Approximate Query Processing for Label-Constrained Reachability Queries

Internship report submitted to the faculty Claude Bernard, Lyon 1 in partial fulfillment of the requirements for the degree of

Master of Information Technology and Web

by

Amaia Nazábal

Research carried out in :

Laboratoire d'InfoRmatique en Images et Systèmes d'information, CNRS UMR 5205

Supervisors:
Prof. Dr. Angela Bonifati
Dr. Stefania Dumbrava

Keywords: property graphs, summarization, approximative query processing, regular path queries, navigational queries.

To my parents and my boyfriend Diego for providing me with unfailing support and continuous encouragement throughout my years of study...

Data is the new science.
BigData holds the answers.
Are you asking the right questions?

— Patrick P. Gelsinge, 2012

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor Prof. Dr. Angela Bonifati for her sage advice, insightful criticisms, and patient encouragement during the writing of this thesis.

The door of Dr. Stefania Dumbrava's office was always open, thanks for her help, support, and patience during the development of this work.

I would like to thank Oracle Labs for the anticipated release of the version 3.1 of Oracle Labs PGX, that allowed us to improve the results during the experimentation phase.

The last, but not the least, to my country, Paraguay. I am grateful for the funding sources that allowed me to pursue my master studies.

ABSTRACT

The current surge of interest in graph-based data models and graph query processing reflects the usage of increasingly complex reachability queries, as witnessed by analytical studies on real-world graph query logs. Despite the maturity of graph DBMS capabilities, complex *label-constrained reachability queries* remain difficult to evaluate along with their corresponding aggregate versions. In this work, we focus on an approximate evaluation of *counting* label-constrained reachability queries and we offer a human-explainable solution of graph Approximate Query Processing (AQP) by designing a graph summarization algorithm (GRASP) outputting a graph summary. Although the problem of node group minimization associated with the creation of GRASP summaries is NP-complete, our GRASP summaries are reasonably small in practice even for large graph instances and guarantee approximate graph query answering paired with controllable error estimates. We experimentally gauge the scalability and efficiency of our GRASP algorithm and verify the accuracy and error estimation of the graph AQP module. To the best of our knowledge, ours is the first system capable of approximate graph analytics for complex label-constrained reachability queries.

RÉSUMÉ

L'intérêt actuel pour les bases de données de graphes et le traitement des requêtes de graphes reflète que les requêtes complexes de type d'accessibilité (*reachability*) deviennent progressivement plus utilisées, comme l'atteste les études analytiques de requêtes de graphes réels. Malgré la maturité des capacités de SGBD des graphes, les requêtes d'accessibilité complexe avec des restrictions des étiquettes restent difficiles à évaluer avec leurs versions agrégées correspondantes. Dans ce travail, nous nous concentrons sur une évaluation approximative du *comptage* des requêtes d'accessibilité avec des restrictions des étiquettes et nous proposons une solution compréhensible d'AQP en concevant un algorithme de *summarization* des graphes (GRASP) produisant un graphe résumé. Bien que le problème de la minimisation des groupes de noeuds associé à la création de résumés GRASP soit NP-complet, nos résumés sont relativement petits dans la pratique même pour les grandes instances de graphes et garantissent une réponse approximative aux requêtes associées à des estimations d'erreur contrôlables. Nous évaluons expérimentalement l'évolutivité et l'efficacité de notre algorithme GRASP et vérifions la précision et l'estimation des erreurs du module AQP. À notre connaissance, notre système est le premier système capable d'analyser approximativement les graphes pour des requêtes complexes d'accessibilité avec des restrictions des étiquettes.

THE LABORATORY

LIRIS(Laboratoire d'InfoRmatique en Image et Systèmes d'information) is a mixed unit of research. It has 330 members, and its main scientific field is computing science and information technology.

Some of its activities of research are at the cross-road between engineering, human and social sciences, life sciences, and environmental sciences. Six of the centers participating in these lines of research.

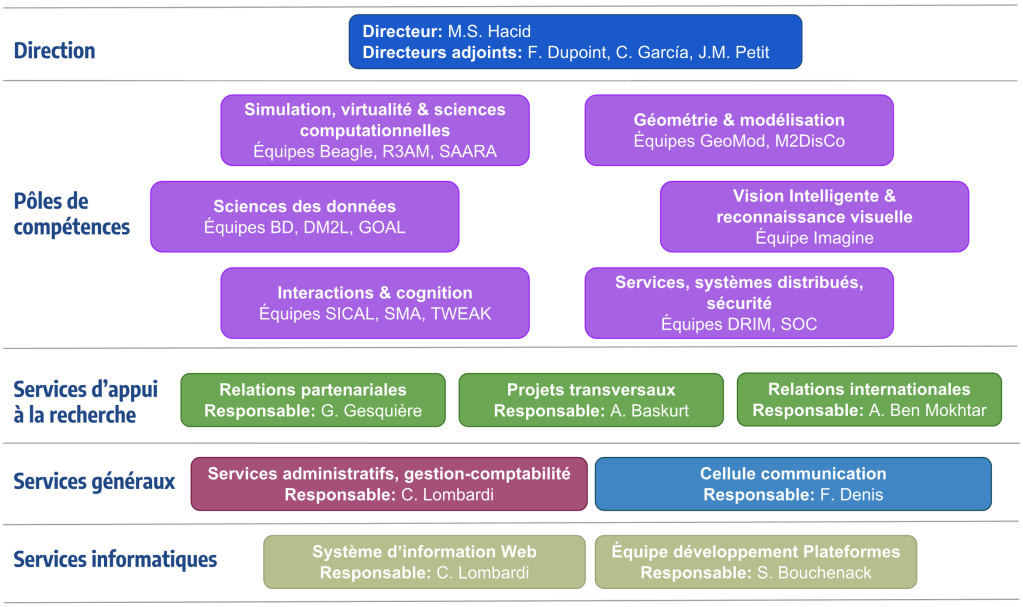


Figure 0.2: Organization chart

LIRIS covers multiples scientific themes structured in six centers including 14 teams:

- Intelligent vision and visual recognition (Imagine team).
- Geometry and Modeling (GeoMod and M2DisCo teams)
- Data Science (DB, DM2L and GOAL teams)
- Interactions and cognition (SICAL, SMA, and TWEAK teams)
- Services, Distributed Systems and Security (DRIM and SOC teams)
- Simulation, virtuality and computational sciences (Beagle, R3AM and SAARA teams)

My internship was carried out in the DB team. The group is focused on the principles, techniques, and applications of data management systems and it is divided around three themes: *Models and Languages*, *Architecture of systems* and *Security and data quality*.

THE MISSION

The goal of this internship is to develop *an efficient approximate query processing algorithms* for graph databases. In the settings we consider:

(i) data is stored as property graphs;

(ii) users submit queries that request aggregate values for navigational paths;

(iii) the data volume makes it prohibitive to retrieve all of them at query time.

CONTRIBUTIONS

(a) A Query-oriented Graph Summarization technique (GRASP Algorithm).

(b) An AQP implementation that allows exploiting the pre-computed properties in the GRASP Algorithm to approximate the answers of the original graph.

(c) A query translator from query workloads on the original graph to the query workloads on the summary graph.

(d) Small Error Bounds and Experimental Analysis.

(e) A prototype.

(f) A regular research paper submitted(in process of reviewing) in *The Forty-fifth International Conference on Very Large Data Bases - Los Angeles, California* conference.

CONTENTS

INTRODUCTION

During the last decade, we have seen a resurgence interest in graph databases. Part of this resurgence is due to the need for simplified representations which benefit from graph pattern and navigational queries between nodes with an arbitrary path length. Representing entities and relationships as nodes and edges allow us to take advantage of these benefits.

It is also true that graph processing has become an integral part of big-data analytics. Of course, this tremendous amount of information can be analyzed by leveraging the already mature query capabilities of graph DBMSs. However, complex graph reachability queries, entailing rather intricate and possibly recursive graph patterns (as required by extracting friendship relationships in social networks or protein-to-protein interactions in biological networks), prove difficult to evaluate on even small-sized graph datasets. Moreover, the usage of these queries has radically increased in real-world graph query logs.

The primary objective of this research is to propose an approach to reduce the time complexity of such queries, using techniques for obtaining small summaries, preserving topological properties of the original graph, and allowing for improved query execution times.

In this work, we introduce the GRASP algorithm, a query-oriented, summarization algorithm aiming at preserving information about the label-constrained reachability of the initial graph and bookkeeping additional statistics in the node properties of the graph summary to allow approximate graph query evaluation. Our goal is also to offer a human-explainable approximate graph query processing by focusing on regular path queries that identify paths in graphs through regular expressions over the edge labels. We focus on counting reachability queries that are label-constrained.

The rest of this thesis is organized in ten chapters, where the introduction is general and the others are divided into two parts:

- *First part: Theoretical fundamentals.* Chapters 1 to 4, where we review the concepts of property graphs, graph patterns, summarization and graph databases.
- *Second part: Our Approach.* Sections 5 to 9, where we introduce our GRASP algorithm, the experimental analysis, the related work, future perspectives and conclusion.

Part I

THEORETICAL FUNDAMENTALS

1

PROPERTY GRAPHS

Graphs are used to encode data, whereby nodes represent objects in a domain of interest, and edges represent relationships between these objects [4]. However, different graph data models exist, namely, plain graphs, labeled graphs and property graphs. In this thesis, we will focus only on property graphs.

In property graphs, edges and nodes can be labeled. Each edge and node is enriched with a unique identifier that can be used as an index to associate additional meta-information[1] directly to that edge or node. The graph property instances are multi-edge digraphs, in which objects are represented by typed data vertices, and the relationships between these, as typed, labeled edges.

Both vertices and edges can have any number of properties (meta-information) associated to them; we denote the set of all properties as $\mathcal{P}$.

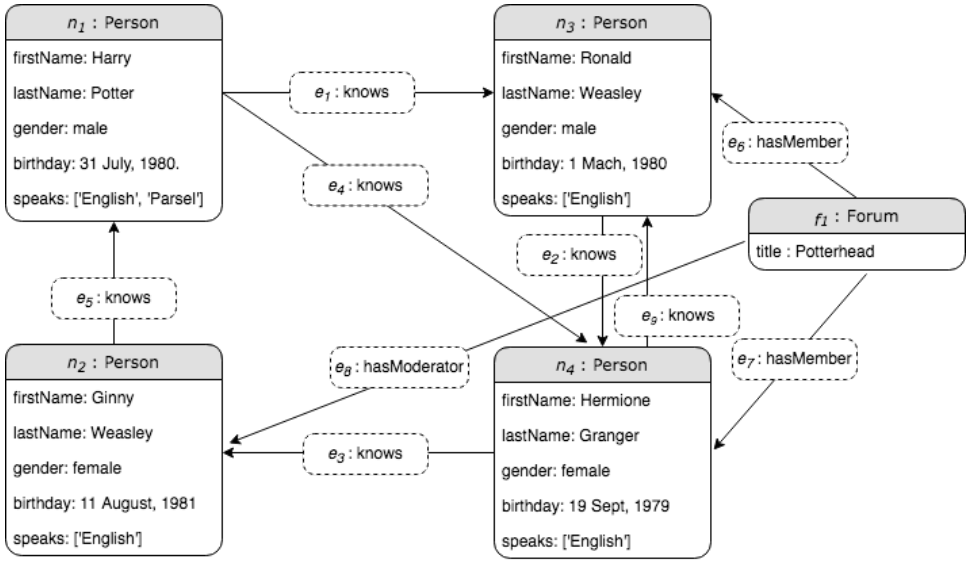


Figure 1.1: A property graph inspired in LDBC Social-Network schema with attribute values storing information of people who know each other.

Example 1 *A property graph representation of a fictitious social network is shown in Figure 1.1. Each node is labeled Person or Forum and each edge is labeled knows, hasMember or hasModerator. Nodes with label Person may have attributes for* `firstName`, `lastName`, `gender`, `birthday` *and* `speaks`, *nodes with Forum label may have an attribute* `title`; *and in this example, the edges do not have any attribute, only a single label each one.*

For instance, in Figure 1.1, we show a graph inspired in LDBC Social-Network schema, representing a subset of characters that includes labels and attributes on nodes and edges. In this figure, the identifiers are shown in italic and the attributes are shown inside the round rectangle box, as property-value pairs. Thus, for example, for the node with the identifier n_1 the first two properties are `firstName` and `lastName`, their corresponding values are `Harry` and `Potter`, and its label is *Person*. In this example, the edges do not contain any property, only labels.

In this model, we can directly encode multiple edges (having different identifiers) with the same label between the same two nodes. This condition is called: *multi-labels*. In our definition of a property graph, each node, and edge can be associated with a single label; however, a variation of property graphs allows the association of multiple labels, which we call *multi-valued property graph*.[4]

1 The meta-information is represented in the form of a set of property-value pairs called attributes.

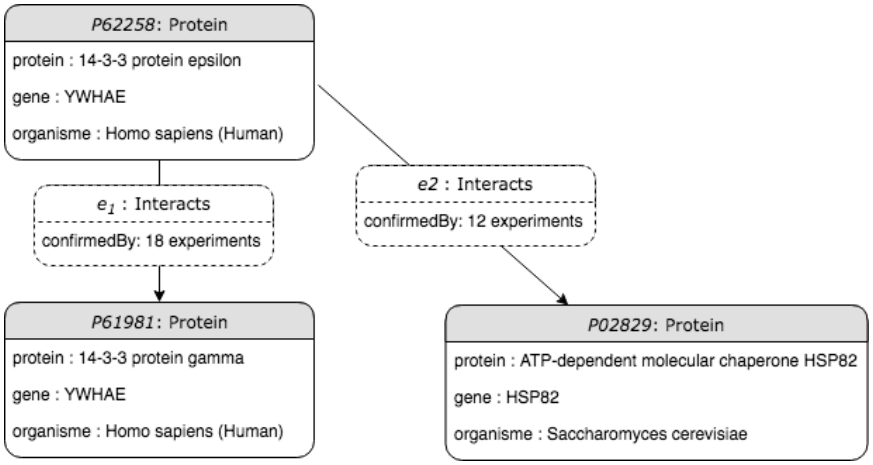


Figure 1.2: A property graph inspired by UniProt schema, storing information about protein-to-protein interactions in a biological network.

Property graphs can either be homogeneous or heterogeneous [26]. A homogeneous property graph is a graph where every vertex/edge has the same properties with the same names and types as all the other vertices/edges (c.f. Figure 1.2). An heterogeneous property graph is a graph where the vertices/edges may have different number of properties and different types (c.f. Figure 1.1).

We now provide a formal definition of the notion of a property graph, using the example in the Figure 1.1.

Definition 1 (Property graph) A property graph $\mathcal{G}$ is a tuple $(V, E, \rho, \gamma, \sigma)$, where [4][25]:

(a) V is a finite set of vertices, such that $V = \{v_1, v_2, ..., v_n\}$. In our example $V = \{n_1, n_2, n_3, n_4, f_1\}$.

(b) E is a finite set of edges $E = \{e_1, e_2, ..., e_m\}$ such that V and E are disjoint. In our example $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$.

(c) $\rho : E \to (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ indicates that e is a directed edge from node v_1 to node v_2 in $\mathcal{G}$. For example: $\rho(e_1) = (n_1, n_3)$.

(d) $\gamma : (V \cup E) \to Lab$ is a total function where Lab is a set of labels. And for each $v \in V$ (resp. $e \in E$), $\gamma(v) = l_v$ (resp. $\gamma(e) = l_e$), where l_v (resp. l_e) is the label of the node v (resp. edge e) in $\mathcal{G}$. For example: $\gamma(n_1) = Person$.

(e) $\sigma : (V \cup E) \times \mathcal{P} \to \mathcal{V}$ is a partial function with $\mathcal{V}$ a set of values of $\mathcal{P}$. And for each $v \in V$ (resp. $e \in E$), $p \in \mathcal{P}$, there exists $\sigma(v, p) = s$ (resp. $\sigma(e, p) = s$), s.t. $s \in \mathcal{V}$ and it is the value of the property p for the node v (resp., edge e) in $\mathcal{G}$. For example: $\sigma(n_1, \texttt{firstName}) = \texttt{Harry}$.

(f) γ_v and γ_e are disjoints vertex and edge types, whose elements we denote as L_V, resp., L_E, such that $Lab = L_V \cup L_E$, and $E \subseteq V \times L_E \times V$.

(g) Each vertex $v \in V$ consist of a label id, l_v of type L_V, and a set of properties (attributes) $\{l_1, ..., l_k\}$, each of which is included in $\mathcal{P}$ and has a certain, potentially, undefined respective term value $\{t_1, ..., t_k\}$, all of them included in $\mathcal{V}$.

2

GRAPH PATTERNS

Multiple declarative query languages have emerged in the last decade, for querying instances of graph data models presented in the previous section. The main operations for interrogating graph are *graph pattern matching* and *graph navigation*.

2.1 GRAPH PATTERN MATCHING

Renzo et al. in [4] have defined a basic graph pattern (BGP) as a graph-structured query to be matched against the graph database. A BGP follows the same structure as the latter and it is used for querying while allowing variables. In other words, a BGP for querying a graph is just another graph where variables can appear as nodes or edge labels. A BGP mapping assigns to each pattern variable a constant in the original database.

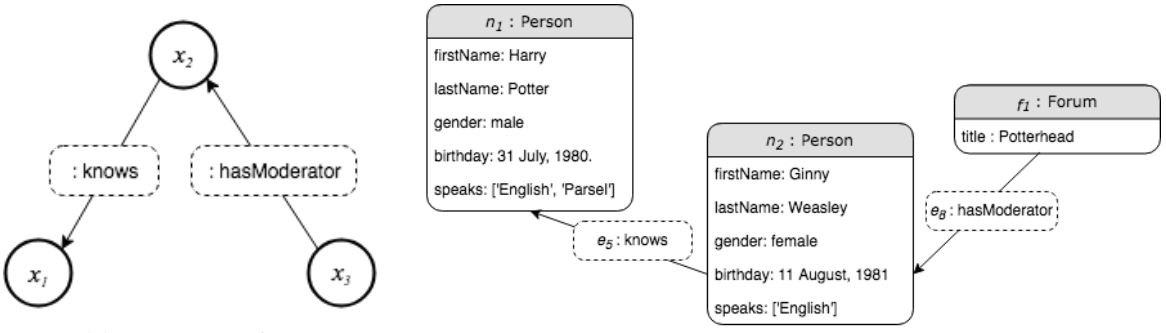
Example 2 *Let $\mathcal{G}$ be the graph in Figure 1.1. Assume we want to find the friends of all the moderators in this graph. We can do this by matching the BGP in Figure 2.1a, called Q, against $\mathcal{G}$. In Q, we use terms x_i as variables that will match any term in the database. On the other hand, knows and hasModerator are constants from the set Lab that will be the only matched edges with the corresponding labels in the original graph. The result of evaluating the BGP Q against the graph $\mathcal{G}$, which we denote as $Q(\mathcal{G})$, is shown in the Figure 2.1b.*

The evaluation of Q is made by replacing the variable x_1 by the vertex n_1, x_2 by the vertex n_2 and x_3 by the vertex f_1, we get then a subgraph (c.f. Figure 2.1b) of the original graph; thus we call this mapping a match for Q against $\mathcal{G}$. The result of Q consist of all such valid matches. The example 2 is a basic one; however, it is also possible to include conditions on the value of properties (e.g. x_1.`gender` = `'male'`) or equalities between attributes (e.g. x_1.`speaks` = x_2.`speaks`) or any boolean expression that it will filter the results. Evaluating a BGP Q against a graph $\mathcal{G}$ corresponds to listing all possible matches of Q in $\mathcal{G}$. More formally, we can define a match as follows.

Definition 2 (Match) Given a graph $\mathcal{G} = (V, E)$ and a BGP $Q = (V', E')$, a match h of Q in $\mathcal{G}$ is a mapping from $Const \cup Var$ to $Const$, where $Const$ is a set of constants and Var is a set of variables, such that:

(a) For each constant $a \in Const$, it is the case that $h(a) = a$; that is, the mapping maps constants to themselves and;

(b) For each edge $(src, label, dst) \in E'$, it holds that $(h(src), h(label), h(dst)) \in E$; this conditions imposes that each edge of Q is mapped to an edge of $\mathcal{G}$, and the structure of Q is preserved in its image under h in $\mathcal{G}$, or in other words, when applied to all the terms in Q, the result is a subgraph of $\mathcal{G}$.



(a) Query graph Q.

(b) The matching subgraph of $\mathcal{G}$, after evaluating Q.

Figure 2.1: An example of a BGP, a query graph Q and the affected sub-graph of $\mathcal{G}$. Both figure apply to the graph database depicted in Figure 1.1.

When evaluating a query Q over a graph $\mathcal{G}$, one can consider different semantics. In this work, we focus on the following ones: [4]

(a) *Homomorphism-based semantics.* In technical terms, a match h corresponds to a homomorphism from Q to $\mathcal{G}$. Thus, multiple variables in Q can map to the same term in $\mathcal{G}$. This is the unconstrained semantics, as no additional restriction is imposed.

(b) *Isomorphism-based semantics.* In some cases, it is desirable that different variables map to distinct terms. Under this type of semantics, certain types of variables are restricted to match distinct constants in the database. This leaves us with a variety of different possible isomorphism-based semantics, the main ones being:

   i *No-repeated-anything semantics.* This means that no two variables can be mapped to the same term in a given match.

   ii *No-repeated-node semantics.* As the name implies, the restriction is imposed only on the nodes.

   iii *No-repeated-edge semantics.* The restriction only applies to variables that map to edges (i.e. edge variables), whereas other types of variables (nodes, labels, properties, etc.) can still be mapped to the same term.

While some of the previous semantics may restrict the duplication of terms within a single match (isomorphism-based semantics) we can also consider another orthogonal preference of semantics with respect to duplicates matches in the result of evaluating a BGP Q over a graph database G, as follows: [4]

– *Set semantics.* $Q(\mathcal{G})$ is defined as a set of matches; in other words, the result of evaluating Q cannot contain duplicate matches.

– *Bag semantics.* $Q(\mathcal{G})$ is defined as a bag of matches, meaning that the number of times that a match appears in the result corresponds to the number of mappings that appear in the match.

Examples of these different semantics will be provided in Section 4.

2.2 GRAPH NAVIGATION QUERIES & RPQ

There also exists another type of queries to describe paths. One example of such query is to find all acquaintances-of-an-acquaintance of a given person in a social setting, as described in Figure 1.1. Here we are not only interested in immediate acquaintances of a person, but also in the people that person might know through other people. Queries such as the above are called *path queries* since they require us to navigate through the graph using paths of arbitrary length. Of course, sometimes the paths alone are not enough, and we are interested in repetitions of graph patterns inside the graph, to discover *pathways* or *patterns* that are often repeated. Such queries are called *navigational queries*.

Paths are the most basic navigational object in a graph database. The most fundamental type of path query is determining *path existence*. This is a fundamental notion related to the problem of reachability and transitive closure in directed graphs. Path existence refers to establishing whether there exists a directed path between a pair of nodes in a property graph. [4]

Definition 3 (Path Query) We denote a path query as $\pi = x \xrightarrow{\alpha} y$, where α specifies conditions on the paths, we wish to retrieve and x and y denote the endpoints of the path. The length of $|\pi|$ is the number of edges in π. A π path is called *nonempty* if $|\pi| > 0$. A path π is *simple* if it does not go through the same node twice.[8] There are a variety of formalisms under which α can express more complex *path constraints*. One of the prominent such formalisms is regular expressions defined over the set *Lab* of edge labels. For example, π can be also defined as the corresponding path $v_1 \xrightarrow{l_e^1} v_2 \ldots v_{k-1} \xrightarrow{l_e^k} v_k$, where the edge label concatenation are $l_e^1 \cdot \ldots \cdot l_e^k$.

A *regular expression* can be used as a path constraint, in order to retrieve all the paths that match the expression. Path queries using regular expressions are commonly known as Regular Path Queries (RQP).

Let us review first the notion of regular expressions and RPQ.

Definition 4 (Regular Expression) A regular expression over a set of labels L is an expression of the following form [25]:

$$E ::= L|(E \cdot E)|(E + E)|E^*|E^+|E?$$

A regular expression E matches a word $w = l_1...l_n$ if any of the following conditions is satisfied:

- $E \in L$ is a label and $w = E$.
- $E = (E_1 \cdot E_2)$ and there is $i \in 0, ..., n$ such that E_1 matches $l_1, ..., l_i$ and E_2 matches $l_{i+1}, ..., l_n$.
- $E = (E_1 + E_2)$ and w_i s matched by E_1 or by E_2.
- $E = E_1^*$ and w has the form $w_1w_2...w_n$ for $n \geq 0$, where each word w_i is matches by E_1.
- $E = E_1^+$ and w has the form $w_1w_2...w_n$ for $n > 0$, where each word w_i is matches by E_1.
- $E = E_1?$ and w is matched by E_1 or not.

Definition 5 (Regular Path Queries) A Regular Path Query (RPQ) is an expression of the form $E(s, t)$ where E is a regular expression and s and t are terms (constants or variables). Let a and b be constants. An RPQ $E(a, b)$ is entailed by a graph $\mathcal{G}$ if there exists a directed path π from node a to node b that is labeled with a word matched by E [25].

Example 3 *Let $\mathcal{G}$ be the graph over $\sigma = knows, hasMember, hasModerator$ in Figure 1.1, where σ is a finite alphabet. The following RPQ matches the **acquaintance-of-an-acquaintance relationship** in our social network configuration and can be expressed via the following RPQ:*

$$Q_1 : Ans(x, y) \leftarrow (x, knows^+, y)$$

Here the symbol '+' denotes one or more relations. The regular expression $knows^+$ is used to specify all paths formed from a sequence of one or more edges with the label knows (c.f. Figure 2.2).

RPQs can be used to express typical reachability queries. However, there are various navigational operations not supported that seem quite natural. Thus, various extensions have been introduced to express these expressions and more intricate ones. Among these extensions, we note that allowing for inverse navigation, which results in *Two-Way Regular Path Queries* (2RPQs). Another notable extension is that allowing for conjunctive queries, called *Conjunctive Regular Path Queries* (CRPQs) that extends the power of RPQs with conjunctive queries.

Figure 2.2: A finite automaton representing the query Q_1

Definition 6 (Two-Way Regular Path Queries) The Two-Way Regular Path Queries (2RPQ) extend the vocabulary of RPQs by the inverse of each relationship symbol. For each symbol a in Lab there exists a^- which specify the traversal of edges in a backward direction.

Example 4 *The following 2RPQ match all the acquaintances of a moderator of a given forum in our social network graph in Figure 1.1, and can be expressed as follows:*

$$Q_2 : Ans(x, y) \leftarrow (x, knows \cdot hasModerator^-, y)$$

To understand the example, we have to take into account that the domain of the predicate hasModerator is a vertex of label Forum and the range a vertex of a label Person. Note; however, that the domain and range of the predicate knows is the same, a vertex of label Person. That is the reason why we need to use the inverse of the predicate hasModerator, to maintain the typing consistency between both vertex labels.

Definition 7 (Conjunctive Two-Way Regular Path Queries) The Conjunctive Two-Way Regular Path Queries (C2RPQs) is another extension from RPQs. They include 2RPQ and also allow free path variables in the query. [8] [25]

Example 5 *The following C2RPQ matches all the pairs consisting of a person and the moderator of a forum, of which an acquaintance of hers is a member (see graph $\mathcal{G}$ in Figure 1.1). This can be expressed as follows:*

$$Q_3 : Ans(x,y) \leftarrow (x, knows, y) \wedge (y, hasMember^{-}, z) \wedge (z, hasModerator, y)$$

It is easy to see that this query cannot be expressed using only 2RPQ because we are using the same variable y in two conditions, i.e., the person must be a member and a moderator of the same forum.

BGPs can also be extended to *Navigational Graph Patterns*(NGP), in which edges can also be labeled by an RPQ denoting an arbitrary path. While other further additions to BGPs exist, in this work, we will focus the previous definitions.

In RPQs, the possibility of having paths involving cycles or other topologies gives rise to more semantics for the evaluation of path queries. The most common forms of evaluation are the no-repeated-node and no-repeated-edge semantics. Additionally, we can find these approaches:[4]

(a) *Arbitrary path semantics.* All paths in $\mathcal{G}$ that satisfy the constraints of α are included in the result. Under this semantics, the result of a query with a transitive cloture operation (* or +) may contain an infinite number of paths (for example if there exist cycles inside the topology of the graph) which quickly becomes impractical.

(b) *Shortest path semantics.* In this case, the result of a query is defined only in terms of the *shortest paths*, i.e., paths of minimal length that satisfy the constraint specify in of P. A variation of this approach is also the *cheapest path semantics*, which keeps the same principle with the only difference that instead of looking for the shortest path, it looks for the cheapest one.

3

SUMMARIZATION

As we have seen in the previous sections, the properties graphs are increasingly bigger and, thus, data analysis using RPQs becomes expensive. Despite the maturity of graph DBMS capabilities, complex label-constrained reachability queries remain difficult to evaluate along with their corresponding aggregate versions. These large amounts of data and the need for a fast analysis call for data summarization. The main goal of summarization is to apply some techniques to produce small summaries, preserving some particular properties of the original graph, in order to improve some defined measures (e.g. less storage space, better response time, identify some patterns, entity resolution, etc.).

Some of the benefits of graph summarization include:[27]

- Reduction of data volume and storage.
- Speedup of graph algorithms and queries.
- Interactive analysis support. The summarization is introduced to handle information extraction and speedup user analysis.
- Noise elimination. Summarization serves to filter out noise and reveal patterns in the data.

A summarization is often application-dependent and can be defined with respect to various goals: preservation of structural patterns, focus on some entities in the network (e.g. influencers in social networks), preservation of query answers, etc. Graph summarization methods are categorized based on the type of data handled and the core techniques employed.

Overall, the main challenge in summarizing labeled graphs is the efficient combination of two different types of data: structural connections and attributes. We will explore now the different core techniques used to summarize static and labeled graphs.

3.1 CORE TECHNIQUES EMPLOYED

3.1.1 *Grouping-based methods*

Grouping-based methods aggregate nodes into *super-nodes* connected by *super-edges* based on both structural properties and node attributes. Grouped nodes are usually structurally close in the graph and share similar attribute values[27].

Here, attributed clustering, or community detection techniques do not perform summarization. However, they could be leveraged for clustering purposes. In this setting, provided a relevant similarity is maximized, super-nodes can be obtained from such clusters resulting in a compact representation of the original graph.

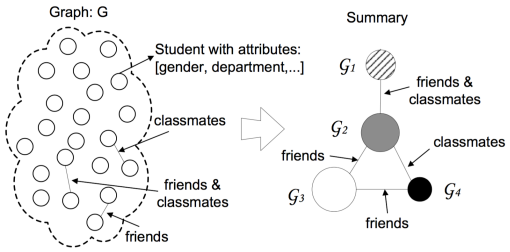


Figure 3.1: SNAP summary. Graph summarization by Aggregation of a student graph.

A fundamental difference between summarizing and clustering is that the former finds coherent sets of nodes with similar connectivity patterns,while clustering results in coherent, densely-connected groups of nodes and the connectivity to the rest of graph is ignored. [27]

SNAP and *k*-SNAP are two popular database-style approaches to summarize graphs. Both of them start by creating groups of nodes that share the same list of user-selected attributes , and then iteratively split these groups until the grouping is compatible with the relationship (*(A,R)-Compatible*, i.e., attribute and relationship compatible). The nodes of the summary graph correspond to the identified groups, also called *super-nodes*, and the edges are the group relationship, which are called *super-edges*. For example, in Figure 3.1, each student in $\mathcal{G}_1$

has at least one friend and one classmate in $\mathcal{G}_2$. The super-node size reflects the number of people per group $\mathcal{G}_i$ [40][27].

3.1.2 *Bit compression-based methods*

In graph summarization, the goal is to use bit compression techniques to minimize the number of bits needed to describe the input graph.[27] The graph summary or model is significantly smaller than the original graph and often reveals various structural patterns, that enhance the understanding of the original graph structure. Most compression-based summaries methods leverage the *Minimum Description Length* (MDL) principle to guide the grouping of nodes or the discovery of frequent structures to be replaced with virtual nodes in the summary.

The first and most prominent frequent-subgraph-based summarization schema, SUBDUE, employs a two-part MDL representation. Beyond the networks structure, the MDL encoding accounts for node and edge labels. This approach is used to iteratively replace the most frequent subgraph in a labeled graph, which minimizes the MDL cost, with a meta-node. Multiple passes of SUBDUE eventually produce a hierarchical description of the structural regularities in the graph. The resulting representation can be used to either identify anomalous structures (instances that do not compress well) or the most common substructures (substructures that have very low compression cost). For example, SUBDUE allows to discover of substructures in chemical compounds graphs (where the vertices are atoms and the edge are bonds).

3.1.3 *Influence-based methods*

Influence-based summarization methods for labeled graphs are currently scarce[27]. The representative method in this category leverages both structural and node attribute similarities to summarize the influence or diffusion process in a large-scale network. The sole work in this category, VEGAS, summarizes influence propagation in citation networks via a matrix decomposition-based algorithm. The summarization problems aim to find the community membership matrix of the nodes such that the similarity is minimized.

3.2 CHALLENGES

The goals of graph summarization can be numerous,among which mitigating the information overload, but also improving the efficiency and query execution time, or to allow pattern discovery maintaining some measure like "interesting", or to improve an influence analysis. However, several challenges may arise:
(i) *The volume and complexity of the data.* The requirement of efficiency to deal with a large amount of data, and their possible heterogeneous sources (CSV files formats, databases, images, etc) will lead to exacerbate the problem of summarization due to data integration needs. Finally, real datasets often contain noise or missing information which may also interfere with the quality of the output.
(ii) *The complexity of the representation.* As we are addressing the property graphs model, the summarization algorithm requires a detailed design that takes into accounts all the attributes of the vertices and edges. Characteristics like the heterogeneity and density, the presence of multi-labels or a multi-valued property graph model (see chapter 1) will determine the time complexity during the construction of such summary.
(iii) *Evaluation.* The evaluation of such summary also represents the main challenge. Using a database perspective, a summary can be considered good if it has a high accuracy in the evaluation of queries, but also it should have a significant runtime gain that justifies the construction of such summary. A translation algorithm might be necessary to translate the queries on the original graph to corresponding queries on the graph summary.

As demonstrated by these challenges, graph summarization is a difficult and multi-faceted problem.

4 GRAPH DATABASES

A graph database is a database designed to treat the relationships between data as equally important as the data itself. It is intended to hold data without constricting it to a predefined model. Instead, the data are stored showing how each individual entity connects with or is related to others.

Only a database that natively embraces relationships is able to store, process, and query connections efficiently. While other databases compute relationships at query time through expensive JOIN operations, a graph database stores connections alongside the data in the model.

Here a brief description of some of the graph databases. We choose these engines because they are most widely-used in practice, but they exhibit significant differences.

4.1 NEO4J

Neo4j is an open-source native graph database that offers functionality similar to traditional RDBMSs such as full transactional support, a declarative query language (Cypher), availability and scalability through a distributed version. The major benefit of Neo4j is its intuitive way of modeling and querying graph-shaped data. Internally, it stores edges as double linked lists. Properties are stored separately, referencing the nodes with corresponding properties.[31]

Cypher introduced the idea of using visual ASCII-Art[1] syntax to express the topological constraints of a pattern. This syntax made pattern matching functionality approachable, firmly established declarative pattern matching in the market, and was a great success with real-world users.[38]

Example 6 *A BGP query using the Cypher syntax matching all the pairs of people who are the know- of-a-known, using the graph of Figure 1.1.*

```
MATCH (x1:Person) -[:knows]-> (:Person) <-[:knows]- (x2:Person)
RETURN x1, x2
```

The above query will match two paths in $\mathcal{G}$: $(n_1, e_1 \cdot e_9, n_4)$ and $(n_1, e_4 \cdot e_2, n_3)$. The result will be then the two pairs: (n_1, n_4) and (n_1, n_3).

Cypher, unlike other languages, uses isomorphism-based no-repeated-edges bag semantics[4] (Section 2.1), and thus the evaluation of the query on Example 6 against $\mathcal{G}$ would not include the match that sends both $x1$ and $x2$ to the same node n_1 (resp. n_4), since it would require mapping the same edge e_1 (resp. e_4) twice in a single match.

Example 7 *A RPQ query using the Cypher syntax. This expression selects pairs of nodes that are linked by a path completely labeled by the knows predicate included the empty set. To do this, it applies the Kleene star operator (i.e. *) over the label knows.*

```
MATCH (x1:Person) -[:knows*]-> (x2:Person)
RETURN x1, x2
```

Currently, Cypher does not support full regular expressions; however, still allows transitive closure over a single label. Apart from this restriction, Cypher also offers the repetition quantifiers for specifying the length of the path. For instance, in Example 7, we can also use $-[:knows*2..4]->$ as a path constraint to specify that the path must be between two and four edges.

4.2 ALLEGROGRAPH

AllegroGraph is a high-performance, software-oriented database model that came as a precursor to the current generation of graph databases. It is implemented as an RDF database, and serves

1 ASCII art is a graphic design technique that uses computers for presentation and consists of pictures or representations pieced together defined by the ASCII Standard.

as a reference implementation for the SPARQL query language. Implementations of geo-temporal reasoning and social network analysis extend the functionality of the database as well as a Prolog extension. Although it was born as a graph database, its current development is oriented to meet the Semantic Web standards (i.e., RDFS, SPARQL, and OWL).[45]

SPARQL is a declarative language recommended by the W3C for querying RDF graphs. The basic building blocks of SPARQL are *triple patterns*, which are RDF triples where the subject, object or predicate may be a variable. SPARQL supports all the complex graph pattern features. The evaluation of BGPs is done following homomorphism-based bag semantics that was explained in Section 2.1. Originally, the simple path queries were based on a bag semantics too. However, since it was shown that such semantics quickly renders query evaluation impractical, the semantics was changed. Now, in order to evaluate any query containing the transitive cloture (* or +), SPARQL uses a set semantics. Otherwise, if a property path can be rewritten as a BGP, SPARQL instead uses the bag semantics defined for BGPs.[4]

Example 8 *A BGP query using the SPARQL syntax matching all the pair of people who are the know-of-a-known, using the graph of Figure 1.1.*

```
PREFIX : <http://my-graph.com/#>
SELECT ?x1, ?x2
WHERE {
    ?x1 :knows ?x0 ,
        (^:knows) ?x2 .
    FILTER (?x1 != ?x2)
}
```

The above query will have the same result as the Example 6, however, this is due to the FILTER clause where we specify that we want an isomorphic no-repeated-node semantic, if we had avoided the use of this clause, the result would have included also the paths: $(n_1, e_1 \cdot e_1, n_1), (n_1, e_4 \cdot e_4, n_3)$*, etc.*

SPARQL also supports a wide range of *FILTER* expressions, variable assignments, arithmetic operations, conditionals, federation and so forth. Since the version 1.1, it also allows the use of property paths, which are an extended form of regular expression that, beyond usual RPQs, also allows inverses and a limited form of negation.

Example 9 *The same query in Example 7, using the SPARQL syntax.*

```
PREFIX : <http://my-graph.com/#>
SELECT ?x1, ?x2
WHERE {
    ?x1 (:knows*) ?x2 .
    FILTER (?x1 != ?x2)
}
```

4.3 ORACLE PGX

Parallel Graph AnalytiX (PGX) is a fast, parallel, in-memory graph analytic framework developed by Oracle Lab, which allows users to load customized graphs into memory, run efficient algorithms and query them. Some of these algorithms are part of the runtime of the Green-Marl[2] Domain Specific Language. PGX provides rich features such as graph pattern matching, built-in parallel graph algorithms, customization of graph algorithms, and interactive shell. It even has a powerful declarative graph query language PGQL that provides many familiar functionalities in a SQL-like syntax, such as aggregation, order by, group by, and negation.

PGQL was inspired by both SQL and Cypher and features a similar ASCII-Art pattern syntax. PGQL was the first property graph query language that supported the regular path patterns. Syntactically, PGQL aims to follow SQL syntax where possible.

2 Green-Marl is a domain-specific language written in C++ that is specially designed for graph data analysis.

PGX Engine is focused on providing an alternative solution to the *subgraph isomorphism problem*. Theoretically, the problem of subgraph isomorphism involves finding all subgraphs of graph $\mathcal{G}$ that are isomorphic to another graph Q. The graph $\mathcal{G}$ (the data graph) is typically huge while the graph Q (the query graph) is typically small. The term isomorphic in the above definition says that there is a one-to-one mapping between the nodes and edges of the two graphs (Section 2.1). Note that when the nodes (edges) of the graph Q have certain properties, the matching nodes (edges) in graph $\mathcal{G}$ must have the same-valued properties.

Example 10 *A BGP query using the PGQL syntax matching all the pair of people who are the know- of-a-known, using the graph of Figure 1.1.[34]*

```
SELECT x1, x2
MATCH (x1: Person) -[:knows]-> () <-[:knows]- (x2:Person)
WHERE x1 != x2
```

The above query will have the same result as the Example 6 and 8, however, in PGQL as in SPARQL, we need to introduce a non-equality constraint in the WHERE clause to have an isomorphic match similar to Neo4j engine with the difference that the introduced clause ensure a ***no-repeated-node semantics****, whereas Neo4j has a* ***no-repeated-edge semantics****. The result is preserved because the graph $\mathcal{G}$ don't have multi-edges.*

The built-in semantic of PGQL is based on graph homomorphism, none additional restriction imposed, but patterns can still be matched in an isomorphic manner by specifying *non-equality constraints* between vertices and/or edges as we can appreciate in the previous example.

Example 11 *A RPQ query using the PGQL syntax. This expression selects pairs of nodes that are linked by a path completely labeled by the knows predicate included the empty set as the examples 7 and 9.*

```
SELECT x1, x2
MATCH  (x1:Person) -/:knows*/-> (x2:Person)
```

In PGQL syntax, when we want to add path constraints instead of simple graph patterns, we need to use the slash instead of the bracket.

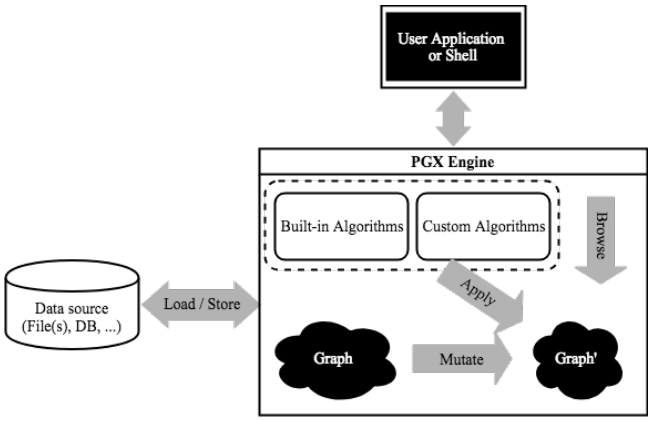


Figure 4.1: PGX Overview

Figure 4.1 depicts an overview of using PGX for graph analysis. Some of its features are [26]:

- Loading graphs into memory. From files or databases.
- Running built-in graph algorithms: PGX provides built-in implementations of many popular graph algorithms.
- Between then we can find: Dijkstra Algorithms, Local Clustering Coefficient, Triangle Counting and Page Rank.
- Running BGP queries or RPQs.

- Mutating Graphs. For example, one may want to create an undirected version of the graph, renumber the vertices
- in the graph, or remove repeated edges between vertices. PGX provides fast, parallel built-in implementation
- of such operations.
- Browsing and exporting results.

PGX also provides a Java API that is designed for synchronous or asynchronous execution [26]. This API also benefits of all the above-detailed features and the latest version is 3.1.

4.4 COMPARISON BETWEEN THOSE THREE LANGUAGES

Language design scope:
All three languages support BGP with enumeration and reachability queries, as well as projection, sorting, filtering, and aggregation of matches. Cypher and SPARQL also support a full data manipulation language (DML) for creating, updating, and deleting nodes and relationships [38]. The future addition of DML capabilities have been discussed for PGQL, but no specification of such features has been put forth so far, however, these operations are possible using the PGX API, where, even if the graph is immutable, it proposes functions to mutate the original graph.

Query structure and clause order:
The languages differ in the chosen syntactic order of clauses. PGQL and SPARQL follow SQL in that it interprets clauses from bottom-to-top (i.e. the first SELECT clause describes the returned data). Cypher, on the other hand, uses top-to-bottom-order as a natural way of expressing sequences of DML statements [38].

Path variables:
Cypher allows storing immutable paths, assigning their value to a variable. PGQL allows declaring a path to be use it in a query, but its value is ephemeral. SPARQL, however, does not allow declaring or assign a path to a variable.

Path patterns:
PGQL and SPARQL have support for the full regular path patterns. Cypher, however, have proposed similar features.

Pattern matching semantics:
As mentioned in the above, while both the SPARQL and PGQL languages are based on graph-homomorphism, they are also amenable to a graph-isomorphism approach, consisting of adding a non-equality constraint. Nevertheless, Cypher uses only a graph-isomorphism no-repeated-edge semantics.

Other notorious differences exist, such as the support for graph construction in Cypher and SPARQL, between those three languages. However, we are not interested in those features for our purpose since the PGX API supports these functionalities.

We chose PGX for the construction of our prototype. The reasons for this choice are various: (a) Unlike existing in-memory solutions, PGX is designed to handle very large graphs; (b) It also captures the inherent large degree of parallelism of the problem and exploits contemporary big multiprocessor machines; (c) It supports full RPQs.

Part II

OUR APPROACH

We illustrate our proposed methodology and its potential in the running example described in Section 5.1. The notations and grammars that we will use are in Section 5.2. The GRASP algorithm and its characterizations are in the sections 5.3 and 5.3.4 respectively.

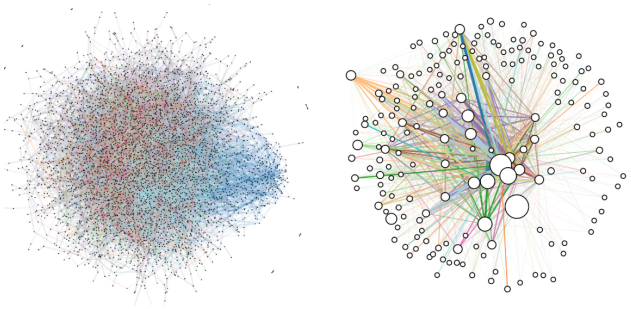


Figure 5.1: Original Graph (left) and Graph Summary (right) for 5K LDBC social network

5.1 CONTEXT

Example 12 (Graph AQP for Social Network Advertising) *Let $\mathcal{G}_{SN}$ (see Fig. 5.2) represent a social network, whose schema is inspired by the LDBC benchmark [20]. Entities can be people (type* `Person`*, P_i) that* know *(l_0) and/or* follow *(l_1) either each other or certain forums (type* `Forum`*, F_i). The latter are* moderated *(l_2) by specific people and can* contain *(l_3) messages/ads (type* `Message`*, M_i), to which persons can* author *(l_4) other messages in* reply *(l_5) or re-share (l_6) such replies (type* `Reply`*, R_i). $\mathcal{G}_{SN}$ exemplifies a graph instance adhering to the property graph model (PGM) that we will define in Section 1 Definition 1. Our goal is to* ***perform graph AQP to obtain high-accuracy, fast, query estimates****.*

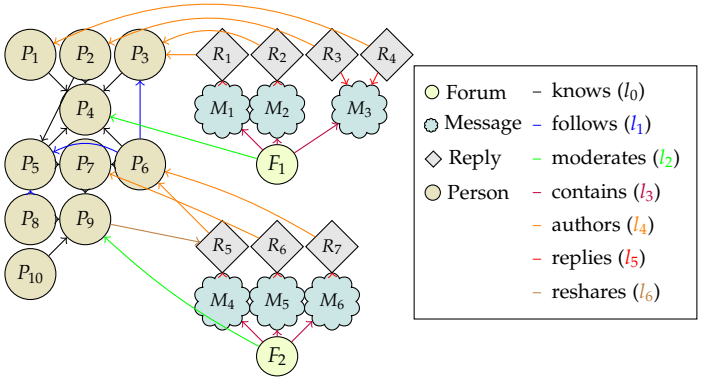


Figure 5.2: Example Social Graph $\mathcal{G}_{SN}$

A practical application in this scenario would be leveraging AQP to obtain targeted advertising markers in social networks. In order to make use of the heterogeneity of real-world networks, we need to express aggregate queries in a query language allowing labeled constraints, corresponding to a dialect of the Regular Path Queries [3, 5, 43]. The expressivity of this language suffices to express the aggregate RPQ-based queries of the types illustrated below [1].

Simple and Optional Label. The count of node pairs that satisfy Q_1, i.e., $()\xrightarrow{l_5}()$, captures the number of ad *reactions*, while the corresponding count for Q_2, i.e., $()\xrightarrow{l_2?}()$ indicates the number of *actual and potential moderators*.

Kleene Plus/Kleene Star. The number of the *connected acquaintances/potentially connected acquaintances* can be determined as the count of node pairs satisfying Q_3, i.e., $() \leftarrow l_0^+()$, and, respectively, Q_4, i.e., $() \leftarrow l_0^*()$

Disjunction. The number of the *targeted subscribers* is computable as the sum of counting all node pairs satisfying Q_5, i.e., $()\xleftarrow{l_4}()$ or $()\xleftarrow{l_1}()$.

1 For ease of exposition, their translation in a high-level syntax is reported in Figure 5.4 in Section 2.

Clauses C	$::= A \leftarrow A_1, \ldots, A_n \mid Q \leftarrow A_1, \ldots, A_n$
Queries Q	$::= Ans(p_1, \ldots, p_m, count(x_1, \ldots, x_n))$
Atoms A	$::= \pi(l_{v_1}, l_{v_2})$, for $l_{v_1}, l_{v_2} \in L_V \mid \leq (l_{v_1}, l_{v_2}) \mid < (l_{v_1}, l_{v_2}) \mid \geq (l_{v_1}, l_{v_2}) \mid > (l_{v_1}, l_{v_2})$
Paths π	$::= \epsilon \mid l_e \mid l_e? \mid l_e^{-1} \mid l_e^*$, for $l_e \in L_E \mid \pi + \pi \mid l_e \cdot l_e$

Figure 5.3: Graph Query Language

Conjunction. The *direct reach* of a company via its page ads can be measured as the count of all node pairs satisfying Q_6, i.e., $()\xleftarrow{l_4}()\xrightarrow{l_5}()$.

A direct evaluation of the above analytical queries on the original graph can be costly in terms of runtime. We develop a novel, *query-oriented*, summarization algorithm aiming at preserving information about *label-constrained reachability* of the initial graph and bookkeeping additional statistics in the node properties of the graph summary and some *topological characteristics* to allow approximate graph query evaluation. In contrast, with existing graph summarization techniques, some of then explained in Section 3 like SNAP, k-SNAP, SUBDUE or VEGAS, which do not take into account the queries to be evaluated on the summary, our summary fulfills this objective for label-constrained reachability queries relevant to graph analytical applications using the grouping-based method (see Section 3.1.1).

The produced graph summary, grouping nodes into *supernodes* (SN) and merging them in *hypernodes* (HN), is guaranteed to be encoded as a property graph similarly to the original graph, thus permitting the evaluation of approximate queries directly in the graph database. Figure 5.1 depicts in the right-hand-side for a quick grasp of the latter in the case of a 5K LDBC social network graph.

5.2 PRELIMINARIES

Graph Model.

We take the property graph model defined in Section 1 and we denote $\mathcal{G} = (V, E)$. We henceforth use a binary notation for edges and, given $e \in E$, $e = l(v_1, v_2)$, we abbreviate l as $\gamma(e)$, v_1 as $e.1$ and v_2 as $e.2$. For a given edge label, l, we abbreviate its number of occurrences in $\mathcal{G}$ as $\#l$. For a label set, $\Lambda = \{l_e^1, \ldots, l_e^n\}$, we denote its associated *frequency list* as $\vec{\Lambda} = [l_1, \ldots, l_n]$, where $l_1 = l_e^{i_1}, \ldots, l_n = l_e^{i_n}$ and $\{i_1, \ldots, i_n\}$ is a permutation of $\{1, \ldots, n\}$, such that $\#l_e^{i_1} \geq \ldots \geq \#l_e^{i_n}$. For a graph $\mathcal{G} = (V, E)$, we denote its set of edge labels as $\Lambda(\mathcal{G}) = \{\gamma(e) \mid e \in E\}$. For a $\mathcal{G}$-subgraph, $\mathcal{G}' = (V', E')$, we denote the set of edge labels incoming to/outgoing from $\mathcal{G}'$ as $\Lambda_+(\mathcal{G}') = \{\gamma(e) \mid e \in E \wedge e.2 \in V' \wedge e.1 \notin V'\}$ and $\Lambda_-(\mathcal{G}') = \{\gamma(e) \mid e \in E \wedge e.1 \in V' \wedge e.2 \notin V'\}$.

Graph Query Language.

To query the above property graph model, we rely on a fragment of the *regular path queries* (RPQ) ([10], [12], [13]), which we enrich with aggregate operators, as depicted in Fig. 5.3. RPQs correspond to property paths in SPARQL 1.1 [41] and are a well-studied query class tailored to express *label-constrained graph reachability patterns*, consisting of one or more *label-constrained reachability paths* (see more details in section 4.2).

Given an alphabet L_E of edge labels l_e and vertices v_1 and v_k, the *labeled path* π (see Definition 3). In their full generality, RPQs allow selecting vertices connected via such labeled paths that belong to a *regular language* over L_E. To our ends, 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 $\pi + \pi$). The expressivity of the fragment we have identified is thus on par with that of Cypher [30]. While not as general as SPARQL, it retains relevance since it, for example, already captures more than 60% of the SPARQL queries with property paths users write in practice, as reported in [9].

$Q_1(l_5)$	$Ans(count(\_)) \leftarrow l_5(\_,\_)$
$Q_2(l_2)$	$Ans(count(\_)) \leftarrow l_2?(\_,\_)$
$Q_3(l_0)$	$Ans(count(\_)) \leftarrow l_0^+(\_,\_)$
$Q_4(l_0)$	$Ans(count(\_)) \leftarrow l_0^*(\_,\_)$
$Q_5(l_4, l_1)$	$Ans(count(\_)) \leftarrow l_4 + l_1(\_,\_)$
$Q_6(l_4, l_5)$	$Ans(count(\_)) \leftarrow l_4 \cdot l_5(\_,\_)$

Figure 5.4: Targeted Advertising Marker Queries

Moreover, this also captures queries with property paths, as found in both the Wikidata online query collection [42] and the Wikidata large corpus studied in [28]. These are further enriched with the *count* operator, in order to support basic graph reachability estimates.

Example 13 *We report in Figure 5.4 the queries of Example 12 expressed by using the syntax of Figure 5.3.*

5.3 GRASP ALGORITHM

Let us assume a graph $\mathcal{G} = (V, E)$ and an edge label set $\Lambda_Q \subseteq \Lambda(\mathcal{G})$. We introduce the *GRASP summarization* algorithm, which compresses $\mathcal{G}$ into an AQP-amenable property graph, $\hat{\mathcal{G}}$, that is tailored for counting label-constrained reachability queries (see Fig. 5.3), whose labels are all in Λ_Q.

As described in Algorithm 1, the GRASP summarization consists of three phases. The **grouping phase** computes Φ, a label-driven partitioning of $\mathcal{G}$ into *groupings*, following the label-connectivity on the most frequently occurring edge labels in $\Lambda(\mathcal{G})$. Next, the **evaluation phase** refines the previous step, by further isolating into *supernodes* the grouping components that satisfy a custom property concerning label-connectivity. The **merge phase** then fuses supernodes into *hypernodes*, based on label-reachability similarity conditions, as specified by each heuristic mode m.

Algorithm 1 GRASP($\mathcal{G}$, Λ_Q, m)

Input: $\mathcal{G}$ – a graph; $\Lambda_Q \subseteq \Lambda(\mathcal{G})$ – a set of query labels; m – heuristic mode
Output: $\hat{\mathcal{G}}$ – a graph summary

1: $\Phi \leftarrow$ GROUPING($\mathcal{G}$)
2: $\mathcal{G}^* \leftarrow$ EVALUATION(Φ, Λ_Q)
3: $\hat{\mathcal{G}} \leftarrow$ MERGE($\mathcal{G}^*, \Lambda_Q, m$)
4: **return** $\hat{\mathcal{G}} = (\hat{V}, \hat{E})$

The GRASP summarization phases are detailed in Sections 5.3.1, 5.3.2, and 5.3.3. In Section 5.3.4, we give a characterization of the underlying algorithm.

5.3.1 *Grouping Phase*

The **grouping phase** aims to output a partitioning Φ of $\mathcal{G}$, such that $|\Phi|$ is *minimized* and, for each $\mathcal{G}_i \in \Phi$, the number of occurrences of the most frequent edge label in $\Lambda(\mathcal{G}_i)$, $\max_{l \in \Lambda(\mathcal{G}_i)} (\#l)$, is *maximized*. To this end, we first sort the set of edge labels in $\mathcal{G}$, $\Lambda(\mathcal{G})$, into a *frequency list*, $\overrightarrow{\Lambda(\mathcal{G})}$. Next, for each $l_i \in \overrightarrow{\Lambda(\mathcal{G})}$, in descending frequency order, we set to identify the largest subgraphs of $\mathcal{G}$ that are weakly-connected on l_i. By relying on a *most-frequently-occurring-label* heuristic, we thus bias the graph partitioning towards a coarser-level of granularity.

The key notion required to define Φ is that of *maximal weak label-connectivity*, introduced below. Let us first introduce needed preliminary notions. In the following, we denote by $\overline{\mathcal{G}} = (V, \overline{E})$, where $|E| = |\overline{E}|$, the transformation of $\mathcal{G}$ into an undirected graph. Also, we say that a graph $\mathcal{G}'$ is a subgraph of $\mathcal{G}$ if and only if $V' \subseteq V$ and $E' \subseteq E$.

Definition 8 (Weak Connectivity) $\mathcal{G}$ is *weakly connected* iff $\overline{\mathcal{G}}$ is *connected*, i.e., there exists a path between any pair of vertices in V.

Definition 9 (Maximal Weak Connectivity) A subgraph of $\mathcal{G}$, $\mathcal{G}' = (V', E')$ is *maximal weakly connected* iff: 1) $\mathcal{G}'$ is weakly connected and 2) no edge in E connects any of the nodes in V' to $V \setminus V'$.

As we are interested in compressing $\mathcal{G}$ by label-connectivity, we strengthen the definition of weak-connectivity below.

Definition 10 (Weak Label-Connectivity) Given a graph $\mathcal{G} = (V, E)$ and a label $l \in \Lambda(\mathcal{G})$, we say that $\mathcal{G}$ is *weakly label-connected on* l iff: 1) when converting all edges in E into undirected ones, the resulting graph, $\overline{\mathcal{G}} = (V, \overline{E})$, where $|E| = |\overline{E}|$, is *connected* and 2) when removing any edge labeled with l from $\overline{E}$, there exist vertices v_1, v_2 in V that are not connected in $\overline{\mathcal{G}}$ by a path labeled l^+.

Since we aim to capture as many vertices in each *weakly label-connected* subgraph of $\mathcal{G}$, we further enforce on the latter the notion of *maximality*, as defined next.

Algorithm 2 GROUPING($\mathcal{G}$)

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

▷ Initialization
1: $n \leftarrow |\Lambda(\mathcal{G})|$, $\overrightarrow{\Lambda(\mathcal{G})} \leftarrow [l_1, \ldots, l_n]$, $\Phi \leftarrow \emptyset$, $i \leftarrow 1$
▷ Label-driven partitioning computation
2: **for all** $l \in \overrightarrow{\Lambda(\mathcal{G})}$ **do**
3: $\quad \mathcal{G}_i \leftarrow \{\mathcal{G}_i^k = (V_i^k, E_i^k) \subseteq \mathcal{G} \mid \lambda(\mathcal{G}_i^k) = l\}$
4: $\quad \Phi \leftarrow \Phi \cup \{\mathcal{G}_i\}$
5: $\quad V \leftarrow V \setminus \bigcup_{\mathcal{G}_i^k \in \mathcal{G}_i} \{v \in V \mid v \in V_i^k\}$
6: $\quad i \leftarrow i + 1$
7: $V(\Phi) \leftarrow \bigcup_{\mathcal{G}_i \in \Phi} \bigcup_{\mathcal{G}_i^k \in \mathcal{G}_i} \{v \in V \mid v \in V_i^k\}$
8: $\Phi \leftarrow \Phi \cup \{\mathcal{G}_i = (V_i, E_i) \subseteq \mathcal{G} \mid V_i = V \setminus V(\Phi)\}$
9: **return** Φ

Definition 11 (Maximal Weak Label-Connectivity) Given a graph $\mathcal{G} = (V, E)$ and a label $l \in \Lambda(\mathcal{G})$, we say that a subgraph of $\mathcal{G}$, $\mathcal{G}' = (V', E')$, is *maximal weakly label-connected* on l, denoted as $\lambda(\mathcal{G}') = l$, iff: 1) $\mathcal{G}'$ is *weakly label-connected on* l and 2) no edge in E, with label l, connects any of the nodes in V' to $V \setminus V'$.

Based on the above, we can now outline the **grouping phase**, as described in Algorithm 2.

We henceforth denote $\Phi = GROUPING(\mathcal{G})$ and name each $\mathcal{G}'$, $\mathcal{G}' \in \Phi$, a $\mathcal{G}$-*grouping* and each $\mathcal{G}''$, $\mathcal{G}'' \in \mathcal{G}'$, a $\mathcal{G}'$-*subgrouping*. Note that Φ is not unique, as, for $l_1, l_2 \in \Lambda(\mathcal{G})$, such that $\#l_1 = \#l_2$, we arbitrarily order l_1 and l_2 in $\overrightarrow{\Lambda(\mathcal{G})}$.

Definition 12 (Non-Trivial (Sub)Groupings) A $\mathcal{G}$-grouping, $\mathcal{G}'$, $\mathcal{G}' = (V', E')$, is called *trivial*, if $\mathcal{G}' = \mathcal{G}$ or $E' = \emptyset$, and *non-trivial*, otherwise. A $\mathcal{G}'$-subgrouping, $\mathcal{G}'' = (V'', E'')$, is called *trivial*, if $E'' = \emptyset$, and *non-trivial*, otherwise.

Lemma 1 (Non-Trivial Grouping Properties) *Let $\mathcal{G}'$ be a non-trivial $\mathcal{G}$-grouping. Then, it holds that:*
P1: *For any non-trivial $\mathcal{G}'$-subgrouping, $\mathcal{G}''$, there exists l'', $l'' \in \Lambda(\mathcal{G}')$, such that $\lambda(\mathcal{G}') = l''$*
P2: *For any non-trivial distinct $\mathcal{G}'$-subgroupings, $\mathcal{G}_1''$, $\mathcal{G}_2''$: a) $\lambda(\mathcal{G}_1'') = \lambda(\mathcal{G}_2'')$ and b) $\mathcal{G}_1''$ and $\mathcal{G}_2''$ are edge-wise disjoint.*

Proof **P1** is provable by contradiction. If $\nexists l''$, $l'' \in \Lambda(\mathcal{G}')$, such that $\lambda(\mathcal{G}') = l''$, then $E' = \emptyset$, contradicting the non-triviality of $\mathcal{G}'$. **P2.a)** holds by construction and **P2.b)**, by contradiction. Assume $\mathcal{G}_1'' \cap \mathcal{G}_2'' \neq \emptyset$; then, $\mathcal{G}_1''$ and $\mathcal{G}_2''$ share at least a node, which is impossible by construction. ∎

Next, let us characterize the *GROUPING* algorithm. We start with the following observations.

Lemma 2 (Subgrouping Maximal Label-Connectivity) *For each $\mathcal{G}_i$, $\mathcal{G}_i \in \Phi$, each of its* maximally weakly connected *components, $\mathcal{G}_i^k$, $\mathcal{G}_i^k \in \mathcal{G}_i$ is also* maximally label-connected *on l, where $\#l = \max_{l \in \Lambda(\mathcal{G}_i)} (\#l)$.*

Proof By construction, we know that, if $\mathcal{G}_i^k \in \mathcal{G}_i$, then there exists a label l', $l' \in \Lambda(\mathcal{G})$, such that $\lambda(\mathcal{G}_i^k) = l'$. Assume that $l' \neq l$. By definition, there exists at least one edge in E_i^k labeled l. Since

$\mathcal{G}_i^k$ is maximally weakly label-connected on l', then each such edge has to connect vertices that are also connected by an edge labeled l'. As $\#l \geq \#l'$, then there exists at least one pair of vertices in V_i^k that are connected by more edges labeled l than by edges labeled l'. This implies that $\lambda(\mathcal{G}_i^k) = l$, contradicting our hypothesis. ∎

Theorem 1 (*GROUPING* Properties) *If* $|V| \geq 1$, *the following properties hold:*
P1: $\forall \mathcal{G}_i \in \Phi, V_i \neq \emptyset$
P2: $\forall \mathcal{G}_i, \mathcal{G}_j \in \Phi$, *where* $i \neq j$, $\mathcal{G}_i \cap \mathcal{G}_j = \emptyset$
P3: $\bigcup_{i \in [1,k]} V_i = V$ *and* $\bigcup_{i \in [1,k]} E_i \subseteq E$
P4: $\Phi = \{\mathcal{G}_i = (V_i, E_i) \subseteq \mathcal{G} \mid i \in [1, |\Lambda(\mathcal{G})| + 1]\}$

Proof **P1**, **P2**, **P3** trivially hold. Let us prove **P4**. If $E = \emptyset$, $\Phi = \{\mathcal{G}\}$. Otherwise, there exists a label $l \in \overrightarrow{\Lambda(\mathcal{G})}$ and we can construct a grouping $\mathcal{G}_i$, where $\lambda(\mathcal{G}_i) = l$. Assume $\Phi > |\Lambda(\mathcal{G})| + 1$. Then, there exist at least two groupings $\mathcal{G}_i, \mathcal{G}_j \in \Phi$, that have the same most frequently occurring label, l. Since $|\mathcal{G}_i| \geq 1$ and $|\mathcal{G}_j| \geq 1$, each contains a maximally weakly connected component, $\mathcal{G}_i^{k_i}$, respectively, $\mathcal{G}_j^{k_j}$. From Lemma 2, $\lambda(\mathcal{G}_i^{k_i}) = \lambda(\mathcal{G}_j^{k_j})$, contradicting $\mathcal{G}_i \cap \mathcal{G}_j \neq \emptyset$. ∎

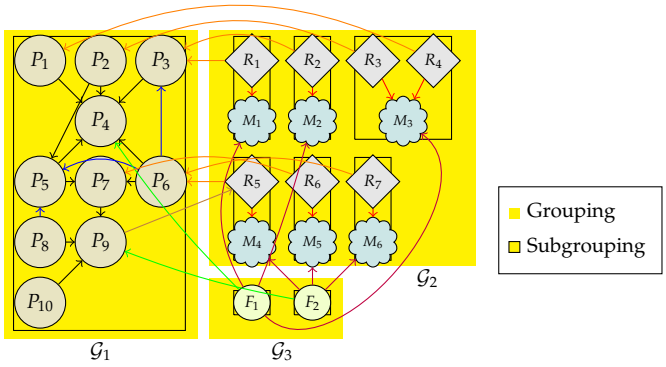


Figure 5.5: Summarizing $\mathcal{G}_{SN}$ (Grouping Phase)

We illustrate the above algorithm, in Figure 5.5, as follows.

Example 14 (Graph Grouping) *Let* $\mathcal{G}$ *be the graph from Figure 5.2. It holds that:* $\#l_0 = 11$, $\#l_1 = 3$, $\#l_2 = 2$, $\#l_3 = 6$, $\#l_4 = 7$, $\#l_5 = 7$, $\#l_6 = 1$. *Hence, we can take* $\overrightarrow{\Lambda(\mathcal{G})} = [l_0, l_5, l_4, l_3, l_1, l_2, l_6]$. *Note that, as* $\#l_4 = \#l_5$, *we can choose an arbitrary order between the labels in* $\overrightarrow{\Lambda(\mathcal{G})}$. *Following Algorithm 2, we first add* $\mathcal{G}_1$ *to* Φ, *as it regroups the maximal weakly-label components on* l_0. *We then have* $V = \{R_1, \ldots, R_7, M_1, \ldots, M_6, F_1, F_2\}$. *Next, we add* $\mathcal{G}_2$ *to* Φ, *as it regroups the maximally weakly-label component on* l_5. *We obtain* $V = \{F_1, F_2\}$. *We add the remaining subgraph,* $\mathcal{G}_3$, *to* Φ *and output* $\Phi = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3\}$, *as illustrated in Figure 5.5.*

5.3.2 *Evaluation Phase*

The **evaluation phase** takes as input a $\mathcal{G}$-partitioning, Φ, as computed by Algorithm 2, along with a set of labels, Λ_Q, and outputs a AQP-amenable compression of $\mathcal{G}$, into a graph $\mathcal{G}^* = (V^*, E^*)$. The phase proceeds in two steps, corresponding to the creation of V^*, the set of *supernodes* (SN), and, respectively, to that of E^*, the set of *superedges* (SE). At the end of each step, $\mathcal{G}^*$ is enriched with AQP-relevant properties that will be exploited in Section 6.

We first explain how *supernodes* are computed. Each such structure (see Definition 13) contains a maximal weakly label-connected subgrouping, as identified in Section 5.3.1.

Definition 13 (Supernodes (SN)) Let Φ be a partitioning of $\mathcal{G}$ into groupings, $\mathcal{G}_i$. Φ is transformed into a set of supernodes, $V^* = VFUSE(\Phi)$, as shown in Algorithm 3, Each *supernode*, $v^* \in V^*$, is obtained through the fusion of all vertices and edges of each subgrouping $\mathcal{G}_i^k$, $\mathcal{G}_i^k \in \mathcal{G}_i$. We denote the label l, such that $\lambda(\mathcal{G}_i^k) = l$, as $\lambda(v^*)$.

Algorithm 3 VFUSE(Φ)

Input: Φ – a graph partitioning; **Output:** V^* – set of *supernodes*

1: $V^* \leftarrow \emptyset$
2: **for all** $\mathcal{G}_i \in \Phi$ **do**
3: **for all** $\mathcal{G}_i^k \in \mathcal{G}_i$ **do**
4: $v_k^* \leftarrow \mathcal{G}_i^k$
5: $V^* \leftarrow V^* \cup \{v_k^*\}$
6: **return** V^*

no

Let $l \in \Lambda(\mathcal{G})$, $v_i^*, v_j^* \in V^*$. We denote the set of l-labeled edges between v_i^* and v_j^* as $E_{i,j}(l) = \{e \in E | e.1 \in v_i^* \wedge e.2 \in v_j^* \wedge \gamma(e) = l\}$ and call each edge in $E_{i,j}(l)$ a *cross-edge*. We associate to each v^*, obtained from a subgrouping $\mathcal{G}_i^k = (V_i^k, E_i^k)$, the property set $\sigma(v^*)$, consisting of:

Compression Properties. We record the number of inner vertices in v^* as $VWeight(v^*) : |V_i^k|$ and the number of inner edges as $EWeight(v^*) : |E_i^k|$.

Label-Connectivity Properties. The percentage-wise occurrence of l in v^* is $LPercent(v^*, l) : \frac{|\{e \in E_i^k |\ \gamma(e) = l|}{EWeight(v^*)}$ and the count of vertex pairs connected with an l-labeled edge is $LReach(v^*, l) : |\{(v_1, v_2) \in V_i^k \times V_i^k |\ l^+(v_1, v_2) \in \mathcal{G}_i^k\}|$.

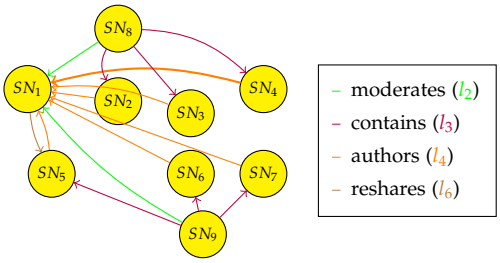


Figure 5.6: Summarizing $\mathcal{G}_{SN}$ (Evaluation Phase)

Pairwise Label-Traversal Properties. Let $l_1, l_2 \in \Lambda(\mathcal{G})$ and $d_1, d_2 \in \{1, 2\}$, direction indices. We compute the number of paths between two cross-edges, with labels l_1, l_2, directions d_1, d_2, and a common node, with the property $EReach(v^*, l_1, l_2, d_1, d_2) = |\{(e_1^*, e_2^*) | \{e_1^*, e_2^*\} \subseteq E^* \setminus E_i^k \wedge \gamma(e_1^*) = l_1 \wedge \gamma(e_2^*) = l_2 \wedge e_1^*.d_1 = v^* = e_2^*.d_2\}|$. We compute the number of *traversal edges*, i.e., inner/cross-edge pairs, e_1, e_2, with respective labels, l_1, l_2, directions, d_1, d_2, and v^* as common endpoint, as $\delta(v^*, l_1, l_2, d_1, d_2) = |\{(e_1, e_2) |\ e_1 \in E^* \setminus E_i^k \wedge \gamma(e_1) = l_1 \wedge e_2 \in E_i^k \wedge \gamma(e_2) = l_2 \wedge e_1.d_1 = v^* = e_2.d_2\}|$. We take the number of *frontier vertices*, for a given label l and direction d, to be $V_F(v^*, l, d) = \{v |\ v \in v^* \wedge \exists e, e \in E \setminus E_i^k \wedge \gamma(e) = l \wedge e.d = v\}|$. We define the *relative label participation* as $RLPart(v^*, l_1, l_2, d_1, d_2) : (\sum_{v \in V_i^k} \delta(v, l_1, d_1, d_2, V \setminus V_i^k))/|V_F(v^*, l_2, d_2)|$, representing the number of l_1-labeled cross-edges relative to that of frontier vertices on l_2.

These properties, illustrated in Example 15, are used in Sec. 6, for counting binary label conjunctions.

Example 15 (Supernode Properties) *In Fig.5.7, we have that* $EReach(v_1^*, l_1, l_2, 1, 1) = 1$, $\delta(v_2^*, l_1, l_2, 1, 2) = 1$ *and* $V_F(v_2^*, l_1, 1) = \{v_1, v_3\}$.

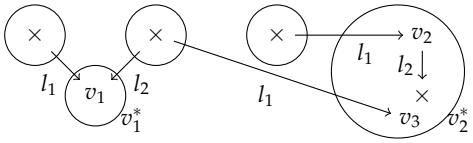


Figure 5.7: Traversal Nodes/Edges and Frontier Vertices

We now proceed to explaining the creation of superedges.

Algorithm 4 VProperties(Φ, Λ)

Input: V – a set of vertices, Λ – a set of labels;
Output: V – property-enriched set of vertices

1: **for all** $v \in V$ **do**
2: $v.vweight \leftarrow VWeight(v)$,
3: $v.eweight \leftarrow EWeight(v)$
4: **for all** $(l_1, l_2) \in \Lambda$, $(d_1, d_2) \in \{1, 2\}$ **do**
5: $v.plabel(l_1) \leftarrow LPercent(v, l_1)$
6: $v.lreach(l_1) \leftarrow LReach(v, l_1)$
7: $v.ereach(l_1, d_1, d_2) \leftarrow EReach(v, l_1, d_1, d_2)$
8: $v.rlpart(l_1, l_2, d_1, d_2) \leftarrow RLReach(v, l_1, l_2, d_1, d_2)$
9: **return** V

Definition 14 (Superedges (SE)) A *superedge*, $e^* \in E^*$, is obtained by merging all cross-edges e, $e \in E_{i,j}(l)$, $l \in \Lambda(\mathcal{G})$, as described in the algorithm presented below. To each such e^* we then associate a weight, $EWeight(e^*) : |\{e \in E \mid e \in e^*\}|$.

The **evaluation phase** is summarized in the next two algorithms. We illustrate below these algorithms by revisiting our running example.

Algorithm 5 EFUSE(V^*, Λ)

Input: Φ – a graph partitioning, Λ_Q – a set of label pairs;
Output: $\mathcal{G}^*$ – a graph with *supernodes* and *superedges*

1: $E^* \leftarrow \emptyset$
2: **for all** $l \in \Lambda$ **do**
3: $E_l \leftarrow \{e \in E | \gamma(e) = l\}$
4: **for all** $v_i^*, v_j^* \in V^*$ **do**
5: **for all** $v_i \in v_i^*, v_j \in v_j^*$ **do**
6: $e^* \leftarrow \{l(v_i^*, v_j^*) | l(v_i, v_j) \in E_l\}$
7: $E^* \leftarrow E^* \cup \{e^*\}$
8: **return** E^*

Algorithm 6 EVALUATE(Φ, Λ)

Input: Φ – a graph partitioning, Λ – a set of labels;
Output: $\mathcal{G}^*$ – a graph with *supernodes* and *superedges*

1: $V^* \leftarrow VFUSE(\Phi)$
2: $V^* \leftarrow VProperties(V^*, \Lambda)$
3: $\hat{E} \leftarrow EFUSE(V^*, \Lambda)$
4: **for all** $e^* \in E^*$ **do**
5: $e^*.weight \leftarrow EWeight(e^*)$
6: **return** $\mathcal{G}^* = (V^*, E^*)$

Example 16 (Graph Compression) *In Figure 5.6, we display the supergraph $\mathcal{G}^*$, obtained from the graph $\mathcal{G} = (V, E)$, after the **evaluation phase**. Each supernode corresponds to a Φ-subgrouping, as depicted in Figure 5.5. Each superedge is obtained by compressing similarly labeled edges, whose source and, respectively, target vertices are in the same supernode. Each superedge e^* has $EWeight(e^*) = 1$, except that connecting SN_4 and SN_1, whose edge weight is 2.*

5.3.3 *Merge Phase*

The **merge phase** takes as input a graph, $\mathcal{G}^*$, as computed by Algorithm 7, along with a set of labels, Λ_Q, and outputs a compressed graph, $\hat{\mathcal{G}} = (\hat{V}, \hat{E})$. The phase proceeds in two steps, corresponding to the creation of $\hat{V}$, the set of *hypernodes* (HN), and, respectively, to that of $\hat{E}$, the set of *hyperedges* (HE). After each step, $\hat{G}$ is enriched with AQP-relevant properties that will be exploited in Section 6.

Hypernodes are computed by merging together supernodes based on various criteria, according to the Definition 15. The primary, *inner-merge*, condition for merging candidate supernodes is for them to be maximal weakly label-connected on the same label. The *source-merge* heuristic additionally requires that they share the same set of outgoing labels, while the *target-merge* heuristic requires that they share the same set of ingoing labels. We define hypernodes as follows.

Definition 15 (Hypernodes (HN)) Let V^* be a set of supernodes, such that $V^* = \{v_1^*, \ldots, v_n^*\}$. V^* is transformed into a set of hypernodes, $\hat{V} = VMERGE(V^*)$, where $\hat{V} = \{\hat{v}_1, \ldots, \hat{v}_m\}$. A *hypernode* $\hat{v}_k \in \hat{V}$ corresponds to fusing a subset V_k^* of V^*, such that, for all $i, j \in [1, n]$, $\{v_i^*, v_j^*\} \in V_k^*$, if $\lambda(v_i^*) = \lambda(v_j^*)$ and either of the following is satisfied: **Case 1.** $\Lambda_+(v_i^*) = \Lambda_+(v_j^*)$, for all $i, j \in [1, n]$. **Case 2.** $\Lambda_-(v_i^*) = \Lambda_-(v_j^*)$, for all $i, j \in [1, n]$.
The first condition corresponds to the target-merge heuristic, while the latter corresponds to the source-merge.

To each such hypernode, $\hat{v}$, we associate a property set.

Definition 16 (Hypernode Properties) The *property set* of $\hat{v}$, $\sigma(\hat{v})$ is the same as in Definition 13. Except for LPercent, the values of hypernode properties are obtained by adding all those corresponding to *supernodes* v^*, such that $v^* \in \hat{v}$. For the label percentage property on a given label l, we have: $LPercent(\hat{v}, l) : \frac{\sum_{v^* \in \hat{v}} LPercent(v^*, l) * EWeight(v^*)}{\sum_{v^* \in \hat{v}} EWeight(v^*)}$.

Superedges are merged into *hyperedges* if they share the same label and endpoints, as defined below.

Definition 17 (Hyperedges (HE)) Let E^* be a set of superedges, $E^* = \{e_1^*, \ldots, e_n^*\}$. E^* is transformed into a set of hyperedges, $\hat{E} = EMERGE(E^*)$, where $\hat{E} = \{\hat{e}_1, \ldots, \hat{e}_m\}$. A *hyperedge*, $\hat{e}_k$, $\hat{e}_k \in \hat{e}$, corresponds to fusing a subset E_k^* of E^*, such that, for a label $l \in \Lambda(\mathcal{G})$ and $i, j \in [1, n]$, $\{e_i^*, e_j^*\} \subseteq E_k^*$, if $e_i^* \stackrel{l}{=} e_j^*$, $e_i^*.1 = e_j^*.1$, and $e_i^*.2 = e_j^*.2$. To each hyperedge $\hat{e}$ we associate a weight, $EWeight(\hat{e})$, corresponding to $|E_k^*|$, the number of superedges $\hat{e}_k$ compressed.

The **merge phase** is captured in Algorithm 7.

Algorithm 7 MERGE($\mathcal{G}^*$, Λ, m)

Input: $\mathcal{G}^*$ – a graph; Λ_Q – a set of label pairs;
m – heuristic mode
Output: $\hat{\mathcal{G}}$ – a graph summary

1: $\hat{V} \leftarrow VMERGE(V^*, \Lambda, m)$
2: $\hat{V} \leftarrow VProperties(\hat{V}, \Lambda)$
3: $\hat{E} \leftarrow EFUSE(E^*, \Lambda(\mathcal{G}^*))$
4: **for all** $\hat{e} \in \hat{E}$ **do**
5: $\quad \hat{e}.eweight \leftarrow EWeight(\hat{e})$
6: **return** $\hat{\mathcal{G}} = (\hat{V}, \hat{E})$

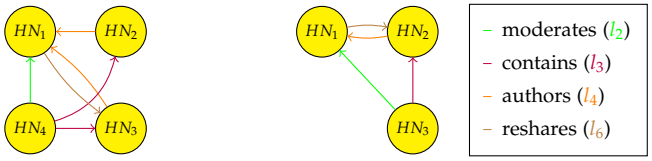
Algorithm 8 VMERGE(V^*, Λ, m)

Input: V^* – a set of supernodes; Λ – a set of labels;
m – heuristic mode
Output: $\hat{V}$ – a set of hypernodes

1: **for all** $v^* \in V^*$ **do**
2: $\Lambda_+(v^*) \leftarrow \{l \in \Lambda \mid \exists v_s^*, v_s^* \in V^* \wedge l(v_s^*, v^*) \in E^*\}$
3: $\Lambda_-(v^*) \leftarrow \{l \in \Lambda \mid \exists v_t^*, v_t^* \in V^* \wedge l(v^*, v_t^*) \in E^*\}$
4: **for all** $v_1^*, v_2^* \in V^*$ **do**
5: $b_\lambda \leftarrow \lambda(v_1^*) = \lambda(v_2^*)$ ▷*Inner-Merge Condition*
6: $b_+ \leftarrow \Lambda_+(v_1^*) = \Lambda_+(v_2^*)$, $b_- \leftarrow \Lambda_-(v_1^*) = \Lambda_-(v_2^*)$
7: **if** $m =$ `true` **then**
8: $\hat{v} \leftarrow \{v_1^*, v_2^* \mid b_\lambda \wedge b_+ =$ `true`$\}$ ▷*Target-Merge*
9: **else**
10: $\hat{v} \leftarrow \{v_1^*, v_2^* \mid b_\lambda \wedge b_- =$ `true`$\}$ ▷*Source-Merge*
11: $\hat{V} \leftarrow \{\hat{v}_k \mid k \in [1, |V^*|]\}$
12: **return** $\hat{V}$

Finally, we depict the resulting GRASP-summarization of our running example, as follows .

Example 17 (Graph Compression) *The graphs in Figure 5.8 are obtained from $\mathcal{G}^* = (V^*, E^*)$, after the **merge phase**. Each hypernode corresponds to the fusion of the supernodes in Figure 5.6, according to the heuristics target-merge (left) and source-merge (right).*



(a) Target Merge heuristic. (b) Source Merge heuristic.

Figure 5.8: Summarizing $\mathcal{G}_{SN}$ (Merge Phase)

5.3.4 *GRASP Characterization*

Theorem 2 (GRASP Invariants) *For a graph $\mathcal{G} = (V, E)$, $GRASP(G) = \hat{G}$, where $\hat{G} = (\hat{V}, \hat{E})$ such that:*

P1: $|\hat{V}| \ll |V|$
P2: $|\hat{E}| \ll |E|$
P3: $\sum_{\hat{v} \in \hat{V}} \hat{v}.vweight = |V|$
P4: $\sum_{\hat{v} \in \hat{V}} \hat{v}.eweight + \sum_{\hat{e} \in \hat{E}} \hat{e}.weight = |E|$
P5: $\Lambda(\hat{\mathcal{G}}) \cup (\sigma(\gamma(\hat{E})) \cap \Lambda(\mathcal{G})) = \Lambda(\mathcal{G})$
P6: *the same pair of HNs cannot be related by multiple HEs with the same label, i.e.,* $\forall \hat{e}_1, \hat{e}_1 \in \hat{E}, \nexists \hat{e}_2, \hat{e}_2 \in \hat{E} \wedge \gamma(\hat{e}_1) = \gamma(\hat{e}_2) \wedge \hat{e}_1.1 = \hat{e}_2.1 \wedge \hat{e}_1.2 = \hat{e}_2.2$

Proof By induction on $|\Lambda(\mathcal{G})|$, based on Theorem 1. ∎

We show the intractability of the graph summarization problem, under the conditions of our algorithm in the Annexes A.1.

As it is explained in the previous section, the main goal of the GRASP Algorithm is to produce a summary graph with meta-information that it will allow approximate the *counting* queries.

6.1 QUERY TRANSLATIONS

Given an input graph $\mathcal{G}$ and a counting reachability query Q, we aim to approximate the result $[\![Q]\!]_{\mathcal{G}}$ of evaluating Q over $\mathcal{G}$. To this end, we translate Q into a query Q^T to be evaluated over the summarization $\hat{\mathcal{G}}$ of $\mathcal{G}$, such that $[\![Q^T]\!]_{\hat{\mathcal{G}}} \approx [\![Q]\!]_{\mathcal{G}}$. The translations for each input query type are presented in Figure A.2 (in the annexes), where we use PGQL [35] as a concrete syntax. We discuss each query class next.

Simple and Optional Label (Q_L, Q_O) There are two plausible configurations in which the label l can occur in $\hat{\mathcal{G}}$: either inside a HN or on a cross-edge. In the first case, we cumulate the number of l-labeled HN inner-edges; in the second, we cumulate the l-labeled cross-edge weights. To account for the potential absence of l, in the optional-label query Q_2, we additionally estimate the number of nodes in $\mathcal{G}'$, by cumulating the number of nodes in each HN.

Kleene Plus and Kleene Star (Q_P, Q_S) To estimate l^+, we cumulate the counts inside hypernodes containing l-labeled inner-edges and, as in (6.1), the weights on l-labeled cross-edges. For the first part, we use the statistics gathered during the *evaluation phase* (Sec. 5.3.2). We distinguish three scenarios, depending on whether the l_+ reachability is due to: 1) inner-edge connectivity – in which case we use the corresponding property counting the inner l-paths; 2) incoming cross-edges – hence, we cumulate the l-labeled in-degrees of HN vertices; or 3) outgoing cross-edges – in which case we cumulate on the number of outgoing l-paths. To handle the ϵ-label in l^*, we use the same formula as in (6.1) to additionally estimate the number of nodes in $\hat{\mathcal{G}}$.

Disjunction (Q_D) As in (6.1), we treat each configuration, considering both labels. In the first case, we cumulate the number of l_1 or l_2-labeled HN inner-edges; in the second, we cumulate over the cross-edge weights with either label.

Binary Conjunction (Q_C) We consider all possible configurations, depending on whether: 1) the label concatenation $l_1 \cdot l_2$ appears on a path *inside* a HN, 2) one of the labels l_1, l_2 occurs on a HN inner-edge and the other, as a cross-edge, or 3) both labels occur on cross-edges.

Example 18 (Approximating Brand Reach Estimates) *Revisiting Example 12, we evaluate the AQP-translation in Figure A.2(in the annexes)over the GRASP summary $\hat{\mathcal{G}} = (\hat{V}, \hat{E})$ in Fig. 5.8, as follows:*

$[\![Q_1]\!]_{\hat{\mathcal{G}}} = Q_L^T(l_5) = \sum_{\hat{v} \in \hat{V}} EWeight(\hat{v}, l_5) * LPercent(\hat{v}, l_5).$

Hence, $[\![Q_1]\!]_{\hat{\mathcal{G}}} = EWeight(HN_2, l_5) * LPercent(HN_2, l_5) = 7$

$[\![Q_2]\!]_{\hat{\mathcal{G}}} = Q_O^T(l_2) = Q_L^T(l_2) + \sum_{\hat{v} \in \hat{V}} AvgSNVWeight(\hat{v}) * VWeight(\hat{v}).$

Hence, $[\![Q_2]\!]_{\hat{\mathcal{G}}} = Q_L^T(l_2) = 27$

$[\![Q_3]\!]_{\hat{\mathcal{G}}} = Q_P^T(l_0) = \sum_{\hat{v} \in \hat{V}} LReach(\hat{v}, l_0) + \sum_{\hat{e} \in \hat{E}} EWeight(\hat{e}, l_0)$

Hence, $[\![Q_3]\!]_{\hat{\mathcal{G}}} = \sum_{\hat{v} \in \hat{V}} LReach(\hat{v}, l_0) = 15.$

$[\![Q_4]\!]_{\hat{\mathcal{G}}} = Q_S^T(l_0) = Q_P^T(l_0) + \sum_{\hat{v} \in \hat{V}} AvgSNVWeight(\hat{v}) * VWeight(\hat{v}).$

Hence, $[\![Q_4]\!]_{\hat{\mathcal{G}}} = 40.$

$[\![Q_5]\!]_{\hat{\mathcal{G}}} = Q_D^T(l_4, l_1) = Q_L^T(l_4) + Q_L^T(l_1) = 14.$

$[\![Q_6]\!]_{\hat{\mathcal{G}}} = Q_C^T(l_4, l_5) = 7.$

Next, we will empirically study the error bounds on various datasets with queries translated according to Figure A.2.

We present in this section our extensive empirical evaluation gauging (1) the succinctness of our GRASP summaries and the efficiency of our graph summarization algorithm; and (2) the suitability of GRASP summaries for approximate evaluation of counting label-constrained reachability queries.
Setup, Datasets and Implementation. We have implemented GRASP in Java using OpenJDK 1.8. The engine has been implemented in Java with query calls (in PGQL) to Oracle Labs PGX 3.1 as underlying graph database backend.

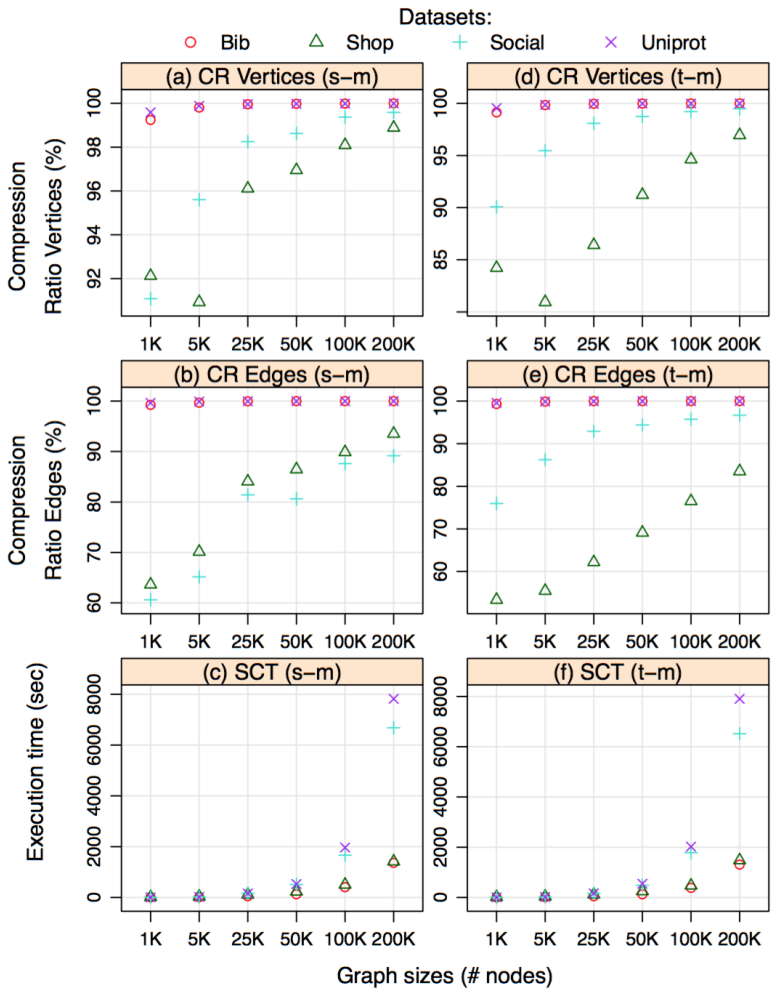
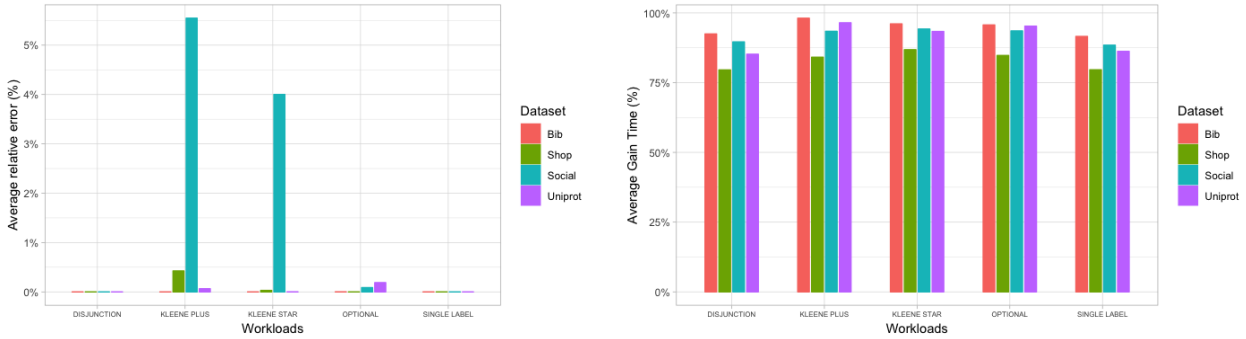


Figure 7.1: Compression Ratios for vertices and edges along with SCT runtime for various sizes of graph datasets for both source-merge (a) to (c) and target-merge (d) to (e).

Since the available version of PGX works on homogeneous graphs rather than on heterogeneous ones, we padded each node in the graph summary with the same properties as in the other nodes. For graph analysis intermediate operations (e.g. weakly connected components), we have used Green-Marl [18], as required by our graph summary construction algorithm. We used four graph datasets for our analysis: (*bib*) a Bibliographic network of varying sizes ($|V|$ from 0.9K to 170K, $|E|$ from 1.3K to 240K), exhibiting 4 edge labels, 5 vertex labels; (*uniprot*) a Uniprot dataset ($|V|$ from 2.1K to 177K, $|E|$ from 3.8K to 773K), having 7 edge labels and 5 vertex labels, encoding Uniprot knowledge graphs [14]; (*social*) a Social network ($|V|$ from 4.4K to 177K, $|E|$ from 10K to 450K) with 27 edge labels and 15 vertex labels, encoding LDBC schema [15]; (*shop*) a Shop dataset ($|V|$ from 3.1K to 109K, $|E|$ from 4.3K to 168K) with 82 edge labels and 24 vertex labels, encoding WatDiv schema [2].

The variations in size of the above datasets have been obtained using gMark [7], a synthetic graph instance and query workload generator. On the same datasets, query workloads of varying size have been generated by reflecting the query characteristics presented in Section 5.2. Precisely, single-label path queries, Kleene-star path queries, path queries with transitive closure ($+$, $*$), path queries with unions and paths queries with concatenation have been considered in each type of workload. Recent studies [9, 28] have shown that practical graph pattern queries formulated by users and in online query endpoints are often small: 56.5% of real-life SPARQL queries consists of a single edge (RDF triple) whereas 90.8% uses 6 edges at most. Hence, we select small-sized template queries whose topology can be found in real-life queries such as chains [9]. These templates are formulated on the four datasets for query workloads containing a total of 616 queries by modifying the vertex labels.



(a) Relative Error per Workload

(b) Time Gain per Workload

Figure 7.2: Experiments on each Dataset, for Single Label, Disjunction, Kleene Star/Plus and Optional Queries.

All experiments were executed on a cloud VM with Intel Xeon E312xx (4 cores) 1.80 GHz CPU and 128GB RAM, running Ubuntu 16.04.4 64-bit. Each data point has been obtained by running the same experiment six times and discarding the first value in the computation of the average.

Compression Ratios of GRASP summaries. First, we evaluate the effect of using the two heuristics (source-merge and target-merge) in the construction of the GRASP summaries. We measure the compression ratio CR obtained on the vertices and edges of the original graph (by using $(1 - |\hat{\mathcal{V}}|/|\mathcal{V}|) * 100$ and $(1 - |\hat{\mathcal{E}}|/|\mathcal{E}|) * 100$, respectively for the CR vertices and edges), along with the summary construction time (SCT). We recall that our graph summaries are encoded using the property graph data model and as such they possess node and edge properties.

In the following, we first discuss the results for one heuristic (source merge) and we defer to the end of this paragraph the comparison between the two heuristics. In Figure 7.1 (a) and (b), we can observe that the most homogeneous datasets (bib) and (uniprot) achieve very high CR (close to 100%) and steadily maintain it when varying the graph sizes. As far as heterogeneity significantly grows for datasets (shop) and (social), the CR becomes eagerly sensitive to the size of the dataset, starting with low values for smaller graph sizes and achieving a plateau between 85% and 90% for larger sizes. As a consequence, our GRASP algorithm let us obtain compact summaries for large sizes of highly heterogeneous datasets. Notice also that the datasets (shop) and (social), which are the most heterogeneous ones, although being close to each other, exhibit a symmetric behavior in terms of CR Vertices and Edges. Social networks get better compressed in terms of vertices, while shop gets better compressed in terms of edges, and vice-versa in our GRASP summaries. When it comes to SCT runtime in Figure 7.1 (c), all datasets keep a reasonable runtime (including the most heterogeneous one (shop)) for larger sizes. We can notice that the SCT runtime is in fact not affected by the heterogeneity degree and is rather sensitive to the number of edges for larger sizes, the latter growing to $450K$ and $773K$ for uniprot and social datasets.

We now contrast the two heuristics source-merge (s-m) and target-merge (t-m), the latter being reported in Figure 7.1 (d-e-f). We can observe that, while the SCT runtime is quite similar for the two heuristics, target-merge exhibits better compression ratios for the social network dataset. The rationale behind it is that the target-merge better compresses the graph edges compared to source-merge. Overall, the dataset with the worst CR across the two heuristics is shop, which has the lowest CR especially for smaller sizes. This is also due to the high number of labels in the initial shop graph instances. Consequently, to the high number of properties necessary for the graph summary for this dataset compared to the other tested datasets: on average across all considered sizes the shop graph summary requires 62,33 properties against 17,67 for the social graph summary, 10,0 properties for the bib graph summary and 14,0 properties for the uniprot graph summary. Nevertheless, even for shop and especially for large sizes, the CR are fairly high. These experiments show that, despite its high complexity, our graph summarization technique behaves well in practice, by providing high CRs and low SCT runtimes.

AQP Accuracy on GRASP summaries. We measured the accuracy and efficiency of our engine by using the usual measures of relative error and time gain, respectively. The relative error (per query Q_i) is given by the following formula: $1 - min(Q_i(\mathcal{G}), Q_i^T(\hat{\mathcal{G}}))/\ max(Q_i(\mathcal{G}), Q_i^T(\hat{\mathcal{G}}))$ (in %), where

Figure 7.3: Relative Error and Time Gain per query, in each Dataset, for Conjunctive Queries, Fig. (above) and (below).

$Q_i(\mathcal{G})$ and $Q_i^T(\hat{\mathcal{G}})$ are the results of the counting query Q_i on the original graph (executed with PGX) and the result of the translated query Q_i^T on the GRASP summary (executed with our engine), respectively. The time gain is given by: $t_{\mathcal{G}} - t_{\hat{\mathcal{G}}}/max(t_{\mathcal{G}}, t_{\hat{\mathcal{G}}})$ (in %), where the times $t_{\mathcal{G}}$ and $t_{\hat{\mathcal{G}}}$ are the query evaluation times of query Q_i on the original graph and on the GRASP summary, respectively.

We have generated workloads for Disjunction, Kleene-plus, Kleene-star, Optional and Single Label workloads, with the size of the workload being bounded by the number of labels in each dataset. For concatenation workloads, we considered binary conjunctive queries with no disjunction, recursion, and optionality. For the time being, our GRASP summaries do not support compositionality in order to cover concatenations of greater length.

Figure 7.2 shows the relative error and the average time gain for Disjunction, Kleene-plus, Kleene-star, Optional and Single Label workloads. In Figure 7.2a, we can observe that the avg relative error is kept low in all cases and is bounded by 5.5% in the case of social dataset's Kleene-plus and Kleene-star workloads. In all the other cases including Kleene-plus and Kleene-star workloads for shop dataset, the error is relatively small and near to 0%. This experiment confirms the effectiveness of our GRASP summaries for approximate evaluation of graph queries. In Figure 7.2b, we studied the efficiency of AQP on GRASP summaries by reporting the time gain (in %) compared with the query evaluation on the original graphs for the four datasets. We can notice a positive time gain (greater or equal to 75%) in the majority of the cases, but the disjunction.

Figure 7.3 (above) and (below) show the comparison among the most heterogeneous datasets (shop and social) on workloads of binary conjunctive queries (on a total of 14 queries, 7 per each dataset). We report the relative error and time gain per query instead of reporting it per workload as before. We can observe in Figure 7.3 (above) a relatively small error for almost all queries (with an average of 1.6%), and an upper bound of 8.44% for query Q_5.

Finally, the shop dataset exhibits a relatively small error with an average of 0.14%. We can observe in Figure 7.3 (below) a fairly high time gain for conjunctive queries, 81.64% faster than on the original graph, for the social dataset, and 70.95% faster for the shop dataset. The difference between social and shop is mainly due to the high heterogeneity of the latter and to the fact that we have to deal with more properties.

8

RELATED WORK

Chaudhuri et al. in [11] have highlighted the importance of AQP in the world of Big data where (i) passing the control to the user to decide the trade-off between accuracy and efficiency and (ii) ensuring accuracy guarantees that are query-independent and applicable to query classes are identified as promising avenues to integrate AQP into Big data platforms. We leverage these two key directions in our work by studying approximate graph query processing and show how blending AQP enables advanced graph analytics.

Previous influential work on APQ has focused on relational languages (SQL and restricted OLAP queries) possibly by embedding samplers directly in the query language and evaluating them in the query plan [1, 11]. Relational rows are sampled uniformly-at-random with a given probability. Uniform sampling is widely supported by RDBMS performing AQP as well as by Big data systems [6, 17, 29, 32, 37] and online aggregation methods [19, 22]. Different classes of samplers ranging from less accurate uniform samplers to distinct samplers and universe sampler working respectively on small groups and on join operators in the query plans have been introduced.

Preliminary work on approximate graph analytics in a distributed setting has recently been pursued in [21]. 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 a query-driven graph summarization for graph navigational queries with aggregates.

RDF graph summarization for cardinality estimation has been tackled in recent work [39]. However, their main goal is RDF query cardinality estimation and, to the best of our knowledge, their graph summary is neither query-driven nor AQP-friendly and their work covers BGP queries.

Aggregation-based graph summarization [24] is at the heart of previous approaches, the most notable of which is SNAP [40]. However, SNAP and its successor k-SNAP [44] summaries are not suitable for AQP and are mainly devoted to discovery-driven graph summarization of heterogeneous networks by relying on a notion of interestingness. These approaches pioneering user interaction as a mean to control the resolutions of the graph summary. However, user intervention here is basically used to drill-down or roll-up to navigate through summaries with different resolutions. More recently, preliminary work by Rudolf et al. [36] has introduced a graph summary suitable for property graphs based on a set of summarization rules provided as input. In the spirit of SNAP, heir summary is conceived for the property graph cube and supports OLAP operations of the kind roll-up, drill-down and slice dice for reducing/expanding the cube dimensions. They further tackle the problems of unbalanced hierarchies and OLAP anomalies. They do not support label-constrained reachability queries as we do in this work.

The AQP++ system [33] blends AQP with aggregate pre-computation such as data cubes in order to deal with aggregate relational queries. They show their superiority with respect to AQP without aggregate pre-computation.

9

CONCLUSION AND FUTURE PERSPECTIVE

This work presents a graph summarization technique suitable for label-constrained counting reachability queries on property graphs. We showed that the problem of deciding whether there exists an optimal graph summary, i.e., such the number of label-constrained graph partitions is minimal, is NP-complete. We leverage the obtained graph summary for approximate graph query processing. The experiments show fairly high compression ratios of our GRASP summaries on various datasets, along with the low relative error and time gain of approximate query processing on the summaries. In future work, we are interested to further explore the interplay between visualization techniques and AQP along with studying further graph analytics and aggregation operators on top of our summaries.

Part III

ANNEXES

ANNEXES 1

A.1 NP-COMPLETENESS PROOF

In this section, we show the intractability of the graph summarization problem, under the conditions of our algorithm.

Definition 18 (Summarization Function) Let $\mathcal{G} = (V, E)$ be a graph and $\Phi = \{\mathcal{G}_i = (V_i, E_i) \,|\, i \in [1, |V|]\}$, a $\mathcal{G}$-partitioning into HNs. Each HN, $\mathcal{G}_i$, contains HN-subgraphs, $\mathcal{G}_i^k$, that are all *maximal weakly label-connected* on a label $l \in \Lambda(\mathcal{G})$, $l \in \Lambda$. A *summarization function* $\chi_\Lambda : V \to \mathbb{N}$ is a function assigning to each vertex, v, a unique HN identifier $\chi_\Lambda(v) \in [1, k]$. χ_Λ is *valid* if, for all vertices, v_1, v_2, such that $\chi_\Lambda(v_1) = \chi_\Lambda(v_2)$, v_1, v_2 belong to either:
Case 1 the same HN-subgraph, $\mathcal{G}_i^k$, that is maximal weak label-connected on l, or to
Case 2 different HN-subgraphs, $\mathcal{G}_i^{k_1}$, $\mathcal{G}_i^{k_2}$, that are each maximal label-connected on l and that are not connected by an l-labeled edge in $\mathcal{G}$.

Theorem 3 (Optimal Summarization NP-completeness) *Let **MinSummary** be the problem that, given a graph $\mathcal{G}$ and an integer $k' \geq 2$, decides whether there exists a label-driven partitioning Φ of $\mathcal{G}$ with $|\Phi| \leq k'$, such that χ_Λ is a* valid summarization. *Then, **MinSummary** is NP-complete. **MinSummary** remains NP-complete, even when restricted to undirected graphs, $|\Lambda(\mathcal{G})| \leq 2$ and $k' = 2$.* ***Proof*** *We establish the result in two steps.*
***MinSummary** is in NP. As a witness, we construct a valid* summarization function, χ_Λ. *For a graph partitioning into k* subgraphs, *one can verify in polynomial time (see [23]) that any two vertices are reachable by a given labeled-constrained path and decide if they belong to the same HN or if their assignation to different HNs is valid (Def. 18).*
***MinSummary** is NP-hard. Let us henceforth reduce the **MinSummary** problem to **IndSet** – the problem of establishing whether an undirected graph contains K independent vertices, for an arbitrary K –, known to be NP-complete (see [16]). We prove **IndSet** $\leq_p$ **MinSummary**. Let $\mathcal{G} = (V, E)$ be an **IndSet** instance, where $\mathcal{G}$ is undirected, $|V| = n \geq 2$, $|E| = m$, $\Lambda(\mathcal{G}) = \{l_1\}$. We consider a polynomial reduction function, f, such that $f(\mathcal{G}) = \mathcal{G}'$, where $\mathcal{G}' = (V', E')$ (see Fig. A.1), $\{v'_1, v'_2, v'_3\} \subset V'$, $\Lambda(\mathcal{G}) = \{l_1, l_2\}$, and $\tilde{\mathcal{G}} \subset \mathcal{G}$, where $\tilde{\mathcal{G}}$ is obtained from $\mathcal{G}$, by adding, between each pair of vertices connected with an l_1-labeled edge, n more l_1-labeled edges. Also let $\mathcal{G}'$ contain three paths of length k, between v'_1 and v'_2 (one that is l_1-labeled and two that are l_2-labeled) and two paths of length n between v'_2 and v'_3 (one of each color). Let K be the number of independent vertices in $\mathcal{G}$, $K \geq 0$. Note that, in $\mathcal{G}'$, $\#l_1 \geq (n+1)(n-K-1) + 2k + n$ and $\#l_2 = 2n + k$. $l_2 = \max_{l \in \mathcal{G}'}(\#l) \Rightarrow K \geq \frac{n^2 - n - 1 + k}{n+1} \geq 1$. We show: $\mathcal{G}$ satisfies **IndSet** $\Leftrightarrow$ $\mathcal{G}'$ satisfies **MinSummary**.*

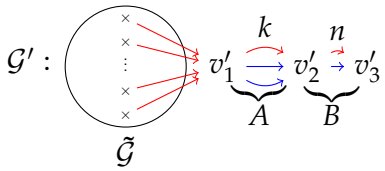


Figure A.1: $\mathcal{G}'$ Construction

$\Rightarrow$ *Let $\mathcal{G}$ satisfy **IndSet**. We can thus choose a set of independent vertices $S \subset V$, $|S| = k$. ap Let $\mathcal{G}_2$ be the $\mathcal{G}'$-subgraph induced by $S \cup A \cup B$. It is* maximal weakly label-connected *on l_2 and contains $2k + n$ edges labeled l_2 and $2k + n$ edges labeled l_1, i.e., $\lambda(\mathcal{G}_2) = l_2$. Let $\mathcal{G}_1$ be the $\mathcal{G}'$-subgraph induced by $V \setminus S$. It is* maximal weakly label-connected *on l_1 and contains $(n+1)m$ edges, all labeled l_1; hence, $\lambda(\mathcal{G}_1) = l_1$. Thus, $\Phi = \{\mathcal{G}_1, \mathcal{G}_2\}$ is a valid summarization of $\mathcal{G}'$, as $l_1 = \max_{l \in \mathcal{G}_1}(\#l)$ and $l_2 = \max_{l \in \mathcal{G}_2}(\#l)$. $\mathcal{G}'$ satisfies **MinSummary**.*

$\Leftarrow$ *Let $\mathcal{G}'$ satisfy* ***MinSummary****. We can thus compute a $\mathcal{G}$-partitioning, Φ, that is a* valid summarization, *where $|\Phi| \leq 2$. If $\Phi = 2$, then there exist two distinct $\mathcal{G}'$-subgraphs, $\mathcal{G}_1$, $\mathcal{G}_2$, where $\Phi = \{\mathcal{G}_1, \mathcal{G}_2\}$. As $\#l_1 = (n+1)m + 2k + n \geq 2n + k = \#l_2$ in $\mathcal{G}'$, one of the subgraphs $\mathcal{G}_1$, $\mathcal{G}_2$, should be such that all of its components are* maximal weakly label-connected *on l_1. Let that subgraph be $\mathcal{G}_1$. Hence, $\mathcal{G}_1 \cap \tilde{\mathcal{G}}$ contains all vertices connected by a l_1-labeled edge. Let us denote by $\tilde{V}_1$ the set of vertices in $\mathcal{G}_1 \cap \tilde{\mathcal{G}}$. The set of vertices in $\mathcal{G}_1$ is thus $\tilde{V}_1 \cup A \cup B$. As Φ has to be a valid summarization, the set of vertices in $\mathcal{G}_2$ is V_2, where $V_2 = V' \setminus (\tilde{V}_1 \cup A \cup B)$. We can thus choose the set of independent vertices of size K in $\mathcal{G}$ to be $S = V_2$. If $|\Phi| = 1$, $\Phi = \{\mathcal{G}'\}$ must be a* valid summarization *of $\mathcal{G}'$. As $\mathcal{G}'$ is* maximal weakly connected *on l_2, it must hold that $l_2 = \max_{l \in \mathcal{G}'}(\#l)$. Hence, $K \geq 1$ and we can choose the set of independent vertices in $\mathcal{G}$ to be $S = V' \cap V$. Thus, $\mathcal{G}$ satisfies* ***IndSet****.* ∎

A.2 QUERY TRANSLATIONS

Figure A.2 depicts the translation, using PGQL syntax, for each query input described in Example 18 in chapter 6.

$Q_L(l)$	`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.RLPART_L2_L1_D2_D1 * e.EWEIGHT)/`
	`(x.LPERCENT_L1 * x.VWEIGHT))`
	`MATCH (x) -[e:l2] -> ()`
	`WHERE x.LPERCENT_L1 > 0`
	`SELECT SUM((y.RLPART_L1_L2_D1_D2 * e.EWEIGHT)/`
	`(y.LPERCENT_L2 * y.VWEIGHT))`
	`MATCH () -[e:l1] -> (y)`
	`WHERE y.LPERCENT_L2 >0`
	`SELECT SUM(x.EWEIGHT *`
	`min(x.LPERCENT_L1, x.LPERCENT_L2))`
	`MATCH (x)`
	`SELECT SUM(x.EREACH_L1_L2_D1_D2) MATCH (x)`

Figure A.2: Query translations onto the graph summary.

BIBLIOGRAPHY

[1] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. "Congressional Samples for Approximate Answering of Group-by Queries." In: *SIGMOD Conference*. ACM, 2000, pp. 487–498.

[2] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. "Diversified Stress Testing of RDF Data Management Systems." In: *International Semantic Web Conference (1)*. Vol. 8796. Lecture Notes in Computer Science. Springer, 2014, pp. 197–212.

[3] Renzo Angles. "The Property Graph Database Model." In: *AMW*. Vol. 2100. CEUR Workshop Proceedings. CEUR-WS.org, 2018.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. "Foundations of Modern Query Languages for Graph Databases." In: *ACM Comput. Surv.* 50.5 (2017), 68:1–68:40.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. "Foundations of Modern Query Languages for Graph Databases." In: *ACM Computing Surveys* 50.5 (2017). DOI: 10.1145/3104031.

[6] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark." In: *SIGMOD Conference*. ACM, 2015, pp. 1383–1394.

[7] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. "gMark: Schema-Driven Generation of Graphs and Queries." In: *IEEE Trans. Knowl. Data Eng.* 29.4 (2017), pp. 856–869.

[8] Pablo Barceló Baeza. "Querying graph databases." In: *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. ACM. 2013, pp. 175–188.

[9] Angela Bonifati, Wim Martens, and Thomas Timm. "An Analytical Study of Large SPARQL Query Logs." In: *PVLDB* 11.2 (2017), pp. 149–161.

[10] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. "Rewriting of Regular Expressions and Regular Path Queries." In: *J. Comput. Syst. Sci.* 64.3 (2002), pp. 443–465.

[11] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. "Approximate Query Processing: No Silver Bullet." In: *SIGMOD Conference*. ACM, 2017, pp. 511–519.

[12] Mariano P. Consens and Alberto O. Mendelzon. "GraphLog: a Visual Formalism for Real Life Recursion." In: (1990), pp. 404–416.

[13] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. "A Graphical Query Language Supporting Recursion." In: *Proceedings of the ACM Special Interest Group on Management of Data*. 1987, pp. 323–330.

[14] Elixir. *Uniprot KB*. https://www.uniprot.org/. 2018.

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

[16] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[17] Google. *Sampling in google bigquery.* https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operators#approx_top_sum.. 2017.

[18] Green-Marl. *Green-Marl DSL*. https://github.com/stanford-ppl/Green-Marl. 2018.

[19] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation." In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD '97. Tucson, Arizona, USA, 1997, pp. 171–182. ISBN: 0-89791-911-4.

[20] Alexandru Iosup et al. "LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms." In: *PVLDB* 9.13 (2016), pp. 1317–1328.

[21] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. "Bridging the GAP: towards approximate graph analytics." In: *GRADES/NDA@SIGMOD/PODS*. ACM, 2018, 10:1–10:5.

[22] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. "Scalable Approximate Query Processing with the DBO Engine." In: *ACM Trans. Database Syst.* 33.4 (Dec. 2008), 23:1–23:54. ISSN: 0362-5915.

[23] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. "Computing Label-constraint Reachability in Graph Databases." In: *SIGMOD Conference*. ACM, 2010, pp. 123–134.

[24] Arijit Khan, Sourav S. Bhowmick, and Francesco Bonchi. "Summarizing Static and Dynamic Big Graphs." In: *PVLDB* 10.12 (2017), pp. 1981–1984.

[25] Markus Krötzsch. *DATABASE THEORY. Lecture 13: Graph Databases and Path Queries*. https://iccl.inf.tu-dresden.de/w/images/5/53/DBT2016-Lecture-13.pdf. 2016.

[26] Oracle Labs. *PGX Docs*. https://docs.oracle.com/cd/E56133_01/latest/reference/overview/. 2017.

[27] Yike Liu, Abhilash Dighe, Tara Safavi, and Danai Koutra. "Graph Summarization Methods and Applications: A Survey." In: *CoRR* abs/1612.04883 (2016). arXiv: 1612.04883. URL: http://arxiv.org/abs/1612.04883.

[28] Stanislav Malyshev, Markus Kröotzsch, Larry Gonzalez, Julius Gonsior, and Adrian Bielefeldt. "Getting the Most out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph." In: *ISWC Conference (to appear)*. LNCS, 2018.

[29] Microsoft. *Microsoft powerbi*. https://powerbi.microsoft.com/en-us/. 2017.

[30] Neo4J. *Cypher Query Language*. https://www.opencypher.org/. 2018.

[31] Inc. Neo4j. *What is Neo4j?* https://neo4j.com/developer/graph-database/#_what_is_neo4j. 2018.

[32] Oracle. *Oracle data mining blog: To sample or not to sample*. https://blogs.oracle.com/datamining/entry/to_sample_or_not_to_sample. 2017.

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

[34] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. "Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism." In: *Proceedings of Workshop on GRAph Data management Experiences and Systems*. ACM. 2014, pp. 1–6.

[35] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. "PGQL: a property graph query language." In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*. 2016, p. 7.

[36] Michael Rudolf, Hannes Voigt, Christof Bornhövd, and Wolfgang Lehner. "SynopSys: Foundations for Multidimensional Graph Analytics." In: *Enabling Real-Time Business Intelligence - International Workshops, BIRTE 2013, Riva del Garda, Italy, August 26, 2013, and BIRTE 2014, Hangzhou, China, September 1, 2014, Revised Selected Papers*. 2014, pp. 159–166.

[37] SnappyData. *SnappyData.IO*. http://www.snappydata.io.. 2017.

[38] US Stefan Plantikow. "Summary Chart of Cypher, PGQL, and G-Core." In: (2018).

[39] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. "Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation." In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. 2018, pp. 1043–1052.

[40] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. "Efficient aggregation for graph summarization." In: *SIGMOD Conference*. ACM, 2008, pp. 567–580.

[41] W3C. *SPARQL 1.1 Query Language*. https://www.w3.org/TR/sparql11-query/. 2013.

[42] Wikidata SPARQL Query Service. *Wikidata Query Examples*. https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples. 2017.

[43] Peter T. Wood. "Query languages for graph databases." In: *SIGMOD Record* 41.1 (2012), pp. 50–60. DOI: 10.1145/2206869.2206879.

[44] Ning Zhang, Yuanyuan Tian, and Jignesh M. Patel. "Discovery-driven graph summarization." In: *ICDE*. IEEE Computer Society, 2010, pp. 880–891.

[45] AllegroGraph ®. *AllegroGraph Support Documentation*. https://franz.com/agraph/allegrograph/.