collapsed via the target-merge (left) and source-merge (right) heuristics.

We study our summarization's *optimality*, i.e., the size of the obtained compressed graph, to grasp its tractability. Specifically, we investigate the following *MinSummary* problem, to establish whether one can always *minimize* the number of nodes of an input graph, when constructing its *valid* summary.

Problem 1 (Minimal Summary). Let *MinSummary* be the problem that, for a graph $\mathcal{G}$ and an integer $k' \geq 2$, decides if there exists a label-driven partitioning Φ of $\mathcal{G}$, $|\Phi| \leq k'$, such that χ_Λ is a *valid summarization*.

Each MinSummary h-node is thus intended to regroup as many nodes from the original graph as possible, while ensuring these are connected by frequently occurring labels. This condition (see Def. 2) reflects the central idea of our framework, namely that the connectivity of such prominent labels can serve to both compress a graph and to approximately evaluate label-constrained reachability queries. Next, we establish the difficulty of solving MinSummary.

Theorem 1 (MinSummary NP-completeness). *Even for undirected graphs,* $|\Lambda(\mathcal{G})| \leq 2$*, and* $k' = 2$*, MinSummary is NP-complete* [7].

The intractability of constructing an optimal summary thus justifies our search for heuristics with good performance in practice.

4 Approximate Query Evaluation

Query Translation. For $\mathcal{G}$ and a counting reachability query Q, we approximate $[\![Q]\!]_\mathcal{G}$, the evaluation of Q over $\mathcal{G}$. We translate Q into a query Q^T, evaluated over the summarization $\hat{\mathcal{G}}$ of $\mathcal{G}$, s.t $[\![Q^T]\!]_{\hat{\mathcal{G}}} \approx [\![Q]\!]_\mathcal{G}$. The translations by input query type are given in Fig. 6, with PGQL as concrete syntax. *(1) Simple and Optional Label Queries.* A label l occurs in $\hat{\mathcal{G}}$ either within a h-node or on a cross-edge. Thus, we either cumulate the number of l-labeled h-node inner-edges or the l-labeled cross-edge weights. To account for the potential absence of l, we also estimate, in the optional-label queries, the number of nodes in $\hat{\mathcal{G}}$, by cumulating those in each h-node. *(2) Kleene Plus and Kleene Star Queries.* To estimate l^+, we cumulate the counts within h-nodes containing l-labeled inner-edges and the weights on l-labeled cross-edges. For the former, we distinguish whether the l_+ reachability is due to: 1) inner-connectivity – we use the property counting the inner l-paths; 2) incoming cross-edges – we cumulate the l-labeled in-degrees of h-nodes; or 3) outgoing cross-edges – we cumulate the number of outgoing l-paths. To handle the ϵ-label in l^*, we also estimate the number of nodes in $\hat{\mathcal{G}}$. *(3) Disjunction.* We treat each possible configuration, on both labels. Hence, we either cumulate the number of h-node inner-edges or that of cross-edge weights, with either label. *(4) Binary Conjunction.* We distinguish whether the label pair appears on an inner h-node path, on a cross-edge path, or on a traversal one.

Example 6. We illustrate the approximate evaluation of these query types on Fig. 4. To evaluate the number of single-label atomic paths, e.g., $Q^T_L(l_5)$, as l_5 only occurs inside h-node $\hat{v}_2$, $[\![l_5]\!]_{\hat{\mathcal{G}}}$ is the amount of l_5-labeled inner edges in $\hat{v}_2$, i.e., $EWeight(\hat{v}_2, l_5) * LPercent(\hat{v}_2, l_5) = 7$. To estimate the number of optional label atomic paths, e.g., $Q^T_O(l_2)$, we add to $Q^T_L(l_2)$ the total number of graph vertices, $\sum_{\hat{v}\in\hat{\mathcal{V}}} \mathcal{V}^* Weight(\hat{v}) * \mathcal{V}Weight(\hat{v})$ (empty case). As l_2 only appears on a h-edge of weight 2 and there are 25 initial vertices, $[\![l_2?]\!]_{\hat{\mathcal{G}}}$ is 27. To

[7] Proof given at: http://web4.ensiie.fr/~stefania.dumbrava/SUM19_appx.pdf.

estimate Kleene-plus queries, e.g., $Q_P^T(l_0)$, as no h-edge has label l_0, we return $LReach(\hat{v}_1, l_0)$, i.e., the number of l_0-connected vertex pairs. Thus, $[\![l_0^+]\!]_{\hat{\mathcal{G}}}$ is 15. For Kleene-star, we add to this, the previously computed total number of vertices and obtain that $[\![l_0^*]\!]_{\hat{\mathcal{G}}}$ is 40. For disjunction queries, e.g., $[\![l_4 + l_1]\!]_{\hat{\mathcal{G}}}$, we cumulate the single-labeled atomic paths on each label, yielding 14. For binary conjunctions, e.g., $[\![l_4^- \cdot l_5]\!]_{\hat{\mathcal{G}}}$, we rely on the traversal connectivity, $LPart(v^*, l_4, l_5, 2, 2)$, as l_4 appears on a h-edge and, l_5, inside h-nodes; we thus count 7 node pairs.

$Q_L(l)$	`SELECT COUNT(*) MATCH () -[:l]-> ()`
$Q_L^T(l)$	`SELECT SUM(x.LPERCENT_L * x.EWEIGHT) MATCH (x)` `SELECT SUM(e.EWEIGHT) MATCH () -[e:l]-> ()`
$Q_O(l)$	`SELECT COUNT(*) MATCH () -[:l?]-> ()`

Q_O^T	`SELECT SUM(x.LPERCENT_L * x.EWEIGHT) MATCH (x)`
	`SELECT SUM(e.EWEIGHT) MATCH () -[e:l]-> ()`
	`SELECT SUM(x.AVG_SN_VWEIGHT * x.VWEIGHT) MATCH (x)`

$Q_P(l)$	`SELECT COUNT(*) MATCH () -/:l+/-> ()`
$Q_P^T(l)$	`SELECT SUM(x.LREACH_L) MATCH (x) WHERE x.LREACH_L > 0` `SELECT SUM(e.EWEIGHT) MATCH () -[e:l]-> ()`
$Q_S(l)$	`SELECT COUNT(*) MATCH () -/:l*/-> ()`

$Q_S^T(l)$	`SELECT SUM(x.LREACH_L) MATCH (x) WHERE x.LREACH_L > 0`
	`SELECT SUM(e.EWEIGHT) MATCH () -[e:l]-> ()`
	`SELECT SUM(x.AVG_SN_VWEIGHT * x.VWEIGHT) MATCH (x)`

$Q_D(l_1, l_2)$	`SELECT COUNT(*) MATCH () -[:l1|l2]-> ()`
$Q_D^T(l_1, l_2)$	`SELECT SUM(x.LPERCENT_L1 * x.EWEIGHT + x.LPERCENT_L2 * x.EWEIGHT) MATCH (x)` `SELECT SUM(e.EWEIGHT) MATCH () -[e:l1|l2]-> ()`
$Q_C(l_1, l_2, 1, 1)$	`SELECT COUNT(*) MATCH () -[:l1]-> () <-[:l2]- ()`
$Q_C(l_1, l_2, 1, 2)$	`SELECT COUNT(*) MATCH () -[:l1]-> () -[:l2]-> ()`
$Q_C(l_1, l_2, 2, 1)$	`SELECT COUNT(*) MATCH () <-[:l1]- () <-[:l2]- ()`
$Q_C(l_1, l_2, 2, 2)$	`SELECT COUNT(*) MATCH () <-[:l1]- () -[:l2]-> ()`

$Q_C^T(l_1, l_2, d_1, d_2)$	`SELECT SUM((x.LPART_L2_L1_D2_D1 * e.EWEIGHT)/(x.LPERCENT_L1 * x.VWEIGHT))`
	`MATCH (x) -[e:l2] -> () WHERE x.LPERCENT_L1 > 0`
	`SELECT SUM((y.LPART_L1_L2_D1_D2 * e.EWEIGHT)/(y.LPERCENT_L2 * y.VWEIGHT))`
	`MATCH () -[e:l1] -> (y) WHERE y.LPERCENT_L2 >0`
	`SELECT SUM(x.CREACH_L1_L2_D1_D2) MATCH (x)`
	`SELECT SUM(x.EWEIGHT * min(x.LPERCENT_L1, x.LPERCENT_L2)) MATCH (x)`

Fig. 6: Query translations onto the graph summary.

5 Experimental Analysis

In this section, we present an empirical evaluation of our graph summarization, recording (1) the succinctness of our summaries and the efficiency of the underlying algorithm and (2) the suitability of our summaries for approximate evaluation of counting label-constrained reachability queries.

Setup, Datasets and Implementation. The summarization and approximation modules are implemented in Java using OpenJDK 1.8[8]. As the underlying graph database backend, we have used Oracle Labs PGX 3.1, which is the only property graph engine allowing for the evaluation of complex RPQs.

To implement the intermediate graph analysis operations (e.g., weakly connected components), we used the Green-Marl domain-specific language and mod-

[8] Available at: https://github.com/grasp-algorithm/label-driven-summarization.

Dataset	$\|L_V\|$	$\|L_E\|$	$\sim 1K$		$\sim 5K$		$\sim 25K$		$\sim 50K$		$\sim 100K$		$\sim 200K$	
			$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
bib	5	4	916	1304	4565	6140	22780	3159	44658	60300	88879	119575	179356	240052
social	15	27	897	2127	4434	10896	22252	55760	44390	110665	88715	223376	177301	450087
uniprot	5	7	2170	3898	6837	18899	25800	97059	47874	192574	91600	386810	177799	773082
shop	24	82	3136	4318	6605	10811	17893	34052	31181	56443	57131	93780	109205	168934

Fig. 7: Datasets: no. of vertices $|V|$, edges $|E|$, vertex $|L_V|$ and edge labels $|L_E|$.

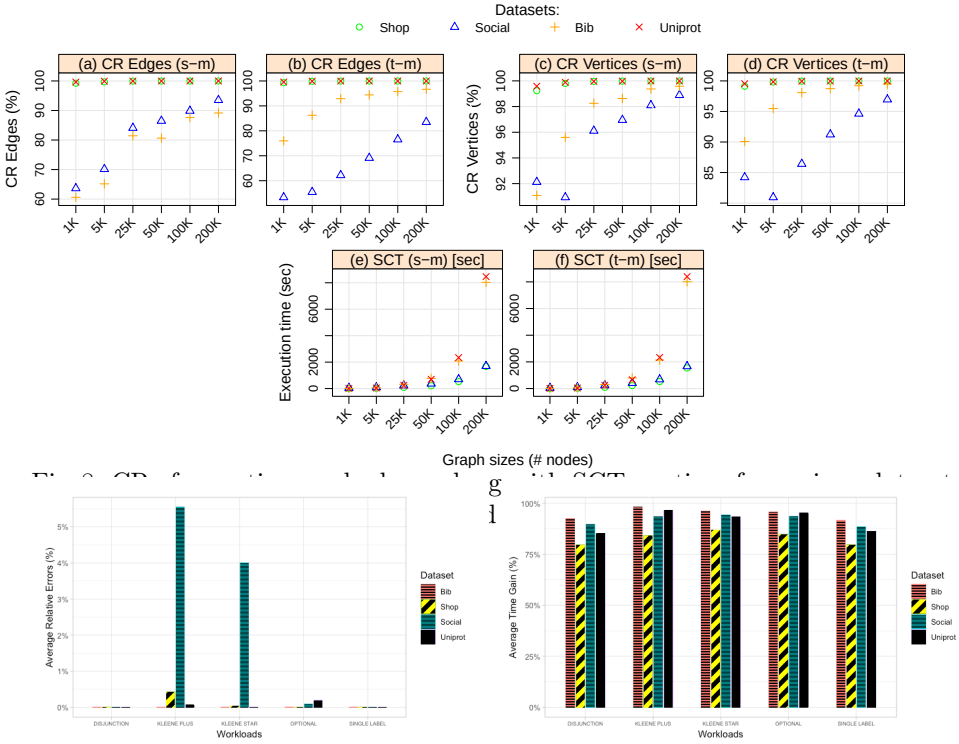
ified the methods to fit the construction of node properties required by our summarization algorithm. We base our analysis on the graph datasets in Fig. 7, encoding: a Bibliographic network (*bib*), the LDBC social network schema [12] (*social*), Uniprot knowledge graphs (*uniprot*), and the WatDiv schema [1] (*shop*).

We obtained these datasets using gMark [5], a synthetic graph instance and query workload generator. As gMark tries to construct the instance that best fits the size parameter and schema constraints, the resulting sizes vary (especially for the very dense graphs *social* and *shop*). Next, on the same datasets, we generated workloads of varying sizes, for each type in Section 2. These datasets and related query workloads have been chosen since they provide the most recent benchmarks for recursive graph queries and also to ensure a comparison with SumRDF [19] (as shown next) on a subset of those supported by the latter. Studies [8,17] have shown that practical graph pattern queries formulated by users in online query endpoints are often small: 56.5% of real-life SPARQL queries consist of a single edge (RDF triple), whereas 90.8% use 6 edges at most. Hence, we select small-sized template queries with frequently occuring topologies, such as chains [8], and formulate them on our datasets, for workloads of ~ 600 queries.

Experiments ran on a cloud VM with Intel Xeon E312xx, 4 cores, 1.80 GHz CPU, 128GB RAM, and Ubuntu 16.04.4 64-bit. Each data point corresponds to repeating an experiment 6 times, removing the first value from the average.

Summary Compression Ratios. First, we evaluate the effect that using the source-merge and target-merge heuristics has on the *summary construction time* (SCT). We also assess the *compression ratio* (CR) on the original graph's vertices and edges, by measuring $(1 - |\hat{\mathcal{V}}|/|\mathcal{V}|) * 100$ and, respectively, $(1 - |\hat{\mathcal{E}}|/|\mathcal{E}|) * 100$.

Next, we compare the results for source and target merge. In Fig. 8 (a-d), the most homogeneous datasets, *shop* and *uniprot*, achieve very high CR (close to 100%) and steadily maintain it with varying graph sizes. As far as heterogeneity significantly grows for *bib* and *social*, the CR becomes eagerly sensitive to the dataset size, starting with low values, for smaller graphs, and stabilizing between **85%** and **90%**, for larger ones. Notice also that *bib* and *social*, while similar, display a symmetric CR behavior: the former better compresses vertices, while the latter, edges. Concerning the SCT runtime in Fig. 8 (e-f), all datasets keep a reasonable performance for larger sizes, even the most heterogeneous one *shop*. The runtime is, in fact, not affected by heterogeneity, but is rather sensitive, for larger sizes, to $|E|$ variations (up to $450K$ and $773K$, for *uniprot* and *social*). Also, while the source and target merge SCT runtimes are similar, the latter achieves better CRs for *social*. Overall, the dataset with the worst CR for the two heuristics is *shop*, with the lowest CR for smaller sizes. This is also due to the



(a) Avg. Rel. Error/Workload (b) Avg. Time Gain/Workload

Fig. 9: Rel. Error (a), Time Gain (b) per Workload, per Dataset, 200K nodes.

high number of labels in the initial *shop* instances, and, hence, to the high number of properties its summary needs: on average, for all considered sizes, 62.33 properties, against 17.67, for *social* graph, 10.0, for *bib*, and 14.0, for *uniprot*. These experiments show that, despite its high complexity, our summarization provides high CRs and low SCT runtimes, even for large, heterogeneous graphs.

Approximate Evaluation Accuracy. We assess the *accuracy* and *efficiency* of our engine with the *relative error* and *time gain* measures, respectively. The relative error (per query Q_i) is $1 - min(Q_i(\mathcal{G}), Q_i^T(\hat{\mathcal{G}}))/\ max(Q_i(\mathcal{G}), Q_i^T(\hat{\mathcal{G}}))$ (in %), where $Q_i(\mathcal{G})$ computes (with PGX) the counting query Q_i, on the original graph, and $Q_i^T(\hat{\mathcal{G}})$ computes (with our engine) the translated query Q_i^T, on the summary. The time gain is: $t_{\mathcal{G}} - t_{\hat{\mathcal{G}}}/max(t_{\mathcal{G}}, t_{\hat{\mathcal{G}}})$ (in %), where $t_{\mathcal{G}}$ and $t_{\hat{\mathcal{G}}}$ are the query evaluation times of Q_i on the original graph and on the summary.

For the Disjunction, Kleene-plus, Kleene-star, Optional and Single Label query types, we have generated workloads of different sizes, bound by the number of labels in each dataset. For the concatenation workloads, we considered binary conjunctive queries (CQs) without disjunction, recursion, or optionality. Note that, currently, our summaries do not support compositionality.

ID	Query Body	Approx. Answer		Rel. Error (%)		Runtime (ms)	
		SumRDF	APP	SumRDF	APP	SumRDF	APP
Q_1	`(x0)-[:producer]->()<-[:paymentAccepted]-(x1)`	75	76	1.32	0.00	136.30	38.2
Q_2	`(x0)-[:totalVotes]->()<-[:price]-(x1)`	42.4	44	3.64	0.00	50.99	17
Q_3	`(x0)-[:jobTitle]->()<-[:keywords]-(x1)`	226.7	221	2.51	0.18	463.85	12.8
Q_4	`(x0)<-[:title]-()-[:performedIn]->(x1)`	19.5	20	2.50	0.00	831.72	8.8
Q_5	`(x0)-[:artist]->()<-[:employee]-(x1)`	143.3	133	7.19	0.37	196.77	10.6
Q_6	`(x0)-[:follows]->()<-[:editor]-(x1)`	524	528	0.38	0.48	1295.83	19

Fig. 10: Performance Comparison: SumRDF vs. APP (our approach): approx. eval. of binary CQs, `SELECT COUNT(*) MATCH` Q_i, on the summaries of a shop graph instance (31K nodes, 56K edges); comparing estimated *cardinality* (no. of computed answers), *rel. error* w.r.t the original graph results, and *query runtime*.

Fig.9 (a) and (b) show the relative error and average time gain for the Disjunction, Kleene-plus, Kleene-star, Optional and Single Label workloads. In Fig. 9 (a), we note that the avg. relative error is kept low in all cases and is bound by 5.5%, for the Kleene-plus and Kleene-star workloads of the *social* dataset. In all the other cases, including the Kleene-plus and Kleene-star workloads of the *shop* dataset, the error is relatively small (near 0%). This confirms the effectiveness of our graph summaries for approximate evaluation of graph queries. In Fig. 9 (b), we studied the efficiency of approximate evaluation on our summaries by reporting the time gain (in %) compared with the query evaluation on the original graphs for the four datasets. We notice a positive time gain ($\geq$ **75%**) in most cases, but for disjunction. While the relative approximation error is still advantageous for disjunction, disjunctive queries are time-consuming for approximate evaluation on our summaries, especially for extremely heterogeneous datasets, such as shop (having the most labels). This is due to the overhead introduced by considering all possible connectivity combinations on the disjunctive labels. The problem of scaling our method, without prohibitive accuracy loss, to queries involving multiple labels and further compositionality, e.g., Kleene-star over disjunctions [22], is challenging and falls under the scope of future work.

Baseline for approximate query evaluation performance. The closest system to ours is SumRDF [19] (see Section 6), which, however, operates on a *simpler edge-labeled model rather than on property graphs and is tailored for estimating the results of conjunctive queries only.* As a performance baseline, we considered the shop dataset in gMark [5], simulating the WatDiv benchmark [1] (also a benchmark in [19]). From this dataset with 31K nodes and 56K edges, we generated the corresponding SumRDF and our summaries. We obtained a better CR than SumRDF, with **2737** nodes vs. **3480** resources and **17430** edges vs. **29621** triples. This comparison is, however, tentative, as our approach compresses vertices independently of the edges, while SumRDF returns triples. We then considered the same CQ types as in Fig. 10. Comparing our approach vs. SumRDF (see Fig. 10), we recorded an *average relative error* of estimation of only **0.15%.** vs. **2.5%** and an *average query runtime* of only **27.55 ms** vs. **427.53 ms**. As SumRDF does not support disjunctions, Kleene-star/plus queries and optional queries, further comparisons were not possible.

6 Related Work

Preliminary work on approximate graph analytics in a distributed setting has recently been pursued in [15]. They rather focus on a graph sparsification technique and small samples, in order to approximate the results of specific graph algorithms, such as PageRank and triangle counting on undirected graphs. In contrast, our approach operates in a centralized setting and relies on query-driven graph summarization for graph navigational queries with aggregates.

RDF graph summarization for cardinality estimation has been tackled in [19], albeit for a less expressive data model than ours (plain RDF vs. property graphs). They focus on Basic Graph Patterns (BGP), hence their considered query fragment has limited overlap with ours. As shown in Section 5, our approximate evaluation is faster and more accurate on a common set of (non recursive) queries.

An algorithm for answering graph reachability queries, using graph simulation based pattern matching, is given in [13], to construct query preserving summaries. However, it does not consider property graphs or aggregates.

Aggregation-based graph summarization [16] is at the heart of previous approaches, the most notable of which is SNAP [20]. This method is mainly devoted to discovery-driven graph summarization of heterogeneous networks and is unsuitable for approximate query evaluation.

More recently, Rudolf et al. [18] have introduced a graph summary suitable for property graphs based on a set of input summarization rules. However, it does not support the label-constrained reachability queries in this paper. Graph summaries for answering subgraphs returned by keyword queries on large networks are studied in [24]. Our query classes significantly differ from theirs.

7 Conclusion

Our paper focuses on a novel graph summarization method that is suitable for property graph querying. As the underlying MinSummary decision problem is NP-complete, this technique builds on an heuristic that compresses label frequency information in the nodes of the graph summary. We show the practical effectiveness of our approach, in terms of compression ratios, error rates and query evaluation time. As future work, we plan to investigate the feasibility of our graph summary for other query classes, such as those described in [22]. Also, we aim to apply formal methods, as described in [6], to ascertain the correctness of our approximation algorithm, with provably tight error bounds.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: ISWC. pp. 197–212 (2014)
2. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A core for future graph query languages. In: SIGMOD. pp. 1421–1432 (2018)

3. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017)
4. Arenas, M., Conca, S., Pérez, J.: Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In: WWW. pp. 629–638 (2012)
5. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gmark: Schema-driven generation of graphs and queries. IEEE Trans. Knowl. Data Eng. **29**(4), 856–869 (2017)
6. Bonifati, A., Dumbrava, S., Arias, E.J.G.: Certified graph view maintenance with Regular Datalog. TPLP **18**(3-4), 372–389 (2018)
7. Bonifati, A., Fletcher, G., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2018)
8. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. PVLDB **11**(2), 149–161 (2017)
9. Bonifati, A., Martens, W., Timm, T.: Navigating the maze of Wikidata query logs. In: WWW. pp. 127–138 (2019)
10. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries. J. Comput. Syst. Sci. **64**(3), 443–465 (2002)
11. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. In: SIGMOD. pp. 323–330 (1987)
12. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., Boncz, P.A.: The LDBC Social Network Benchmark: Interactive Workload. In: SIGMOD. pp. 619–630 (2015)
13. Fan, W., Li, J., Wang, X., Wu, Y.: Query preserving graph compression. In: SIGMOD. pp. 157–168 (2012)
14. Hernández, D., Hogan, A., Riveros, C., Rojas, C., Zerega, E.: Querying wikidata: Comparing sparql, relational and graph databases. In: ISWC. pp. 88–103 (2016)
15. Iyer, A.P., Panda, A., Venkataraman, S., Chowdhury, M., Akella, A., Shenker, S., Stoica, I.: Bridging the GAP: towards approximate graph analytics. In: GRADES. pp. 10:1–10:5 (2018)
16. Khan, A., Bhowmick, S.S., Bonchi, F.: Summarizing static and dynamic big graphs. PVLDB **10**(12), 1981–1984 (2017)
17. Malyshev, S., Krotzsch, M., Gonzalez, L., Gonsior, J., Bielefeldt, A.: Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In: ISWC (2018)
18. Rudolf, M., Voigt, H., Bornhövd, C., Lehner, W.: Synopsys: Foundations for multidimensional graph analytics. In: BIRTE. pp. 159–166 (2014)
19. Stefanoni, G., Motik, B., Kostylev, E.V.: Estimating the cardinality of conjunctive queries over rdf data using graph summarisation. In: WWW. pp. 1043–1052 (2018)
20. Tian, Y., Hankins, R.A., Patel, J.M.: Efficient aggregation for graph summarization. In: SIGMOD. pp. 567–580. ACM (2008)
21. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)
22. Valstar, L.D.J., Fletcher, G.H.L., Yoshida, Y.: Landmark indexing for evaluation of label-constrained reachability queries. In: SIGMOD. pp. 345–358 (2017)
23. Wood, P.T.: Query languages for graph databases. SIGMOD Rec. **41**(1), 50–60 (2012)
24. Wu, Y., Yang, S., Srivatsa, M., Iyengar, A., Yan, X.: Summarizing answer graphs induced by keyword queries. PVLDB **6**(14), 1774–1785 (2013)