Distributed Querying of Large Labeled Graphs

Sairam Gurajada

Thesis for obtaining the title of Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken
2016

Colloquium

Date	:	February 6, 2017
Place	:	Saarbrücken
Dean	:	Prof. Dr. Frank-Olaf Schreyer

Examination Board

Chair	:	Prof. Dr. Jens Dittrich
Reviewer & Advisor	:	Prof. Dr. Martin Theobald
Reviewer & Supervisor	:	Prof. Dr. Gerhard Weikum
Reviewer	:	Prof. Dr. M. Tamer Özsu
Reviewer	:	Prof. Dr. Sebastian Michel
Academic Assitant	:	Dr. Luciano Del Corro

Abstract

Graph is a vital abstract data type that has profound significance in several applications. Because of its versitality, graphs have been adapted into several different forms and one such adaption with many practical applications is the "Labeled Graph", where vertices and edges are labeled. An enormous research effort has been invested in to the task of managing and querying graphs, yet a lot challenges are left unsolved. In this thesis, we advance the state-of-the-art for the following query models, and propose a distributed solution to process them in an efficient and scalable manner.

- *Set Reachability.* We formalize and investigate a generalization of the basic notion of reachability, called *set reachability.* Set reachability deals with finding all reachable pairs for a given source and target sets. We present a non-iterative distributed solution that takes only a single round of communication for any set reachability query. This is achieved by precomputation, replication, and indexing of partial reachabilities among the boundary vertices.

- *Basic Graph Patterns (BGP).* Supported by majority of query languages, BGP queries are a common mode of querying knowledge graphs, biological datasets, etc. We present a novel distributed architecture that relies on the concepts of asynchronous executions, join-ahead pruning, and a multi-threaded query processing framework to process BGP queries in an efficient and scalable manner.

- *Generalized Graph Patterns (GGP).* These queries combine the semantics of pattern matching and navigational queries, and are popular in scenarios where the schema of an underlying graph is either unknown or partially known. We present a distributed solution with bimodal indexing layout that individually support efficient processing of BGP queries and navigational queries. Furthermore, we design a unified query optimizer and a processor to efficiently process GGP queries and also in a scalable manner.

To this end, we propose a prototype distributed engine, coined "TriAD" (*Triple Asynchronous and Distributed*) that supports all the aforementioned query models. We also provide a detailed empirical evaluation of TriAD in comparison to several state-of-the-art systems over multiple real-world and synthetic datasets.

Kurzfassung

Graphenorientierte Datenmodelle haben in den vergangenen Jahren zunehmend an Relevanz im Bereich der Datenverarbeitung mittels moderner Informationssysteme gewonnen. Eine sehr vielseitige, allgemeine Form der graphenorientierten Repräsentation von Datenobjekten und deren Beziehungen zueinander bieten sogenannte „beschriftete Graphen“, in denen sowohl die Knoten als auch die Kanten zwischen den Datenobjekten Beschriftungen tragen. Wegen der enorm vielseitigen Anwendbarkeit dieser graphenorientierten Datenmodelle beschäftigt sich eine große Anzahl aktueller Forschungsarbeiten insbesondere mit der verteilten Verarbeitung und Anfragebearbeitung von großen Graphdatensätzen. Dennoch bleiben viele Herausforderungen gerade bezüglich der Effizienz und der Skalierbarkeit dieser Ansätze weiterhin offen. Die vorliegende Dissertation erweitert die aktuellen Forschungsergebnisse für die folgenden Anfragemodelle auf großen, beschrifteten Graphen.

- *Verteilte Mengenerreichbarkeit.* Auf Basis des bekannten Erreichbarkeitsproblems in gerichteten, beschrifteten Graphen formulieren wir eine Verallgemeinerung dieses Problems, welches wir als „verteilte Mengenerreichbarkeit“ bezeichnen. Mengenerreichbarkeit bezeichnet das Erreichbarkeitsproblem für Mengen von Quell- und Zielknoten, zwischen denen wir alle Paare von Quell- und Zielknoten, die jeweils im zu Grunde liegenden Datengraphen erreichbar sind, suchen. Im Gegensatz zu bestehenden Ansätzen zur Anfrageauswertung auf verteilten Graphen präsentieren wir einen nicht-iterativen Lösungsansatz, der nur einen einzigen Kommunikationsschritt zwischen allen Rechenknoten in einem Rechnerverbund benötigt. Diese Garantie gilt für beliebige Graphen und Mengenerreichbarkeitsanfragen und wird durch eine Kombination aus Vorausberechnungen, Replikation und Indexierung der partiellen Erreichbarkeitseigenschaften des partitionierten Datengraphen erreicht.

- *Einfache Graphenmuster.* Anfragen mit sogenannten „einfachen Graphenmustern“ werden von einer Reihe aktueller Anfragesprachen unterstützt und bilden die häufigste Form von Anfragen in semantischen Graphen, biologischen Datensätzen und vielen weiteren Formen von graphenorientierten Daten. Zur effizienten und skalierbaren Auswertung dieser Form von Anfragen präsentieren

wir eine neuartige, verteilte Architektur, die verschiedene Konzepte der Optimierung von Ausführungsplänen innerhalb eines Rechnerverbundes, der parallelen Ausführung dieser Ausführungspläne innerhalb eines jeden Rechenknotens, sowie der asynchronen Kommunikation zwischen den Rechenknoten miteinander verbindet.

- *Verallgemeinerte Graphenmuster.* Diese Form der Anfragen kombinieren einfache Graphenmuster mit zusätzlichen Navigationsbedingungen, die in Form von regulären Ausdrücken zwischen den einfachen Graphenmustern vorliegen. Diese Anfragen kommen insbesondere dann zum Einsatz, wenn das Schema des zu Grunde liegenden Datengraphen nicht oder nur teilweise bekannt ist. Zur verteilten Auswertung dieser verallgemeinerten Graphenmuster präsentieren wir eine Kombination unserer Indexstrukturen zur Auswertung einfacher Graphenmuster mit unseren Indexstrukturen zur Auswertung von Mengenerreichbarkeitsanfragen. Des Weiteren entwickeln wir einen einheitlichen Ansatz zur Optimierung und der – sowohl verteilten als auch parallelen – Auswertung von Anfragen mit verallgemeinerten Graphenmustern.

Zusammenfassend stellt die vorliegende Dissertation die Architektur eines verteilten Prototypens (genannt „TriAD“ für „Triple-Asynchronous-Distributed“) zur effizienten und skalierbaren Auswertung der oben genannten Anfragen auf großen, beschrifteten Graphen vor. Des Weiteren präsentiert die Dissertation eine detaillierte, empirische Evaluation von TriAD im Vergleich zu einer Reihe aktueller Systeme auf großen Graphdatensätzen mit unterschiedlichen Eigenschaften.

To my parents

Acknowledgements

I would take this moment to thank my advisor Prof. Martin Theobald for all the support and freedom he provided to pursue my research interests. I am glad to have an advisor like him, who is very responsible and easily accessible for all the insightful discussions when needed and for all the advice when sought, irrespective of the place he stays. I would also like to thank him for the immense help he provided in writing good research articles.

Many thanks to Prof. Sebastian Michel, Prof. Tamer Özsu, and Prof. Gerhard Weikum for agreeing to review my thesis. I would like to extend my thanks to Prof. Gerhard Weikum and International Max Planck Research School for providing financial assistance and a conducive environment to pursue my studies.

I thank all my colleagues and staff at D5 group for making the workplace an exciting atmosphere. Thanks to Patrick Ernst, Dat Ba Nguyen, and Amy Siu for being wonderful officemates and bringing in a tradition to gift souveniors.

I thank my co-authors Sourav Dutta, Iris Miliaraki, Stephan Seufert, and Jacopo Urbani for their inspiring team work and thoughtful technical discussions.

Thanks to Syama Sundar Rangapuram for helping me to adjust to Saarbrücken and for being an easily approachable person. I thank all my current and former friends in Saarbrücken for making my stay joyful.

Finally, I would like to distinctively thank and remember my father for all the care and constant support he had on me to pursue my goals, all while comprising on his wishes. Special thanks to my mother, brother, and sister for their continuous encouragement. Last, and surely not least, I thank my fiancée Lavanya for patiently waiting to finish my studies.

Contents

Chapter **1**

Introduction

1.1 Labeled Graphs

Graph is a simple, versatile, and an age old model for representing complex data. Starting with the famous "*Seven Bridges of Königsberg*" problem introduced by the mathematician Leonhard Euler in 1798, graphs became inherent in many applications. A graph, basically a pair of vertex and edge sets, capture relationships among a set of objects. Some of the popular scenarios where a graph is a de facto model include social networks, knowledge graphs, biological datasets, etc. Because of its versatility, each application either extend or adapt the graph model to their respective needs. One of the most common adaption or extension is attaching labels to vertices and edges in a graph, and further allowing multiple edges between a pair of vertices. This notion of graph, called "*Labeled Graph*", is a popular graph model for expressing large variety of real-world datasets.

Social networks are one of the instance-classes of labeled graph model and are some of the large graphs available today. These include Facebook [1], Twitter [2], etc., which model the information about people, places, events, things, etc. as vertices and their relationships as edges. Knowledge graphs such as Google's Knowledge Graph (Singhal, 2012), Microsoft's Bing Satori Knowledge Base (Qian, 2013), DBpedia (Bizer et al., 2009), YAGO (Suchanek et al., 2007), etc., on the other hand, encode semantically rich information in the form of facts such as "Albert Einstein born in Ulm" or "Diabetes can cause fatigue". Such an information can collectively form a labeled graph with the entities such as "Albert Einstein", "Ulm" denote vertices and the relation "bornIn" represent an edge in the graph. Applications such as web search or domain-specific medical analysis can benefit from such rich knowledge for enhancing the quality of results. Other networks such as biological datasets (protein-protein interactions, gene co-expressions), LinkedMDB, GeoSpecies are some of the real-world instances of the labeled graph data model.

[1] http://facebook.com
[2] http://twitter.com

1.2 Querying Labeled Graphs

Querying is one of the most common task on labeled graphs. Depending on the application in use, queries can be of different models and are of varying complexities. Typical queries on labeled graphs include simple lookups like "*find whether two persons are friends on a social network*", or navigational queries like "*check if there exists a multi-hop connecting flights between two cities*", or complex queries like "*find all musicians who are born in Germany and won a Nobel Prize in physics*". Although simple lookup and navigational queries can be performed in linear time with respect to the number of edges in a graph, complex queries require a polynomial time in terms of data complexity (Chandra and Merlin, 1977; Vardi, 1982). In the following we list some of the popular query models which are prominently used in several applications.

- *Connectivity Queries.* This class includes simple Boolean *reachability* queries that check whether there exists a path between a pair of vertices s, t in a given graph. Further constraints can be imposed on a path connecting s and t via regular expressions. Generalizing this notion of simple reachability queries, connectivity queries include *set reachability* queries, also known as multi-source multi-target queries (Gao and Anyanwu, 2013), where the goal is to find all reachable pairs for a given pair of source and target vertex sets.

- *Pattern Matching Queries.* Pattern matching queries, on the other hand, are used for querying complex sub-structures, as against connectivity queries which look only for paths. These queries come in two flavors.

    i *Basic Graph Patterns* (BGP) queries comprise of set of triple patterns, each representing zero or more edges. Moreover, a BGP query collectively denotes a set of subgraphs whose vertex (or edge) labels form an answer to the query.

    ii *Generalized Graph Patterns* (GGP) queries, alternatively called conjunctive regular path queries (CRPQs), combine the semantics of BGP and connectivity queries that can be used to expresses more complicated query needs, especially when the schema of the underlying graph is unknown or partially known, or to query transitive relations.

Although, other query forms such as approximate matching, graph creation, and aggregation queries are possible, in this thesis, we specifically focus on the three — *set reachability, basic graph patterns,* and *generalized graph patterns* — query models.

Graph Query Languages. Several query languages such as GraphLog (Consens and Mendelzon, 1990), **G+** (Cruz et al., 1988), UnQL (Buneman et al., 1996), SPARQL 1.1 (Prud'hommeaux et al., 2013), Gremlin (Sun et al., 2015), Cypher (Neo4j, 2012) came to prominence with the wide adoption of graphs across multiple domains. Majority of these graph languages support a wide variety of query models,

and prominently, the aforementioned query models. For comprehensive overview of the graph query languages, we refer the reader to (Angles and Gutierrez, 2008; Wood, 2012).

Graph Querying Systems. With the meteoric rise in adoption and availability of large labeled graph datasets across multitude of applications, efficient management and querying of graphs became one of the active research areas in recent times. Much research went into the development of efficient centralized systems, both in relational world and in native graph-based models. Systems like Neo4j (Neo4j, 2012), TORNADO (Atkinson et al., 1989), FERRARI (Seufert et al., 2013), GRAIL (Yildirim et al., 2010), Virtuoso (Erling and Mikhailov, 2010), RDF-3X (Neumann and Weikum, 2010a), SW-Store (Abadi et al., 2009), Hexastore (Weiss et al., 2008), etc. all provide efficient support for specific query types in querying labeled graphs. Due to inherent hardware limitations and unable to pace with the growing size of real-world graph datasets, centralized systems soon became a bottle-neck in handling large graphs with billion or more edges .

Consequently, distributed systems came into active development with the first prototypes built on top of key-value stores such as MapReduce. These include H-RDF-3X (Huang et al., 2011), EAGRE (Zhang et al., 2013), SHARD (Rohloff and Schantz, 2011), which are scalable to large graphs and support BGP queries, whilst no or inefficient support for connectivity queries. Later on, general purpose iterative-based vertex-centric graph processing frameworks like Apache Giraph (Martella et al., 2015), Apache GraphX (Gonzalez et al., 2014), Microsoft's Trinity (Shao et al., 2013), Google's Pregel (Malewicz et al., 2010) rose to prominence and can be programmed to support multiple query types on labeled graphs. Although these architectures are scalable and can handle graphs with billions of edges, they are not ideal for real-time processing of queries which is essential in many applications.

1.3 Challenges

Next, we list some of the challenges with the existing systems that needs to addressed to build a system that is both scalable and efficient in distributed querying of large labeled graphs.

1. Reachability queries, to a great extent, have been addressed in prior works on centralized systems. While very few works exist for processing set reachability queries in a centralized setting, and moreover, there exist — to the extent of our knowledge — no prior works in a distributed setting, except for the work by (Fan et al., 2012) which provides a solution for distributed single-source single-target reachability. On the other hand, general purpose distributed graph approaches provide a framework for processing set reachability queries over large graphs in a scalable manner, but lack sufficient support to index graphs — like in a centralized setting — to accelerate query processing.

2. MapReduce-based systems rely on relational joins implemented via Map and/or Reduce functions to process BGP queries. Although MapReduce-based joins allow for the execution of multiple join operators in parallel, they need to synchronize at each level of a query plan. These synchronization steps are heavily dominated by a few stragglers or imbalanced query plans. On contrary, general purpose graph engines mitigate the problem by using asynchronous parallel graph explorations instead of relational joins. As major query languages such as Cypher (Neo4j, 2012), SPARQL (Prud'hommeaux et al., 2013), etc., require an SQL-style row-oriented output, graph explorations are usually not sufficient without relational joins to generate final answers.

3. Majority of the existing systems (both centralized and distributed) are designed for specific query types, i.e., they either support connectivity or BGP queries, but not both in a single unified system. Queries, like GGP, which are combination of both connectivity and basic graph patterns are merely supported by a few centralized and distributed systems such as (Erling and Mikhailov, 2010; Gubichev et al., 2013).

1.4 Contributions

To overcome the aforementioned challenges and to design an efficient and scalable engine for distributed querying of labeled graphs, we propose a distributed solution based on the duality of graph and relational concepts. In the following, we list the contributions made in this thesis.

- In Chapter 3, we formalize and investigate the *set reachability* querying model, thus handling both types of connectivity queries. We propose an novel distributed solution based on graph-based index structures that are commonly practiced in centralized architectures. Our index structures allow us to process any set reachability query using a single round of message exchange among the compute nodes irrespective of the topology of the graph and the selectivity of the query. We also discuss methods to update our indexes for dynamic graphs and also using existing centralized indexes as plugins to further accelerate query processing.

  The results of this work was published at (Gurajada and Theobald, 2016b).

- In Chapter 4, we propose a distributed solution to tackle BGP queries following the principles of relational systems. Leveraging an asynchronous communication protocol via MPI (The MPI Forum, 1993), we propose an architecture that supports efficient *asynchronous and parallel join executions* via inter-node distributed and intra-node multi-threading executions. We also propose a novel join-ahead pruning technique to prune *dangling* (irrelevant) tuples during distributed join executions. Finally, we design a distributed-aware query optimizer

that generates efficient plans taking all our ingredients — locality of edges, multi-threading, and join-ahead pruning — in to consideration.

The results of this work was published at (Gurajada et al., 2014a,b).

- In Chapter 5, we deal with GGP queries and propose a distributed solution combining the techniques from Chapter 3 and 4. Building on top of the distributed architecture proposed in Chapter 4, we propose a novel unified query optimization and processing framework, thus, handling both the navigational and basic graph patterns aspects of the GGP queries.

  The results of this work is available at (Gurajada and Theobald, 2016a).

To this end, we develop a prototype engine, coined "**TriAD** (**Tri**ple **A**synchronous and **D**istributed)", that integrates all our techniques and supports efficient distributed querying of labeled graphs for connectivity and pattern matching query types.

1.5 Organization

Rest of the thesis is organized as follows. Chapter 2 provides the necessary background on graphs and relational database systems which is essential for better understanding of our work. We also briefly cover existing graph data management techniques and establish our data and query model as part of preliminaries. In Chapter 3, we formalize and investigate set reachability queries in a distributed setting, and propose a novel technique to process set reachability queries in an efficient and scalable manner. In Chapter 4, we propose a distributed architecture to process BGP queries efficiently. In Chapter 5, we discuss in detail the problem of distributed processing of GGP queries, and propose an architecture that combines the techniques discussed in Chapters 3, 4. Finally, Chapter 6 provides a summary of this thesis, and also we briefly point out few possible directions in future work.

Chapter 2

Background & Preliminaries

This chapter serves to provide a sufficient background on graph and on relational models. The duo forms the key building blocks in labeled graph data management. We also briefly discuss graph data management in practice, covering some of the popular graph data models, query models, and state-of-the-art systems. We then present our data and query models as part of preliminaries. Furthermore, this chapter also serves to establish necessary notations that will be used in the remainder of the thesis.

2.1 Background

2.1.1 Graphs

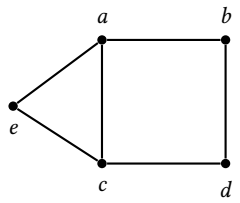
Graphs provide a very versatile, simple, and flexible data model to capture objects and their relationships. Many real-world datasets including social networks, knowledge graphs, and biological datasets are naturally expressible in graph data models.

One of the most concise definitions of a *graph* can be found in the book by Reinhard Diestal (Diestel, 2012), it defines graph as follows.

Definition 2.1. *A* **graph** *G is a pair of sets (V, E), denoted by $G(V, E)$, where V is a set of vertices and E is a set of edges satisfying $E \subseteq V \times V$.*

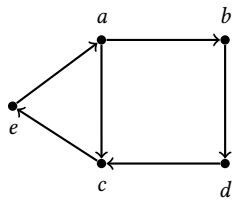
Following the above definition, real-world graphs such as social networks can be modeled by representing people, places, things, etc. as set of vertices V and a relationship (e.g. "friendship" between two persons) as an edge element in the edge set E. Analogously, biological datasets can be expressed as graphs by representing protein sequences as a set of vertices and their interactions as a set of edges. Similarly, knowledge graphs, which typically stores facts about real-world entities can naturally be expressed using graph data models.

In the rest of this section, we briefly discuss some of the generic graph concepts and properties that are used in further chapters; for the comprehensive background about graph theory, we refer the reader to the book (Diestel, 2012).



$V = \{a, b, c, d, e\}$
$E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{a, e\}, \{c, e\}\}$

(a)



$V = \{a, b, c, d, e\}$
$E = \{(a, b), (a, c), (b, d), (c, e), (d, c), (e, a)\}$

(b)

Figure 2.1: An example of (a) an undirected graph and (b) a directed graph

2.1.1.1 Directedness

It is not uncommon in many real-world datasets that the relationships are uni-directional. For instance, in biological datasets, it is often the case that a gene A "affects" gene B but not the *vice-versa*. To capture such information in graphs, edges are augmented with the direction. In our example, the relation between gene A and gene B is represented by a *directed edge* "A $\rightarrow$ B". On contrary, relationships such as "friendship", "marriedto" are bi-directional and are commonly represented by an *undirected edge* "A — B".

A graph is said to be *undirected* if it comprises of only undirected edges, and edge between a pair of vertices u and v is represented by an unordered set $\{u, v\}$. An example of an undirected graph is shown in Figure 2.1(a). On the other side, a graph is said to be *directed*, if all of the edges have direction associated with them. An edge in a directed graph is represented by an ordered pair (u, v) stating that the direction of the edge is from u to v. An example of a directed graph is shown in Figure 2.1(b).

By representing an edge set E as a set of ordered element pairs, i.e., $E = \{ (u, v) \mid u \in V \text{ and } v \in V \}$, an undirected and a directed graph, respectively, hold the following the properties.

$$(u, v) \in E \Leftrightarrow (v, u) \in E \qquad \text{(undirected)}$$
$$(u, v) \in E \not\Leftrightarrow (v, u) \in E \qquad \text{(directed)}$$

2.1.1.2 Subgraph

A *subgraph* G' of a graph G is defined as follows.

Definition 2.2. *A* **subgraph** $G'(V', E')$ *of a graph* $G(V, E)$ *is a graph such that the following holds:*

- $V' \subseteq V$, $E' \subseteq E$, *and*
- *for* $u, v \in V'$, $(u, v) \in E' \Rightarrow (u, v) \in E$.

Furthermore, a subgraph $G'(V', E')$ is called a *vertex-induced subgraph* of graph $G(V, E)$, if G' is a subgraph and it contains all of the edges in E that are incident

on vertices in V', i.e., for $u, v \in V'$, $(u, v) \in E' \Leftrightarrow (u, v) \in E$. On the other hand, $G'(V', E')$ is an *edge-induced subgraph* of graph $G(V, E)$, if G' is a subgraph and it comprises of only vertices that are incident on edges in E', i.e., for all $v \in V'$, either $(u, v) \in E'$ or $(v, u) \in E'$, for some u.

A subgraph, a vertex-induced, and an edge-induced subgraph for the example graph shown in Figure 2.1(b) are depicted in Figures. 2.2(a), 2.2(b), 2.2(c) respectively.

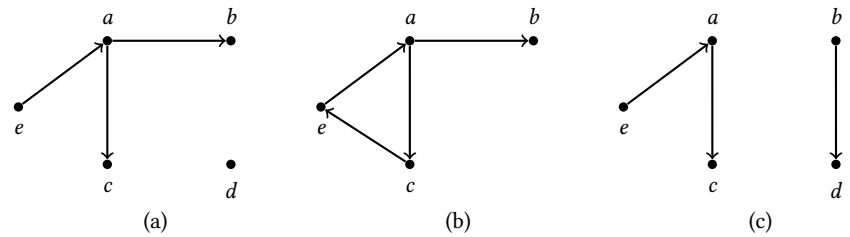


Figure 2.2: An example of (a) a subgraph, (b) a vertex-induced subgraph, and (c) an edge-induced sub-graph for example graph shown in Figure 2.1.

2.1.1.3 Graph Properties

Graph properties (or invariants) capture the important characteristics of a graph. These properties are significant in many applications like isomorphism, connectedness, to name a few. Graph properties can be either globally defined, i.e., with respect to an entire graph, or can be defined locally for each individual vertex or edge in a graph. In the following, we define some of the interesting and fundamental graph properties that are essential in the context of this thesis.

A. Graph Cardinality

- *Size.* For a graph $G(V, E)$, the *size* is the number of edges in G, denoted as $|E|$.
- *Order.* The *order* is the number of vertices in a graph $G(V, E)$, denoted as $|V|$.

Example 2.1. *The* size *and* order *of the example graph shown in Figure. 2.1 (or Figure. 2.1(b)) are five and six, respectively.*

B. Neighborhood

- *Successors.* Given a directed graph $G(V, E)$, successors set of a vertex $v \in V$, denoted by $succ(v)$, is the set of all immediate neighbors, u, such that $(v, u) \in E$, i.e., $succ(v) = \{ u \mid (v, u) \in E \}$.
- *Predecessors.* Analogously, predecessors set of a vertex $v \in V$ in a directed graph $G(V, E)$, denoted by $pred(v)$, is the set of vertices, u, such that $(u, v) \in E$, i.e., $pred(v) = \{ u \mid (u, v) \in E \}$.

- *Neighbors.* For a given vertex v in a directed graph $G(V, E)$, neighbors set of v, denoted by $neigh(v)$, is the *union* of sets $succ(v)$ and $pred(v)$, i.e., $neigh(v) = succ(v) \cup pred(v)$. In case $G(V, E)$ is undirected, $neigh(v) = \{ u \mid \{u, v\} \in E \}$.

Example 2.2. *For the example directed graph shown in Figure. 2.1(b), $succ(a) = \{b, c\}$, $pred(a) = \{e\}$, and $neigh(a) = \{b, c, e\}$.*

C. Degree

- *Out-degree.* The out-degree of a vertex v in a directed graph $G(V, E)$ is the size of the successors set. The out-degree of a vertex v is denoted by $deg^+(v)$ and defined as $deg^+(v) = |succ(v)|$. Specifically, a vertex v with zero out-degree, i.e., $deg^+(v) = 0$, is called a *sink* or *leaf.*

- *In-degree.* On the other side, the in-degree of a vertex $v \in V$ in a directed graph $G(V, E)$ is the size of the predecessors set. Denoted by the $deg^-(v)$, the in-degree for vertex v, is $deg^-(v) = |pred(v)|$. A vertex v with zero in-degree, i.e., $deg^-(v) = 0$ is called a *source* or *root.*

- *Degree.* Degree (or valency) of a vertex $v \in V$ in a graph $G(V, E)$ is the number of edges incident on v. The degree of a vertex v, in a directed graph, is computed as $deg(v) = deg^+(v) + deg^-(v)$, or simply $deg(v) = |neigh(v)|$.

  Specifically, a vertex v with $deg(v) = 0$, is called an *isolated vertex.* Notationally, the maximum and minimum degree of all vertices in a graph G is represented by $\Delta(G)$ and $\delta(G)$, respectively.

Example 2.3. *For the example graph shown in Figure. 2.1(b), we have $deg^+(a) = 2$, $deg^-(a) = 1$, and $deg(a) = 3$ for vertex 'a'. It can also be observed that the minimum degree $\delta(G) = 2$ (for 'b','d','e') and the maximum degree $\Delta(G) = 3$ (for 'a','c').*

D. Connectivity

- *Connected graph.* A graph $G(V, E)$ is said to be connected if there exists a *path* between every unordered pair of vertices in V. A path (defined later in this section) between a pair of vertices u, v is a sequence of distinct vertices that lie in-between u and v. Else, $G(V, E)$ is a *disconnected* graph.

- *Vertex Connectivity.* For a given connected graph $G(V, E)$, vertex connectivity is the smallest set of vertices, X, called *cut-vertices*, whose removal makes the resultant vertex-induced subgraph a disconnected graph. Vertex connectivity of a graph G is denoted by $\kappa(G)$ and is equal to $|X|$, such that

$$\arg\min_{X \subseteq V} \; G'(V \backslash X, E') \text{ is a disconnected induced subgraph.}$$

- *Edge Connectivity.* On the other hand, edge connectivity of a graph is the smallest set of edges whose removal makes the resultant subgraph a disconnected

graph. Edge connectivity of a graph G is denoted by $\lambda(G)$ and is equal to $|Y|$ where set Y is computed as

$$\arg\min_{Y \subseteq E} \; G'(V, E \backslash Y) \text{ is a disconnected subgraph.}$$

Edge connectivity is also referred to as *min-cut*; the corresponding edge set and the resultant subgraph are called *cut-edges* and *cut-graph* (or simply *Cut*), respectively.

- *Diameter.* For a given connected graph, the diameter refers to the longest of all the shortest paths between any pair of vertices.

Example 2.4. *In our running example, graph G shown in Figure 2.1(b) is a connected graph, has vertex connectivity $\kappa(G) = 2$ with cut-vertices = $\{a, c\}$ or $\{a, d\}$ or $\{b, c\}$ and the edge connectivity $\lambda(G) = 2$ with cut-edges = $\{(e, a), (c, e)\}$ or $\{(a, b), (b, d)\}$ or $\{(d, c), (b, d)\}$. It can also be observed that the diameter for the graph in Figure 2.1(b) is three (via a path from vertices 'e' to 'd').*

2.1.1.4 Connectedness

In the following, we discuss some of the graph connectedness concepts which aid in understanding of the topology of a graph $G(V, E)$.

- *Path.* Given pair of vertices $s, t \in V$, a path, denoted by $P_{s,t}$ (or simply P for brevity), is a finite sequence of edges (or vertices) between s and t. Path P, thus, an ordered set of edges is written as

  $$P = ((s := u_0, u_1), (u_1, u_2), \ldots, (u_{n-1}, u_n =: t)),$$

  or simply a vertex sequence

  $$P = (s := u_0, u_1, u_2, \ldots, u_{n-1}, u_n =: t),$$

  where for $i = 1$ to n, $(u_{i-1}, u_i) \in E$. With vertex set $V_p = \{u_0, \ldots, u_n\}$ and edge set $E_p = \{(s := u_0, u_1), (u_1, u_2), \ldots, (u_{n-1}, u_n =: t)\}$, a path $P(V_p, E_p)$, thus, can be alternatively thought as a subgraph of G.

  A path is *simple* if none of the vertices are repeated. We define *path length* as the number of edges in a path P; a path length of size l is denoted by P^l. For the case when $P = \emptyset$, the path length is defined as *infinity* (P^∞). Unless otherwise stated, path length always refers to the length of simple path.

  Abusing the notation and representing path as a set, we call two paths P_1 and P_2, for a pair of vertices (s, t), *distinct* if $P_1 \not\subset P_2$ or $P_2 \not\subset P_1$. Paths P_1 and P_2 are *edge-disjoint*, if P_1, P_2 do not share any edge in common, i.e., $P_1 \cap P_2 = \emptyset$. Likewise, two paths P_1 and P_2 are said to be *vertex-disjoint* if P_1, P_2 do not share any vertex. It is trivial to state that vertex-disjoint paths are also edge-disjoint.

A *path set*, denoted by $\mathcal{P}$, is the set of all distinct paths from s to t, i.e.,

$$\mathcal{P} = \{P_i \mid P_i \text{ is a path from } s \text{ to } t\}.$$

A path P is called *shortest path* if P is simple and the path length P^l is the smallest of (or equal to) all the paths in $\mathcal{P}$.

- *Cycle.* A cycle is a finite sequence of edges (or vertices) such that the removal of an edge (t, s) from the sequence translates into a path P from s to t. A cycle C, thus, can be written as an edge sequence as follows.

  $$C = ((s := u_0, u_1), (u_1, u_2), \ldots, (u_{n-1}, u_n =: t), (t, s)),$$

  or as vertex sequence,

  $$C = (s := u_0, u_1, u_2, \ldots, u_{n-1}, u_n =: t, s).$$

  A cycle C is *simple* if the edges (or vertices) are not repeated. We define *cycle length*, denoted by C^l, as the number of edges (or vertices) in C. The minimum cycle length of a graph G is denoted by $g(G)$, while a maximum cycle length is called *circumference* of G.

- *Reachability.* Given a pair of vertices s, t and a graph $G(V, E)$, we call s and t *reachable*, if there exists at least one path from s to t. Denoted by $s \rightsquigarrow t$, reachability is a Boolean value expressed as follows.

  $$s \rightsquigarrow t = \begin{cases} \texttt{true} & \text{if } \mathcal{P} \neq \emptyset \\ \texttt{false} & \text{otherwise} \end{cases}$$

  Reachability for an undirected graph is symmetric, i.e., $s \rightsquigarrow t \Leftrightarrow t \rightsquigarrow s$, for $s \neq t$. While for a directed graph, reachability is not symmetric. Moreover, if it holds that $s \rightsquigarrow t$ and $t \rightsquigarrow s$ in a directed graph, then there exists a cycle passing through the vertices s and t. Reachability analysis, thus, can be helpful in cycle detection, specifically, in directed graphs, and connected components (which we define shortly) in both directed and undirected graphs.

- *Connected component.* A connected component of a graph G is a subgraph such that for any pair of vertices s, t in the component, if it holds that $s \rightsquigarrow t$ is `true` and/or $t \rightsquigarrow s$ is `true`.

  In a directed graph, a connected component is called *strongly connected component* (SCC), if for any two pairs of vertices s, t in the component, it holds that $s \rightsquigarrow t \Leftrightarrow t \rightsquigarrow s$. Else, a connected component is referred to as a *weakly connected component* (WCC). A WCC, thus, holds either $s \rightsquigarrow t$ or $t \rightsquigarrow s$, but not always both.

  A connected component is *maximal*, if no other vertex can be added to the component without breaking its properties.

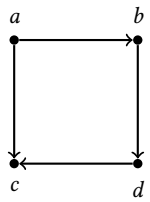
Example 2.5. *Considering the example graph G shown in Figure. 2.1(b), we observe the following.*

- *The pair of vertices e, c is reachable, i.e.,* $e \rightsquigarrow c$ *is* `true` *as the path set* $\mathcal{P} = \{(e, a, c), (e, a, b, d, c)\} \neq \emptyset$.
- *The sequences* (e, a, c, a) *and* (e, a, b, d, c, e) *form two cycles in G.*
- *The vertex-induced subgraphs with vertex sets* $\{a, c, e\}$, $\{a, b, c, d, e\}$ *are SCCs of G, while the vertex-induced subgraph with vertex sets* $\{a, b, c, d\}$ *is a WCC of G.*

2.1.1.5 Special Graphs

Next, we discuss some special graph types which are prominently used in many graph applications and also used in our setting.

- *Directed Acyclic Graph (DAG).* A graph $G(V, E)$ is a directed and acyclic graph if G contains no cycles. That is, for any ordered pair of vertices (u, v), $u, v \in V$, for $u \neq v$, its corresponding path set $\mathcal{P}$ should be non-empty and contains only simple paths. For instance, consider an induced subgraph $G'(V', E')$ (shown below) with vertex set $V' = \{a, b, c, d\}$ of the example graph shown in Figure. 2.1(b).



  G' is a DAG as it contains no (directed) cycles, this can be verified with fact that for the ordered pair of vertices (c, a), its path set $\mathcal{P} = \emptyset$. A directed graph $G(V, E)$ can be transformed into a *unique* directed acyclic graph by applying graph *condensation* operation (which will be introduced shortly).

- *Trees and Forests.* A graph $T(V, E)$ is a *tree* if T has no cycles, and for any pair $u, v \in V$, there exists at most one path from u to v. In other words, the size of the path set $|\mathcal{P}| \leq 1$ for any pair of vertices in T. An example of a tree is shown in Figure. 2.2(a).

  A vertex v in a directed tree is called a *root* if $pred(v) = \emptyset$, and is called a *leaf* if $succ(v) = \emptyset$.

  A forest $F(V, E)$ is (disconnected) graph comprising of one or more trees. An example of a forest is shown in Figure. 2.2(c).

2.1.1.6 Graph Operations

Next, we discuss some of the basic graph operations that are commonly performed and also used in the context of this thesis.

- *Updates.* Graph update operations mainly constitute operations that change the topology of the graph. These include insertion and deletion of new vertices and/or edges. On a graph $G(V, E)$, we define the following update operations.

  - Insert(v): updates graph $G(V, E)$ by adding the vertex v. The resultant graph is $G(V \cup \{v\}, E)$.
  - Delete(v): deletes the vertex v from G and further deletes all the edges that are incident on v. The updated graph is $G'(V', E')$, where $V' = V \backslash \{v\}$ and $E' = E \backslash \{ \{(u, v) \mid (u, v) \in E\} \cup \{(v, u) \mid (v, u) \in E\} \}$.
  - Insert(u, v): adds an edge between the vertex u and v. If v (or u) is a new vertex, Insert(v) (or Insert(u)) is called first.
  - Delete(u, v): deletes an edge between the vertex u and vertex v.

- *Closure.* A closure (or transitive closure) of G is the smallest super graph $G^+(V, E^+)$ such that

  - $E \subseteq E^+$
  - $\forall (u, v) \in E^+$, $u \rightsquigarrow v$ is `true` in G.

  A closure $G^+(V, E^+)$ can be obtained by adding an (transitive) edge between two vertices u, v, if there exists a path from u to v. The size of a closure graph G^+, i.e., $|E^+|$ satisfies the inequality $|E| \leq |E^+| \leq (|V| \cdot (|V| - 1))$.

  As the closure encompasses all the reachability information, a reachability check for a pair of vertices can be performed in $\mathcal{O}(1)$ time complexity. Thus computing and compactly representing a closure is often used as a preprocessing step in many reachability evaluation systems (Yildirim et al., 2010; Seufert et al., 2013).

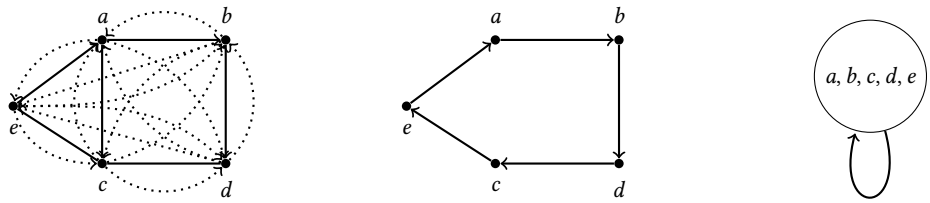
- *Reduction.* On contrary, a reduction (or transitive reduction) of a graph $G(V, E)$ is the minimal graph $G^r(V, E^r)$ obtained by iteratively adding/removing edges, such that addition/removal of an edge (u, v) does not alter the reachability of $u \rightsquigarrow v$. That is,

  - $u \rightsquigarrow v$ is `true` in $G \Leftrightarrow u \rightsquigarrow v$ is `true` in G^r,
  - $(u, v) \in E \not\Rightarrow (u, v) \in E^r$ and $(u, v) \in E^r \not\Rightarrow (u, v) \in E$.

  Unlike closure, reduction may not always generate a unique minimal graph, and also it may be the case that G^r is not a subgraph of G. If G is a directed acyclic graph (DAG) (which we define later in this section), G^r is always unique.

- *Condensation.* Graph condensation is an operation of transforming a graph $G(V, E)$ into a directed acyclic graph $G_c(V_c, E_c)$ such that

  - V_c is the set of non-overlapping maximal SCCs in G;



(a) Closure (b) Reduction (c) Condensation

Figure 2.3: An example of (a) closure, (b) reduction, and (c) condensation for graph shown in Figure. 2.1(b)

– if an edge $(u, v) \in E$ then $(u_c, v_c) \in E_c$ iff $u \in u_c$, $v \in v_c$, and $u_c \neq v_c$, where u_c and v_c are represent SCC for vertices u and v, respectively.

Example 2.6. *Consider the directed graph G shown in Figure 2.1(b). Th closure of G is shown in Figure 2.3(a), where dotted edges denote the closure edges, The reduction of G is shown in Figure. 2.3(b), where the edge* (a, c) *is removed as* $a \rightsquigarrow c$ *is still* `true` *after its removal. Figure. 2.3(c) shows the condensation of the graph G (an SCC by itself), resulting in a single-vertex condensed graph.*

2.1.1.7 Graph Representations

Depending on the applications in use, graphs are often represented in the following popular forms.

- *Edge List.* A simple way of representing graphs is to enumerate all edges as element pairs $\langle u, v\rangle$ in a graph, where each pair denotes a directed edge between vertex u and v. In case of undirected graphs, each edge is enumerated twice, i.e., $\langle u, v\rangle$ and $\langle v, u\rangle$

  Additional metadata, like edge weights, vertex and edge labels can be associated with each edge. N-triples (Carothers, 2014) used for representing knowledge graphs follows an edge list representation.

- *Adjacency List.* Adjacency list is another widely used graph representation format that is predominantly used for representing large graphs. In adjacency list, a graph is typically represented as a key-value pair $\langle K, V\rangle$, where key K denotes the vertex and value V is the list of neighbors (in case of undirected graph) or only successors (in case of directed graphs).

  Turtle or N3 format (Beckett et al., 2014) used for representing RDF data is an example of adjacency list format.

- *Adjacency Matrix.* Adjacency matrix is conceptually similar to an adjacency list, where in, a graph is represented as a matrix of rows and columns denoting vertices. An element (i, j) in the matrix represents an edge between vertex i (row) and vertex j (column).

For sparse graphs, this representation leads to a lot of wasted space, thus an adjacency list is usually preferred over adjacency matrix. However, adjacency matrices are widely used representation when dealing with theoretical aspects of graphs.

- *Incidence Matrix.* An adjacency matrix representation, where columns are represented as edges instead of vertices, is referred to as an incidence matrix. An element (i, j) in an incidence matrix denotes the information that vertex i is *incident* on j^{th} edge, i.e., (i, v) for some vertex v. Incidence matrices are often used in representing multi-graphs where more than one edge exists between a pair of vertices.

Example 2.7. *Various representation of the example graph shown in Figure. 2.1(b) is shown below.*

Edge List
1:(a,b)
2:(a,c)
3:(b,d)
4:(c,e)
5:(d,c)
6:(e,a)

Adjacency List
$a \rightarrow b,c$
$b \rightarrow d$
$c \rightarrow e$
$d \rightarrow c$
$e \rightarrow a$

Adjacency Matrix

	a	b	c	d	e
a	0	1	1	0	1
b	0	0	0	1	0
c	0	0	0	0	1
d	0	0	1	0	0
e	1	0	0	0	0

Incidence Matrix

	1	2	3	4	5	6
a	1	1	0	0	0	0
b	0	0	1	0	0	0
c	0	0	0	1	0	0
d	0	0	0	0	1	0
e	0	0	0	0	0	1

2.1.2 Relational Databases

Besides graph model, we rely on concepts from relational databases for managing and querying graphs. A relational database is one of *the age old* data management techniques that is built on the concept of *relations* (or *tables*), which capture the relationships present in structured data. An enormous amount of research and development went in for many decades in developing relational databases. In this section, we briefly cover some key concepts which are useful in our problem setting.

2.1.2.1 Relational Model

In his seminal work on relational databases, Codd et al. (Codd, 1983) proposed the *relational model* which formed the foundation for relational databases. In the *relational model*, a *relation* forms a basic unit that captures two-dimensional structured data. Also referred to as a *table*, a relation R is a pair of sets A and I where, A is a set of attributes and I is a finite set of instances for relation R. Specifically, each instance, also called as *tuple*, is a set of constants representing the values for attributes (A) in R. The values usually belong to the domain of attributes which explicitly is defined along with the *schema* for R.

The *schema* of a relation represents the name (of a relation) along with its attributes. For example shown in Figure. 2.4, "*Students*(*Name*, *Course*)" denotes the schema for a relation with name "*Students*" and attributes "*Name,Course*". The

Students

Name	*Course*
Alice	Database Systems
Bob	Graph Theory
Charlie	Database Systems
Dan	Algorithms

Courses

Course	*Tutor*
Algorithms	T. H. Cormen
Database Systems	J. D. Ullman
Graph Theory	R. Diestal

Figure 2.4: An example of relational model representing Student-Course information

Primary Key (PK) of a relation is either an attribute or a set of attributes, where no duplicate instance values are allowed. For example, attribute *Name* can be designated as a PK for relation *Students*. Moreover, an attribute which is a PK in one relation is referred to as *Foreign Key (FK)*, if it appears, in another relation. For instance, attribute *Course* is a PK in relation *Courses* and a FK in relation *Students*.

An example of relational model capturing the student-course information is depicted in Figure. 2.4. Here the columns {*Name*, *Course*} forms the attributes for *Students* relation, while each tuple (row), say (Alice, Database Systems), is an instance of attributes *Name* and *Course*, respectively.

2.1.2.2 Relational Algebra

While relational model deals with the structural aspect of the relational databases, all the data manipulation and querying on relational databases are dealt with *relational algebra*, also proposed by Codd et al. (Codd, 1983). According to the book (Garcia-Molina et al., 2008), relational algebra, like any algebra, comprises of variables, constants, and operators defined with respect to *relations*.

A relational algebra mainly constitutes *relations* as variables and constants and *selection*, *projection*, *cross product*, *join* as well as the set algebra operators – *union*, *intersection*, and *difference* – as the main algebraic operators. Next, we discuss some key relational algebraic constructs. For a comprehensive background about relational algebra, we refer the reader to (Garcia-Molina et al., 2008; Date and Darwen, 1997; Ramakrishnan and Gehrke, 2003)

A *selection* operation (σ) is a unary operation defined over a relation. Syntactically denoted by the expression $S = \sigma_C(R)$, selects all tuples from relation R that satisfies the condition C (a Boolean expression) and writes them to relation S. For instance, the query "*Find all students who took* Database systems *course*" can be casted into a selection expression, $Result = \left(\sigma_{course=\text{"Database Systems"}}\ (Students)\right)$. Evaluation of the example selection statement returns the following *Result* relation instance.

Result

Name	*Course*
Alice	Database Systems
Charlie	Database Systems

While the selection operation works on the rows of a relation, a *projection* operation (π) is a unary operator that selects the desired attributes from a relation. Denoted algebraically by the expression $S = \pi_P(R)$, the projection creates a relation S with the attributes $P \subseteq A$ along with the corresponding tuples from the relation R. For instance, the query "*Find all names of students*" can be expressed using a projection operation as $Result = \pi_{Name}(Students)$. Below table shows an instance of *Results* relation containing the names of students from relation *Student*.

Result

Name
Alice
Bob
Charlie
Dan

While selection and projection are unary operators, a *Cartesian-product* ($\times$) takes a pair of input relations R, S and returns a relation with attributes set comprising of attributes from the both relations. The tuples of the output relation are all the pairs formed from the tuples of both the input relations. A Cartesian-product over relations R, S is denoted by $R \times S$, and the number of tuples (i.e., *cardinality*) of the Cartesian product $|R \times S|$ is $|R| \cdot |S|$. To exemplify, Cartesian-product on our example relations *Students*, *Courses*, i.e., (*Students* $\times$ *Courses*), results in the following output relation.

Name	*Course*	*Course*	*Tutor*
Alice	Database Systems	Database Systems	J. D. Ullman
Alice	Database Systems	Graph Theory	R. Diestal
⋮	⋮	⋮	⋮
Bob	Graph Theory	Database Systems	J. D. Ullman
Bob	Graph Theory	Graph Theory	R. Diestal
⋮	⋮	⋮	⋮

Theta-joins ($\underset{\theta}{\bowtie}$), on the other hand, extend the Cartesian-product operation by imposing a condition θ. A theta-join over a pair of input relations R, S is denoted by the expression

$$T = R \underset{\theta}{\bowtie} S.$$

The evaluation of expression creates an output relation T similar to Cartesian-product, while only the tuples that satisfy the condition θ, form the tuples for relation T. Note that, a *theta-join* can always be written using a combination of operators selection (σ_θ) and Cartesian-product ($\times$). The above example theta-join expression can be equivalently written as $T = \sigma_\theta(R \times S)$.

Another binary operator and more commonly used in practice is *join* ($\bowtie$). Analogous to Cartesian-product and theta-joins, join takes a pair of relations as

```
SELECT Names
FROM Students, Courses
WHERE Courses.tutor ="J. D. Ullman"
```

Figure 2.5: An example SQL query

an input and outputs a new relation. Unlike Cartesian-product and theta-joins, the attribute set of the output relation is the *union* of the attribute sets of an input pair of relations. While tuple instances of the output relation depends on the type of join. A more commonly used type of *join* is *natural join*. In natural join, a tuple belongs to an output relation only if the combined tuple from the input relations agree on the common attributes. Consider two input relation schemas $R(A, B)$ and $S(B, C)$, denoted by $T = R \bowtie S$, a natural join operation outputs relation with schema $T(A, B, C)$. A tuple $t = (a, b, c)$ is an instance of T if there exists tuples $r = (a, b)$ and $s = (b, c)$ such that r, s are tuples of relations R, S respectively. For our running example, the natural join operation (*Students* $\bowtie$ *Courses*) results in following relation instance.

Students $\bowtie$ *Courses*

Name	*Course*	*Tutor*
Alice	Database Systems	J. D. Ullman
Bob	Graph Theory	R. Diestal
Charlie	Database Systems	J. D. Ullman
Dan	Algorithms	T. H. Cormen

Analogously to set algebra, relational algebra supports other binary operators such as *union*, *intersection*, and *difference* over relations with identical schemas. A union operation, denoted by $R \cup S$, returns a relation comprising of tuples from input relations R and S. While intersection, denoted by $R \cap S$, returns a relation with tuples that are common to both relations R and S. On the other hand, difference, $R - S$, returns a relation comprising of tuples that are in R but not in S.

2.1.2.3 Structured Query Language (SQL)

SQL is an industry standard declarative language designed for managing and querying relational database management systems. Although not completely adhering to the relational algebra proposed by E. F. Codd (Codd, 1983), SQL became widely popular and is the de facto language used in commercial relational database systems (RDBMS). SQL provides several constructs such as `CREATE`, `ALTER`, `DELETE`, `DROP`, and `RENAME` as part of data definition language (DDL) for manipulating the schema of a relation, and constructs `SELECT`, `INSERT`, `DELETE`, `UPDATE`, and `DROP` as part of data manipulation language (DML) to manipulate the tuples of a relation in a relational database. Covering the entire SQL syntax and semantics is beyond the scope of this thesis; we refer the interested reader to (Garcia-Molina

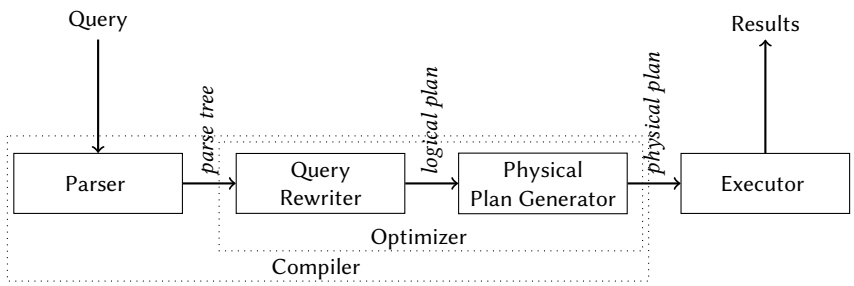


Figure 2.6: A typical relational query processing workflow

et al., 2008; Date and Darwen, 1997; Ramakrishnan and Gehrke, 2003). Here, we briefly discuss the syntax of a simple SQL query to get a glimpse of the language. An English language query "*Find the names of all students who took the course taught by J. D. Ullman*" can be expressed in SQL (using the relations from Figure. 2.4) as follows.

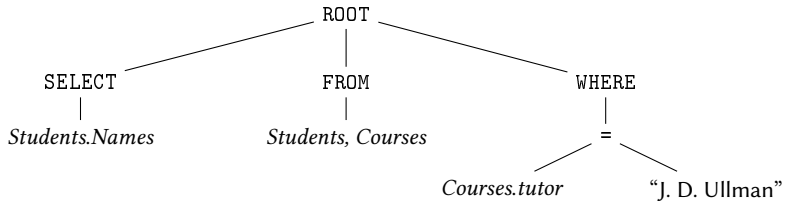
The above query comprises of three clauses `SELECT`, `FROM`, `WHERE`. The clause `SELECT` takes a list of columns as a parameter and is similar to projection operation (π) from relational algebra. While `FROM` clause takes a list of input tables, and the `WHERE` clause is closely related to semantics of the selection operation (σ) in relational algebra that take a Boolean expression (*Course.tutor* = "J. D. Ullman"). The above SQL query can equivalently be expressed in relational algebra as,

$$\pi_{Names}(\sigma_{(Course.tutor=\text{"J. D. Ullman"})}(Student \bowtie Courses)).$$

2.1.2.4 Query Processing

Processing a relational query, such as the one expressed in SQL, involves a series of steps, such as *parsing*, *query rewriting*, *physical plan generation*, and *query execution*. The first three steps come under the purview of the *query compiler* module of a relational system; the two steps of query rewriting and physical plan generation are part of *query optimizer*, a sub-module of the *query compiler*. The last step is performed by *query executor* module. Here, we briefly discuss the salient points in relational query processing and we refer the reader to (Garcia-Molina et al., 2008; Date and Darwen, 1997; Ramakrishnan and Gehrke, 2003) for a comprehensive overview. A pictorial overview of a typical relational query processor is shown in Figure. 2.6

- *Parsing*. The first step in a relational query processor is to parse the input SQL query and translate it into a *parse tree*. A parse tree is a tree (see Section. 2.1.1.5) comprising of a set of operators (forming internal nodes) and a set of constants (forming leaf nodes). SQL constructs such as `SELECT`, `FROM`, and `WHERE` form the operators, while the input relations, literals in the query form the constants. For the example query shown in Figure. 2.5, a parse tree can be constructed as follows.



- *Query Rewriting.* A parse tree is often represented in algebraic form on which several query rewriting principles are applied. For instance, the above parse tree can be expressed in algebraic form as follows.

$$\pi_{Students.Names}\ \sigma_{Courses.tutor=\text{"J. D. Ullman"}}(Students \bowtie Courses)$$

Query rewriting principles are a set of relational algebraic rules that can be applied such that only the syntactic representation is altered but not the semantics of the query. Such rewriting principles are commonly used to make a plan that can be executed efficiently. Some of the rewriting principles include *selection, projection, aggregation rules*, and *join orderings.* A commonly applied selection rule is *push-down* selections. To exemplify, consider a selection expression $Q = \sigma_C(R \bowtie S)$, if C is a set of constraints that applies to only relation R's attributes, then a push-down selection rule can be applied to rewrite the expression in to an equivalent expression $Q = (\sigma_C(R) \bowtie S)$. The benefit of such a rewrite is that the join operation can be made faster as the cardinality of relation R is reduced to only tuples that satisfy C. On the other hand, if C is a set of conjunctive constraints involving attributes from both R and S, then the query Q can be equivalently written as $\sigma_{C'}(R) \bowtie \sigma_{C''}(S)$ where C' is a set of constraints on R's attributes and C'' conditions on S's attributes. By applying push-down selections, the example query can be rewritten into a semantically equivalent form as,

$$\pi_{Students.Names}(Students \bowtie \sigma_{Courses.tutor=\text{"J. D. Ullman"}}(Courses)).$$

Push-down rules can be also applicable to other operators like projections (π) and aggregations (`min`, `max`, `avg`, etc.). More details about rewritings for these operators can be found in (Garcia-Molina et al., 2008; Date and Darwen, 1997; Ramakrishnan and Gehrke, 2003).

Joins are one of the most expensive operators in a query plan. Join orderings are, thus, an important class of query rewriting principles, which rely on the commutative and associative properties (shown below) of the join operators.

$$(R \bowtie S) \equiv (S \bowtie R) \qquad \text{(Commutative)}$$
$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \qquad \text{(Associative)}$$

By applying these properties, a query plan can be rewritten into multiple equivalent forms. A join ordering is said to be "optimal", for a query, if the overall

time taken to execute the query is less than or equal to that of all other possible join orderings. Enumerating all possible join ordering and find an optimal one is considered to be an NP-hard problem with the number of possible join orderings equal to the Catalan number, i.e.,

$$C_n = \frac{1}{1+n} \cdot \binom{2n}{n}$$

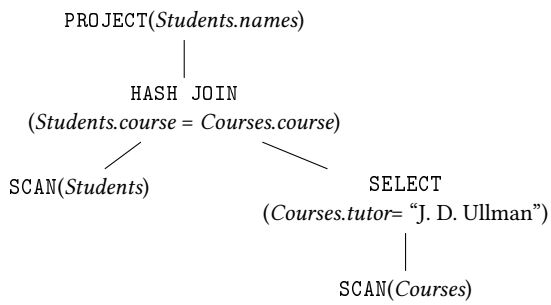
for a query with n join operators. Most query optimizers, thus, rely on either heuristics or cost-based optimizations to compute a low-cost join ordering which might not be always be optimal. As estimating the cost of an ordering is dependent on the implementation of individual operators, finding a good join ordering is often performed while generating a physical plan for the query.

- *Physical Plan Generation.* A *logical plan* generated as a result of several query rewriting rules is translated into a *physical plan* by replacing each logical algebraic operator into a physical operator. The choice of replacements are usually many and is often defined by the underlying system. For instance, a natural join operator can be replaced with physical operators like `HASH-JOIN`, `MERGE-JOIN`, or even `NESTED-LOOPS`. Each choice can have its own cost which is dependent on the physical implementation. While translating a logical query into a physical representation, query optimizers often rely on *cost-based optimizations* to generate a best possible physical plan.

  A typical cost-based query optimizer computes a best possible physical plan, for a specific system setting, by relying on the estimated cost of the individual operators. Then a bottom-up dynamic programming based algorithm is used to exhaustively search for a minimal cost plan by relying on the following.

  1. Cost of scanning/sorting input relations
  2. Cost of unary operations (selections, projections, aggregations)
  3. Cost of binary operations (joins, set operations)
  4. Alternative join orderings
  5. Execution of operators – pipelining or materialization

  An example of a physical plan for our running example query is shown below.



- *Query Execution.* The physical plan (a.k.a *operator tree*) is executed bottom up starting with the scan of the left most input relations. Depending on the way operators share information, a query can be executed either in *pipelined* or *materialized* model.
  - In a pipelined model, multiple (possible) operators are executed simultaneously. Each operator gets a single tuple or a block of tuples as input, processes the tuples and the result is passed on to the higher level operators before working on next block of tuples. Operators like `SELECT`, `MERGE-JOIN`, `PROJECT` are the best fit for a pipelined model of execution, as they can be processed in a streaming fashion. Operators like `HASH-JOIN` and aggregate operators like `COUNT`, `MIN`, `MAX`, `AVG` are pipeline breakers, as they require looking at all tuples before emitting any output tuples.
  - On the other hand, in *materialized* model, operators are sequentially executed, and each operator materializes its output into a temporary relation, called an *intermediate relation.* The intermediate relation is then passed on to the higher level operators. Unlike pipelined model, the materialized model needs to completely process all the lower-level operators before emitting any single result tuple.

2.1.3 Graph Data Management

In this section, we provide a brief background about the graph data and query models that are commonly found in practice. This section also serves to introduce some of the state-of-the-art graph database systems, which we either build on or compare with our proposed ideas.

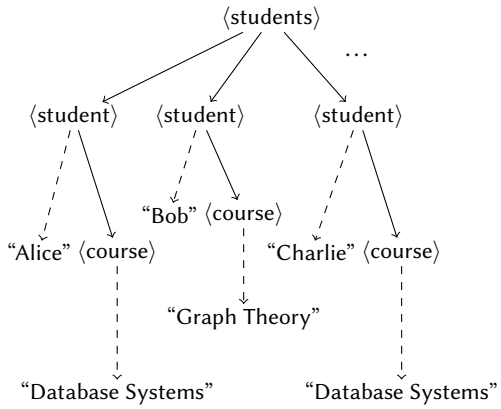
2.1.3.1 Graph Data Models

Graph data models are intrinsic to many application domains. Here, we briefly discuss some of the most commonly used graph data models. A comprehensive overview of the graph database models can be found in (Angles and Gutierrez, 2008).

- *Tree Structured Model.* Tree structured data models occupy a special place in the list of graph data models. Often used to represent semi-structured data, a tree structured data model is a special form of directed, acyclic graph data model (see Section. 2.1.1.5). Extensible Markup Language (XML) is a well known data model that adopts tree structured data model. XML was proposed by W3C as a data format for exchanging information across applications. A typical XML document comprises of *elements*, *attributes* and *content* (character data). An element is a logical section of the document with matching start and end tags, and comprises of zero or more attributes that collectively define the properties of an element. Elements may encompass additional elements or textual content. An example snippet of the XML document is shown in Figure 2.7(a),

```
⟨students⟩
  ⟨student name="Alice"⟩
    ⟨course name = "Database Systems"/⟩
    ⟨/course⟩
  ⟨/student⟩
  ⟨student name="Bob"⟩
    ⟨course name = "Graph Theory"/⟩
  ⟨/student⟩
  ⟨student name="Charlie"⟩
    ⟨course name = "Database Systems"/⟩
  ⟨/student⟩
  ⟨student name="Dan"⟩
    ⟨course name = "Algorithms"/⟩
  ⟨/student⟩
⟨/students⟩
```

(a) XML document



(b) XML tree

Figure 2.7: An example XML document (a) and its tree representation (b)

and its corresponding tree representation is shown in Figure 2.7(b). Here the tags ⟨students⟩,⟨student⟩,⟨course⟩ are called elements, and for a ⟨student⟩ or ⟨course⟩ element, name is an attribute. Elements and attributes are represented as vertices in an XML tree and an edge set constitutes union of (solid) edges between two elements and (dashed) edges between an element and a attribute. For more details about the XML syntax and semantics, we refer the reader to (Bray et al., 2008).

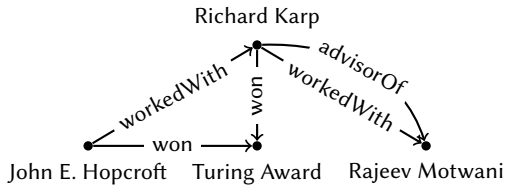
XLink (DeRose et al., 2010) proposed by W3C is an XML markup language to create links (both internal and external) among XML documents, thus extending the tree model of XML into a graph model.

- *Directed Labeled Multi-graph Model.* In this directed graph model, vertices and edges are labeled, and between a pair of vertices there exists more than one edge (all with distinct labels). Real world instances of this graph data model include knowledge graphs, biological datasets, etc. Resource Description Format (RDF) (Klyne and Carroll, 2004), a W3C recommended and one of the most widely used representation format in semantic web community, follows the directed labeled multi-graph model. A typical RDF document comprises of a collection of RDF statements, called *triples*. Each triple is of the form ⟨*Subject*, *Predicate*, *Object*⟩, where *Subject* is a unique web resource or URI, *Object* can be either a unique web resource or a literal (a textual string), and *Predicate* denotes the relation (property) between a given *Subject* and *Object*. An example of an RDF data snippet and its graph representation is shown in Figure 2.8.

  Some large real-world instances include knowledge graphs such as Google Knowledge Graph (Singhal, 2012), Freebase (Bollacker et al., 2008), YAGO (Suchanek et al., 2007), DBpedia (Bizer et al., 2009), biological datasets like UniProt (The UniProt Consortium, 2014), Bio2RDF (Belleau et al., 2008), and others like LinkedMDB (Hassanzadeh and Consens, 2009), SwetoDblp (Aleman-Meza et al., 2007).

⟨John E. Hopcroft, won, Turing Award⟩
⟨John E. Hopcroft, workedWith, Richard Karp⟩
⟨Richard Karp, won, Turing Award⟩
⟨Richard Karp, workedWith, Rajeev Motwani⟩
⟨Richard Karp, advisorOf, Rajeev Motwani⟩

(a) RDF data



(b) RDF graph

Figure 2.8: An example of (a) RDF data and its (b) graph representation

- *Directed Labeled Attributed Multi-graph Model.* This type of graph data model extends the directed labeled multi-graph model by associating attributes to both the vertices and edges. An attribute is a key-value pair that uniquely defines the properties of a vertex or an edge. An instance of this graph data model is the *property graph* used in the popular Neo4j (Neo4j, 2012) graph database. Many real-world graphs like social networks, citation networks, sensor networks can be effectively represented using this model.

  To exemplify more about this data model, consider a social network, say Facebook[1], where the vertex set comprises of people, places, businesses, etc. and the edge set is a set of relationships among vertices. Furthermore, vertices of type person have attributes like Name, Age, Gender, Birthdate, to name a few. An edge between two persons can have attributes like Type (denoting the type of relation), Timestamp (link creation time), etc. Such data is best represented using directed, labeled attributed multi-graphs (or property graphs). Figure. 2.9 shows an example of social network excerpt represented in this model. It contains three vertices with labels Barack Obama, Hillary Clinton, and Michelle Obama. Each vertex has attributes such as Birthplace, Birthdate denoted in a dotted box. Relations between the vertices are labeled, and furthermore, relations like Spouse, workedFor, have attributes like Since, From, To, etc.

2.1.3.2 Graph Query Models

In this section, we briefly cover some of the important graph query models and also point to few relevant query languages that support them. An exhaustive discussion of query languages and query models can be found in (Angles and Gutierrez, 2008; Wood, 2012).

- *Connectivity Queries.* These queries form an important class that are fundamental to many graph applications. Queries that deal with the connectivity such as *reachability*, *shortest path queries*, fall under the category of *connectivity queries*.

  Given a graph $G(V, E)$ and two vertices $s, t \in V$, *reachability* checks whether there exists at least one path between s and t in G (see Section 2.1.1.4). This

[1] http://facebook.com

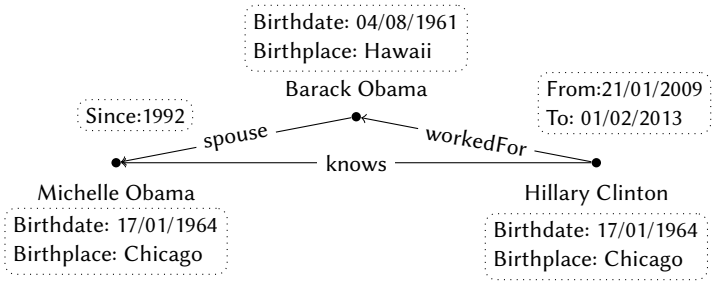


Figure 2.9: An example of a property graph

simple notion of reachability has been comprehensively studied in past resulting in many approaches that can scale well to large graphs. A few of there are (Agrawal et al., 1989; Jagadish, 1990; Cohen et al., 2003; Cheng et al., 2006; Wang et al., 2006; Chen and Chen, 2008; Jin et al., 2009; Yildirim et al., 2010; Schaik and Moor, 2011; Jin et al., 2011; Seufert et al., 2013; Yu and Cheng, 2010).

Often, in practice, a reachability query is associated with regular expression (RE) defined over the vertex and edge labels. Called *regular reachability* queries (Abiteboul et al., 1997; Buneman et al., 1996; Fernández et al., 1998), these check for the existence of a path between s and t that satisfies the RE. For instance, an example query "*Find if two vertices with labels* Barack Obama *and* USA *are connected by a path containing only edge labels:* bornIn, locIn, *and* captialOf. *Additionally with the condition that* bornIn *should appear first followed by a sequence of either* locIn *or* capitalOf" can be expressed as a regular reachability query,

$$\text{Barack Obama} \overset{bornIn.(locIn \mid CaptialOf)*}{\rightsquigarrow} \text{USA}.$$

Another kind of connectivity queries include shortest path queries that take a pair of vertices s and t as input and return the shortest path (or distance) between them (see Section 2.1.1.4). Like reachability queries, shortest path queries can also be composed with additional regular expressions (REs) constraints.

Graph query languages like GraphLog (Consens and Mendelzon, 1989), **G+** (Cruz et al., 1988), UnQL (Buneman et al., 2000), SPARQL 1.1 (Prud'hommeaux et al., 2013), Gremlin (Rodriguez, 2015), Cypher (Neo4j, 2012) provide a framework for expressing connectivity queries. For instance, the above regular reachability query can be expressed in GraphLog as follows.

$$RRQ(\text{Barack Obama}, \text{USA})$$
$$\leftarrow (\text{Barack Obama}, (bornIn.(locIn \mid CaptialOf)*), \text{USA})$$

Similarly, a shortest path query "*Find shortest distance between* Barack Obama *and the* USA" with a regular expression constraint mentioned above can be expressed in GraphLog using the aggregate operators `min`, `sum` as follows.

$$SDQ(\texttt{min}(\texttt{sum}(d)))$$

$$\leftarrow (\text{Barack Obama}, \texttt{dist}(bornIn.(locIn \mid CaptialOf)*)(d), \text{USA})$$

- *Pattern Matching Queries.* A popular model for querying graph data are pattern matching queries. These queries deal with querying the substructures of a graph. A pattern matching query, by itself a graph, comprises of a set of patterns, each of which is a representative expression for zero or more edges in the graph. A pattern, in its typical form, is a triple $\langle u, e, v \rangle$, where all of u, e, v can be either a variable or a constant. The value of u, v takes an element from the vertex label set, while e takes an element from the edge label set. An answer to a pattern matching query is a set of possible bindings for all the query variables such that the graph obtained by replacing each variable with a binding forms a sub graph of the original graph. For instance, consider a graph like YAGO or Google's Knowledge Graph which comprises of information about people and things, an English query like "*Find people who are born in the USA and won both a Nobel Prize and a Grammy Award*" can be expressed as a pattern matching query (in GraphLog) as shown below.

$$\begin{aligned} PMQ(x) \leftarrow\ & (x, \text{bornIn}, y), \\ & (y, \text{locIn}, \text{USA}), \\ & (x, \text{won}, \text{Nobel Prize}), \\ & (x, \text{won}, \text{Grammy Award}) \end{aligned}$$

An answer to the above query is the set of all bindings for the variable x, i.e., all names of people who were born in the USA and won both a Nobel Prize and Grammy Award.

RDF's de facto query language SPARQL (Prud'hommeaux and Seaborne, 2008), an SQL like query language, is an another instance which uses pattern matching as the underlying semantics for expressing queries on RDF knowledge bases. SPARQL provides several clauses such as `SELECT`, `FROM`, `WHERE`, `UNION`, etc. which are alike to SQL constructs. An answer to a SPARQL query is a relational table where the columns denote the variables in the `SELECT` clause and the rows are the combination of bindings such that each combination matches the patterns expressed in the query. The above query can be represented in SPARQL as follows.

```
SELECT ?x
WHERE { ?x bornIn ?y. ?y locIn USA.
        ?x won Nobel Prize. ?x won Grammy Award.
}
```

Cypher [2] a declarative query language supported by the popular graph database Neo4j (Neo4j, 2012) is similar to SPARQL and uses the pattern matching query model for expressing queries. A typical cypher query has the following structure,

[2] https://neo4j.com/developer/cypher-query-language/

$$\texttt{MATCH}\ (\textit{pattern})\ \texttt{RETURN}\ (\textit{value}),$$

where patterns (*patterns*) are represented with the `MATCH` clause and the answer *value* to the query is caught using the `RETURN` clause. Our running example query can be written in Cypher as follows.

```
MATCH (x) - [:bornIn] - (y) - [:locIn] -> (USA)
MATCH (x) - [:won] -> (Nobel Prize)
MATCH (x) - [:won] -> (Grammy Award)
RETURN x
```

- *Aggregation Queries.* Aggregate queries are the class of queries which are used in computing graph properties. These include simple property computations like the size and order of a graph, in- and out-degrees of vertices, and more complex properties like vertex centrality, number of connected components, etc. Query languages like GraphLog, SPARQL, Cyper, and others provide support for aggregation queries. For instance, in GraphLog, constructs like `COUNT`, `MIN`, `MAX` are natively supported (Consens and Mendelzon, 1989) and SPARQL 1.1 (Prud'hommeaux et al., 2013) supports the aggregation constructs `COUNT`, `SUM`, `AVG`, `MIN`, `MAX` . Revisiting the aforementioned shortest distance example query in GraphLog,

$$\begin{aligned}&\textit{SDQ}(\texttt{min}(\texttt{sum}(d)))\\&\quad\leftarrow (\text{Barack Obama}, \texttt{dist}(\textit{bornIn}.(\textit{locIn} \mid \textit{CaptialOf})*)(d), \text{USA}),\end{aligned}$$

uses two aggregate functions `sum` and `min`, where `sum` returns the path length of a path between vertices Barack Obama and USA by summing up intermediate distances returned by the `dist` function, while `min` finds the minimum path among the set of paths.

- *Graph Creation Queries.* Unlike connectivity and pattern matching queries, these queries, on a graph G, create a new graph by extracting or summarizing the information in G. This is usually achieved by running connectivity, pattern matching, and/or aggregation queries as subqueries of a graph creation query. For instance, consider an academic social network where vertices represent people and edges represent collaboration information, and furthermore, each person has attributes such as expertise and affiliation. Given such a graph, a creation query like "*Find an affiliation graph where vertices represent universities and edges represent collaborations between two universities*" can be expressed (using Cypher) as follows.

  MATCH (*univ1:Person1.affiliation*) - [:Collaborate] -> (*univ2:Person2.affiliation*)
  CREATE *(univ1)* - [:Collaborate] -> *(univ2)*

  Evaluation of the above query returns an affiliation graph comprising of universities as vertex set, where an edge denotes a collaboration between two universities if there exist at least two persons, one from each university who have collaborated. Such queries can be natively expressed in many query languages such

as Cypher, SPARQL, GraphLog, G, G+, SNQL (Martín et al., 2011), SoQL (Ronen and Shmueli, 2009).

- *Approximate Queries.* Approximate querying is a common model of querying a graph in a scenarios where the user is not aware of the underlying graph topology, or in the case where queries are expensive to process. A practical application where approximate querying has profound significance, is in the domain of shortest path (or distance) querying over large networks such as road networks or social networks. To exemplify, consider the Facebook [3] social network which contains more than one billion vertices and more than one trillion edges, computing a shortest path between a pair of vertices, which has a time complexity of $\mathcal{O}(|V| + |E|)$ for a graph of size $|E|$ and order $|V|$ using Djikstra's algorithm (Dijkstra, 1959), is impractical to run large graphs like Facebook. In such scenarios, approximate shortest paths are of more interest than an exact shortest path. Approximate shortest path approaches like (Sarma et al., 2010; Gubichev et al., 2010; Cohen et al., 2003) are proposed to quickly estimate the shortest distance between a pair of nodes in real-time with comprising "a little" on the quality.

  Analogously, approximate methods for pattern matching queries have recently gained attention and approaches like (Khan et al., 2011; Kelley et al., 2004; Tian et al., 2007; Liang et al., 2006; Mongiovì et al., 2010; Tian and Patel, 2008; Tong et al., 2007) have been proposed to process pattern matching queries (with approximation) on large graphs.

2.1.3.3 Graph Database Systems

Historically, graph database systems belong to two classical types *relational-based* or *native architectures*. Both models have advantages and disadvantages which make the choice dependent on the application in use. In this section, we provide a quick glance at a few of the state-of-the-art graph database systems.

- *Relational Systems.* Continuing with similarities to the entity-relational model of relational databases, initial graph database systems were built based on the concept of relations. Some of the state-of-the-art graph systems that rely on this model include RDF-3X (Neumann and Weikum, 2008), SW-Store (Abadi et al., 2009), Hexastore (Weiss et al., 2008), SQLGraph (Sun et al., 2015), our own system TriAD (Gurajada et al., 2014a), commercial systems like SAP-HANA (Färber et al., 2012), IBM-DB2-RDF (Bornea et al., 2013), and others.

  Treating the graph as sequence of edges, relational-based systems store a graph either in a single relation or a set of relations partitioned by edge properties (Abadi et al., 2009). Graph queries are then processed, much alike to SQL queries, by a sequence of selection, projection, and join operations. With the high

[3] http://www.facebook.com

performance relational systems underneath, these systems are well suited for queries that do not have implicit navigational semantics, such include simple pattern matching queries like basic graph patterns (BGPs) in SPARQL 1.0. On the other hand, processing connectivity queries such as reachability, shortest paths, etc. require a number of self-joins to simulate the graph traversals, thus making the relational approaches not ideal for such queries. An efficient support for the navigational queries is crucial in the success of a graph database, thus leading to the emergence of many native graph systems.

- *Native Systems.* Native architectures for graph data systems that support navigational queries existed as early as the late 1960s by the introduction of IBM Information Management System (IMS) (Blackman, 1998) based on the principles of hierarchical data management. Later on systems like TORNADO (Atkinson et al., 1989), GemStone (Maier et al., 1986) based on the principles of object oriented databases are seen as an alternative to relational databases to store graph structured data. Late 1990s saw the proposals of numerous XML databases such as BaseX (Grün, 2011), Berkeley DB XML (Oracle, 2006) to name a few to store tree-structured XML data models.

  Recently, with the advent of social networks and knowledge graphs, native graph stores based on the NoSQL principles rose to prominence. Numerous graph stores both in centralized and distributed scenarios were proposed to tackle large graphs. All these systems use an adjacency list representation for easier navigation. Special graph systems such as GRAIL (Yildirim et al., 2010), FERRARI (Seufert et al., 2013), (Jin et al., 2008), etc. use a native graph model to efficiently answer reachability queries. On the other hand, general purpose graph systems like Neo4j (Neo4j, 2012), Microsoft's Trinity (Zeng et al., 2013; Shao et al., 2013), Google's Pregel (Malewicz et al., 2010), Facebook's Tao (Bronson et al., 2013), Apache Giraph (Martella et al., 2015), Apache Spark GraphX (Gonzalez et al., 2014), all provide a framework for performing efficient navigational and pattern matching queries on large graphs.

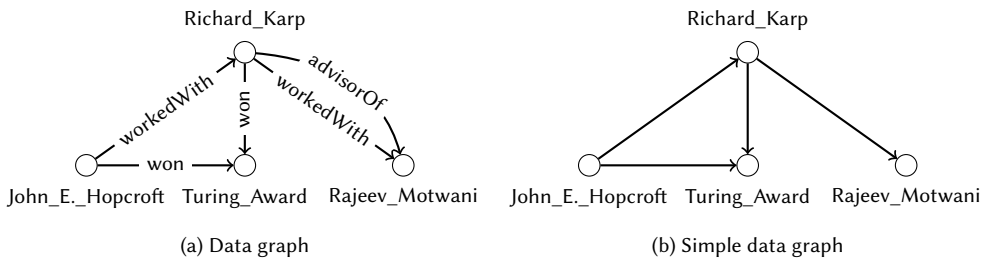
With this, we conclude our background section and next we look at the preliminaries where we discuss our data and query models used in this thesis.

2.2 Preliminaries

In this section, we introduce our data and query models and also establish necessary notations.

2.2.1 Data Model

We consider two graph data models. The first model is a directed, labeled multigraph model (see Section. 2.1.3.1), which forms the basis for the two query models we considered in this thesis, namely *basic graph patterns* and *generalized graph*



(a) Data graph

(b) Simple data graph

Figure 2.10: An example of (a) labeled directed multi-graph model (RDF) and (b) labeled directed graph model

patterns. The second data model is a simplified version of the first model which is used for dealing with a generalized form of reachability query model, called "*set reachability*".

2.2.1.1 Labeled directed multi-graph

A labeled directed multi-graph (see Section. 2.1.3.1) is defined as follows.

Definition 2.3. *A* **labeled directed multi-graph** *is a graph* $G_D(V, E, \Sigma_V, \Sigma_E, \Phi)$ *that comprises of vertex set* V*, vertex labels set* Σ_V*, edge label set* Σ_E*, and an edge set* $E \subseteq V \times \Sigma_E \times V$*. Function* $\Phi : V \rightarrow \Sigma_V$ *is an injective labeling function that maps each vertex in* V *to a unique label in* Σ_V*, i.e.,* $v_i = v_i$ *iff* $\Phi(v_i) = \Phi(v_j)$*.*

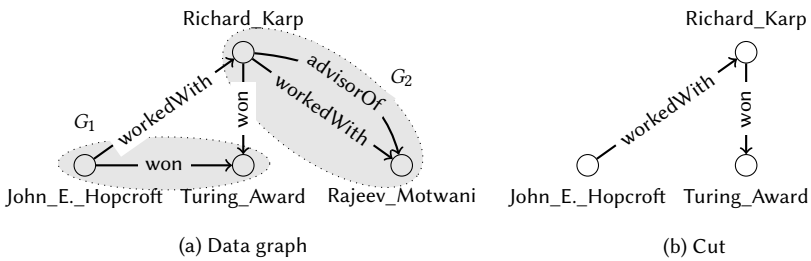
RDF Graph. We consider RDF as an instance for our "*labeled directed multi-graph*" model. As mentioned earlier, RDF is a W3C recommended model for representing linked information on the web. Real-world entities such as people, places, things or even biological entities such as proteins and genes form the vertex set, while an edge set denote the relationships among entities. Typical characteristics of an RDF graph is the injective mapping from vertex set V to vertex label set Σ_V, i.e., for every vertex, $v \in V$, has a unique label $\phi(v) \in \Sigma_V$.

Example 2.8. *An example of an RDF graph is shown in Figure 2.10(a).*

2.2.1.2 Labeled directed graph

We next define our second data model, called "*labeled directed graph*", which is a simplified notion of the labeled directed multi-graph. In this model each vertex has a unique label, while edges are unlabeled. A formal definition of this graph data model goes as follows.

Definition 2.4. *A* **labeled directed graph** *is a graph* $G_S(V, E, \Sigma_V, \Phi)$ *that comprises of vertex set* V*, an edge set* $E \subseteq V \times V$ *and function* $\Phi : V \rightarrow \Sigma_V$ *is an injective labeling function that maps each vertex in* V *to a label in* Σ_V*, i.e.,* $v_i = v_i$ *iff* $\Phi(v_i) = \Phi(v_j)$*.*



(a) Data graph (b) Cut

Figure 2.11: Example of (a) data graph partitioning $\mathcal{G} = \{G_1, G_2\}$ and it corresponding (b) Cut C

Social Network Graphs. As an instance of this model, we consider social network datasets such as Twitter [4], Facebook [5], etc. Twitter graph comprises of people and the "follows" relationship between two persons. Twitter graph can be modeled as a labeled directed graph by representing people as a vertex set, and a directed edge from person p_1 to person p_2 denotes that p_1 follows p_2. SNAP (Leskovec and Krevl, 2014) provides comprehensive list of some of real-world datasets such as Live Journal, Amazon, Google, Stanford, BerkStan, etc. which all are instances of second graph model.

Example 2.9. *An example of labeled directed graph is shown in Figure 2.10(b).*

2.2.1.3 Partitioned Graphs

As we deal with the distributed querying of labeled graphs, we partition our first and second graph models across multiple slaves following a shared-nothing master-slave setup. This allows us to scale our approaches to very large graphs. We partition a labeled directed multi-graph (and labeled directed multi-graph) as follows.

A labeled directed multi-graph $G(V, E, \Sigma_V, \Sigma_E, \phi)$ is partitioned into k vertex-disjoint subgraphs, $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ where each G_i is a edge-induced subgraph of G, and $\mathcal{G}$ is called the *partitioning* of G. Given a partitioning $\mathcal{G}$, we refer to $C(V_C, E_C, \Sigma_V, \Sigma_E, \phi)$ as the *cut*, which is an edge-induced subgraph of G, where $V_C \subseteq V$, $E_C = \{(u, v) \mid (u, v) \in E, u \in V_i, v \in V_j \text{ and } i \neq j\}$ with vertices $u, v \in V_C$, iff edge $(u, v) \in E_C$.

Example 2.10. *Figure 2.11(a) shows a graph partitioning for the example RDF data graph shown in Figure 2.10(a) and Figure 2.11(a) shows the corresponding cut.*

2.2.2 Query Model

In this section, we introduce three query models that we address in the thesis. The first query model is "*set reachability*", which belongs to the class of connectivitiy

[4] http://twitter.com
[5] http://facebook.com

queries. The second query model we consider is the *basic graph patterns* belonging to the class of pattern matching queries. Combining the semantics of first and second models, we address the third query model which we refer to as *generalized graph patterns*. We next discuss our three query models.

2.2.2.1 Set Reachability

Given a graph $G(V, E)$ and a pair of vertices $s, t \in V$, the reachability query $s \rightsquigarrow t$ addresses the problem of finding if there exists a path from s to t in G (see Section 2.1.1.4). Set reachability, also known as multi-source multi-target reachability (Gao and Anyanwu, 2013) is a generalization of the basic reachability problem in directed graphs. Given a pair of vertex sets $S, T \subseteq V$, a set-reachability query, denoted as $S \rightsquigarrow T$, is to find all pairs (s, t), where $s \in S$ and $t \in T$, are reachable.

In our work, we focus on the distributed version of set reachability queries on the partitioned labeled digraphs and refer to the problem as *distributed set reachability* (DSR), a DSR query is a set reachability query over partitioned simple data graph and formally defined as follows.

Definition 2.5. *Given a labeled directed graph $G(V, E, \Sigma_V, \phi)$, a k vertex-disjoint partitioning $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ of G, and a pair of vertex sets $S, T \subseteq V$, a* **DSR query***, denoted $S \rightsquigarrow T$, returns all pairs (s, t), with $s \in S$ and $t \in T$, that are reachable $s \rightsquigarrow t$ in G.*

Example 2.11. *Consider the example graph shown in Figure 2.10(b), and a pair of sets S =John_E._Hopcroft and T =Richard_Karp, Rajeev_Motwani. A set-reachability query $S \rightsquigarrow T$ is the set of all reachable pairs, namely {(John_E._Hopcroft, Richard_Karp), (John_E._Hopcroft,_Rajeev Motwani)}*

2.2.2.2 Basic Graph Patterns

We next define our second query model, *basic graph patterns* (BGP), which is based on the semantics of the pattern matching query model defined in Section 2.1.3.2. The fundamental unit of this query model is a triple pattern $\langle u, e, v \rangle$, where all of u, e, v can be either a variable or a constant. A BGP query is a conjunction of set of triple patterns, which together forms a query graph. An answer to a pattern matching query is, thus, a set of possible bindings for all variables such that a graph obtained by replacing each variable with a binding is the subgraph of the original graph.

For our work, we focus on the distributed version of BGP query model over a partitioned data graph. Formally the definition of a BGP query goes as follows.

Definition 2.6. *A* **BGP query** *is a labeled multi-digraph $Q(V_Q, E_Q, \Sigma_V, \Sigma_E, \mathcal{V}, \Phi_Q)$, where V_Q is a query vertex set, $\mathcal{V}$ is the set of query variables, and edge set $E_Q \subseteq V_Q \times \{\Sigma_E \cup \mathcal{V}\} \times V_Q$. Function Φ_Q is an injective labeling function from V_Q to $\{\Sigma_V \cup \mathcal{V}\}$.*

We consider SPARQL 1.0 (Prud'hommeaux and Seaborne, 2008) query language as the representative language for expressing BGP queries. Furthermore, an answer to a BGP query follows the semantics of SPARQL result semantics, where the result to a BGP query, however, is not itself a graph but – in analogy to SQL – a set of rows, each containing distinct set of bindings for the variables in the query.

Example 2.12. *Consider the example RDF graph shown in Figure 2.10(a). A query to "Find list of collaborators who both won a* Turing_Award *" can be expressed as a BGP query using SPARQL 1.0 query language as follows.*

```
SELECT ?p ?p1
WHERE { ?p won Turning_Award.
        ?p1 won Turning_Award.
        ?p2 workedWith ?p1}.
```

Evaluation of the above query returns the following relation.

?p	?p1
John_E._Hopcroft	Richard_Karp

2.2.2.3 Generalized Graph Patterns

Our third model constitutes *generalized graph patterns* (GGP), which again belong to the class of pattern matching queries and combines the semantics of our first two models, BGP and Set Reachability queries. A generalized triple pattern forms the basic unit of GGP queries and has the same structure as that of the triple pattern of BGP queries, i.e., $\langle u, e, v\rangle$, where all of u, e, v can be either a variable or a constant. However e can additionally be a regular expression over the edge label set Σ_E defining the constraints on the path from u to v. For instance, consider a generalized graph pattern,

$$\langle ?city, \mathsf{locIn}*, \mathsf{USA}\rangle$$

where u :=?*city* is a variable, and v := USA is a constant and e := locIn* is a regular expression stating that the path between u and v should contain only edges with label locIn. This translates to a set reachability query $S \rightsquigarrow T$, say S := {San Francisco$_1$, Atlanta$_2$, Honolulu$_n$, . . .} and T := {USA}.

Thus, GGP queries combines the semantics of BGP and the set reachability queries. For our work, we use SPARQL 1.1 (Prud'hommeaux et al., 2013) as a representative query language for GGP queries. However, we focus on a subset of queries expressible in SPARQL 1.1, in the following called as SwPP (SPARQL 1.0 with Property Paths) queries. Following the notations in SPARQL 1.1, we define below the grammar for a regular expression of a generalized triple pattern in SwPP queries.

Let path represent a regular expression defined for e,

$$\begin{aligned} \text{path} &:= \text{path/path} && (\textit{concatenation of paths}) \\ &:= \sigma && (\textit{single edge element}) \\ &:= \sigma? && (\textit{zero or one edge element}) \\ &:= \sigma* && (\textit{zero or more edge element}) \\ &:= \sigma+ && (\textit{one or more edge element}) \end{aligned} \tag{2.1}$$

where $\sigma \in \Sigma_E$. Using the above grammar, we now formally define GGP queries as follows.

Definition 2.7. *A* **GGP query** *is a labeled directed multi-graph* $Q(V_Q, E_Q, \Sigma_V, \Sigma_E, \mathcal{V}, \mathcal{L}, \Phi_Q)$ *where* V_Q *is a vertex set and edge set* $E_Q \subseteq V_Q \times \{\mathcal{V} \cup \mathcal{L}\} \times V_Q$*, where* $\mathcal{L}$ *is a language set for regular expressions over* Σ_E*, defined using the grammar shown in Equation 5.1.* $\mathcal{V}$ *is the set of query variable and* Φ_Q *is an injective mapping from* V_Q *to* $\{\Sigma_V \cup \mathcal{V}\}$.

Example 2.13. *Consider the example RDF graph shown in Figure 2.10(a). A query to "Find people who won a* Turing_Award *and transitively worked with* Rajeev_Motwani*" can be expressed as a GGP and written as an SwPP language as follows.*

```
SELECT ?p
WHERE { ?p won Turning_Award.
        ?p2 workedWith* Rajeev_Motwani }.
```

Evaluation of the above query returns the following relation.

?p
John_E._Hopcroft
Richard_Karp

Chapter 3

Set Reachability

In this chapter, we investigate our first query model, termed "*set reachability*", a generalization of the well-known reachability problem. Also known as multi-source multi-target reachability (Gao and Anyanwu, 2013), set-reachability is an important problem with a plethora of applications in analytics. We tackle this problem in a distributed setting, where a graph is partitioned and distributed, and refer to the problem as "*distributed set reachability*". We consider a partitioned, labeled distributed graph model, where the vertices are labeled, edges are directed and unlabeled, and furthermore, the graph is partitioned across multiple slaves .

3.1 Introduction

3.1.1 Motivation

With the wide adoption of graph models across many domains and the advent of social networks, knowledge graphs, etc., queries that check the connectivity between two vertices became a fundamental graph operation in many applications. The *reachability problem* in the directed graphs (see Section 2.1.1.4) has been well studied in this context (Agrawal et al., 1989; Jagadish, 1990; Cohen et al., 2003; Cheng et al., 2006; Wang et al., 2006; Chen and Chen, 2008; Jin et al., 2009; Yildirim et al., 2010; Schaik and Moor, 2011; Jin et al., 2011; Seufert et al., 2013; Yu and Cheng, 2010). To recap, the reachability problem on a labeled directed graph $G(V, E, \Sigma_V, \Phi)$, given a *source vertex* $s \in V$ and a *target vertex* $t \in V$, is to determine whether there exists a path from s to t over E.

To avoid redundant computations, many graph applications in fact require a generalization of this basic notion of reachability, where entire *sets S, T of source and target vertices*, respectively, need to be processed "at once". The resulting reachability problem, which we coin as *set reachability* and denote by $S \rightsquigarrow T$, aims to retrieve all pairs of source and target vertices (s, t), with $s \in S$ and $t \in T$, where s is reachable to t. Moreover, in case the graph is partitioned into multiple, vertex-disjoint subgraphs (e.g., when distributed across multiple slaves in a cluster), we

refer to the resulting set reachability problem as *distributed set reachability* (or "DSR" for short).

Applications. The set reachability problem has a plethora of applications in graph analytics and query-processing tasks. For example, with its recent update, SPARQL 1.1 (Prud'hommeaux et al., 2013) underwent a major revision in which the usage of *labeled property paths* allows a user to formulate transitive reachability constraints among the query variables. Since both the source and target variables of a property path may become bound to multiple RDF constants at query processing time, the processing of property paths in SPARQL 1.1 resolves to processing set reachability queries.

Another interesting application of set reachability is *community analysis* in social networks. That is, given a pair of source and target vertex sets, each representing social-net users such as on Twitter or Facebook, we may wish to efficiently detect which communities are densely connected. For example, consider two communities—billionaires and non-profit organizations—, it would be interesting to find the list of billionaires who are also involved in philanthropic activities.

Objectives. The key goals in processing a DSR query over a partitioned graph both efficiently and in a scalable manner are as follows.

(1) Avoid redundant computations within the local slaves as much as possible.

(2) Partially evaluate the local components of a set reachability query $S \rightsquigarrow T$ among all slaves in parallel.

(3) Minimize both the size and number of messages exchanged among the slaves.

3.1.2 State-of-the-art

A simple, or rather a naïve, way of solving a set reachability query is to translate it into a set of reachability queries. In other words, a set reachability query $S \rightsquigarrow T$ can be written as series of (single) reachability queries: $\{s \rightsquigarrow t \mid s \in S \text{ and } t \in T\}$.

The reachability problem has been well studied in the literature with a single reachability query, such as $s \rightsquigarrow t$, can now be efficiently solved using state-of-art indexing techniques like (Gao and Anyanwu, 2013; Jin et al., 2011; Kyrola et al., 2012; Prabhakaran et al., 2012; Seufert et al., 2013; Trißl and Leser, 2007; Schaik and Moor, 2011; Veloso et al., 2014; Yildirim et al., 2010). Most of them being largely limited to a centralized setting and, by design, they only partially address the point (1) of our objectives. Very recently, to fully address the point (1), there were attempts by (Gao and Anyanwu, 2013; Then et al., 2014) to more holistically solve a set reachability query. More specifically, (Gao and Anyanwu, 2013) uses a notion of *equivalence sets* among the graph vertices which effectively resolves to a preprocessing and indexing step of the input graph to predetermine these sets. On

the other hand, (Then et al., 2014) proposed a multi-source BFS (MS-BFS) strategy where BFS computations are shared across multiple vertices. Both being centralized approaches, are naturally limited to the main memory of a single machine and usually do not consider a parallel—in this case multi-threaded—execution of a set reachability query.

With the availability of very large graphs, scalable techniques for reachability query processing has received a lot of research attention. Fan et al. (Fan et al., 2012) recently proposed a method for distributed processing of single reachability queries based on the idea of partial evaluation. Being a single iteration approach, thus requiring only one round of communication, this approach is a great start in satisfying the aforementioned key goals. However, being designed to tackle only single reachability queries, this approach fails to satisfy the first objective, and furthermore, leaving behind the other objectives to be satisfied only partially.

On the other hand, general purpose distributed graph engines, such as Google's Pregel (Malewicz et al., 2010), Berkeley's GraphX (Xin et al., 2013) (based on Spark (Zaharia et al., 2010)), Apache Giraph (Martella et al., 2015), Blogel (Yan et al., 2014) and IBM's very recent Giraph++ (Tian et al., 2013), allow for the scalable processing of graph algorithms over massive, distributed data graphs. All of these provide generic API's for implementing various kinds of algorithms, including set reachability queries. However, a principal assumption we follow in this work is that set reachability queries are *selective*. That is, for any given sets S, T of source and target vertices, both S and T are usually much smaller than V, while the set of reachable pairs in turn usually is much smaller than the cross-product $S \times T$. Just like in relational approaches, an efficient processing of set reachability queries thus calls for the aforementioned usage of indexing strategies that take advantage of the salient properties of the graph. Graph indexing and the processing of selective queries however breaks the *node-centric computing* paradigm of Pregel and Giraph, where major amounts of the graph are successively shuffled through the network in each of the underlying MapReduce iterations (the so-called "supersteps").

Giraph++ is a very recent approach to overcome this rather myopic form of node-centric computing, which led to a new type of distributed graph processing that is coined *graph-centric computing* in (Tian et al., 2013). Blogel (Yan et al., 2014), on the other hand, proposed a *block-centric computing* where a graph is partitioned into coarse grained blocks. By exposing intra-node state information as well as the inter-node partition structure to the local slaves, both Giraph++ and Blogel are a great step towards making these graph computations more context-aware. However, index structures that specifically tackle the iterative communication rounds required for the supersteps are difficult to accomplish even here, such that a direct implementation of a reachability query may still result in as many iterations (and hence communication rounds) as the diameter of the graph in the worst case.

3.1.3 Our Approach & Contributions

3.1.3.1 Our Approach

Being the first, to the best of our knowledge, to investigate set reachability queries in a distributed setting, we propose an efficient and a scalable distributed solution. Our approach (Section 3.5.2) is based on the principles of *precomputation* and *indexing* for achieving efficiency, and *partial evaluation* (Fan et al., 2012) for the scalability. We assume a shared-nothing master-slave architecture as the underlying distributed setup based on which an input graph is partitioned using a either a hash-based or min-cut (Karypis and Kumar, 1998) partitioning schemes.

- *Efficiency.* Many centralized reachability approaches rely on indexes for efficiency. These approaches precompute the reachability for every pair of vertices and store them in a compact form which constitutes the index. In our approach, where the graph is partitioned, computing reachability for every pair of vertices is neither practical nor needed. Instead, we precompute and replicate reachability among a partial set of vertex pairs, specifically among the boundary vertices in each partition. Boundary vertices occur in two forms, *in-boundaries* and *out-boundaries*. An in-boundary is a vertex which has incoming edges from other partitions, while an out-boundary is a vertex which has outgoing edges to other partitions. We compute the reachability from in-boundaries to out-boundaries per partition and represent this reachability information along with the cut-edges in a graph form, called "*boundary graph*". The boundary graph is then replicated across the compute nodes and augmented with the local graph to form a new graph, called "*compound graph*". As the boundary graph encompasses transitive reachability information across the graph partitions, our approach requires at most a single round of communication regardless of the query size, the query vertices' locality and the graph topology. This benefits in minimizing the size and number of messages exchanged during query processing, fulfilling our third objective. Furthermore, the first objective, to avoid redundant computations, can be accomplished locally by constructing any off-the-shelf set-reachability index over the compound graph, and globally by grouping messages communicated among slaves.

- *Scalability.* Scalability is the another aspect which we aim to accomplish in our approach. As we rely on index-based set reachability query processing, scalability in our approach needs to be addressed in two dimensions, i) precomputation and indexing, and ii) query processing. The precomputation step, where a boundary graph is constructed for each partition, can be very expensive to compute. Moreover, precomputation, depending on the partitioning strategy, may result in huge boundary graph sizes. To scale the precomputation step to large graphs, we propose a novel equivalence-sets-based optimization to group in- and out-boundaries into equivalent sets and then compute the boundary graph with respect to the in- and out-equivalent sets. This effectively reduces

the precomputation times and further generates a smaller size boundary graphs, without losing any reachability information. In addition, further scalability can be achieved by compressing the compound graphs at each partition, and applying a graph condensation via computing the SCCs (see Section 2.1.1.6).

On the other hand, to scale a set-reachability query processing, we leverage the *partial evaluation* strategy proposed in (Fan et al., 2012). Here, the query $S \rightsquigarrow T$ is first written into multiple set reachability queries, $S_i \rightsquigarrow T_i$, each of which can be executed over a local compound graph residing at partition i. Further processing involves computing set reachability queries of the form $S_i \rightsquigarrow T_j$, where $i \neq j$. We follow a two-step reachability computation, i) $S_i \rightsquigarrow I_j$ at partition i, and ii) $I'_j \rightsquigarrow T_j$ at partition j. Where I_j is the set of in-boundaries for partition j and $I'_j \subseteq I_j$ is the set of in-boundaries that are reachable from the source vertex set S_i.

- *Extensibility.* Apart from the aspects such as efficiency and scalability, we also look at the extensibility of our indexes to dynamic graphs, which most of centralized and index-based reachability approaches ignore as the updates due to dynamic graphs trigger expensive rebuilding of partial or full indexes. In our approach, which relies on the precomputation and indexing, updates such as insertions and deletions trigger recomputation of boundary graphs. For (incremental) insertions, where new edges or vertices are added to the graph, the recomputation step merely comprises of differential reachability evaluation, i.e., checking only the in- and out-boundary pairs that are not reachable before, but are reachable after the insertions. As the insertions do not the void the existing reachability, the recomputation step can be quickly performed, and moreover, requires no decompression of compound graphs. On the other hand, deletions are tricky to handle in our approach, as it requires decompressing the compound graphs and may result in the recomputation step that is as expensive as the reconstruction of the indexes from scratch.

3.1.3.2 Contributions

We summarize the contributions of this chapter as follows.

- We formalize the problem of *set reachability* over a partitioned, and hence distributed, directed data graph. To our knowledge, our approach is the first to specifically tackle this problem.
- We develop a graph-based index structure that allows us to strictly restrict the communication protocol among the compute nodes to a *single round of message exchange* in order to resolve the results of any set reachability query posed against a given partitioning of the graph. This guarantee holds regardless of the properties of the graph (such as its diameter and partition structure) and the properties of the query (such as the distribution of the source and target vertices among the graph partitions).

- Our indexing strategy allows for *incremental updates* of the underlying graph, with an efficient support for vertex and edge insertions and a still preliminary support for respective deletions.
- Our approach is also *extensible* in the sense that any existing, centralized reachability index can be "plugged-in" at the local compute nodes. We report the results of our distributed approach in combination with a plain DFS search (Cormen et al., 2009), the MS-BFS approach of (Then et al., 2014), and FERRARI (Seufert et al., 2013) as local search strategies.
- Moreover, we provide an *extensive experimental evaluation* of our approach over a variety of both small and large graphs and in comparison to different extensions of Giraph++. We also investigate two application scenarios of our approach for processing SPARQL 1.1 queries with property paths and for detecting dependencies among social-network communities.

3.2 Preliminaries

In this section, we revisit the labeled directed graph model and the set reachability query model defined in Section. 2.2, and also establish specific notations that are particularly used in this chapter.

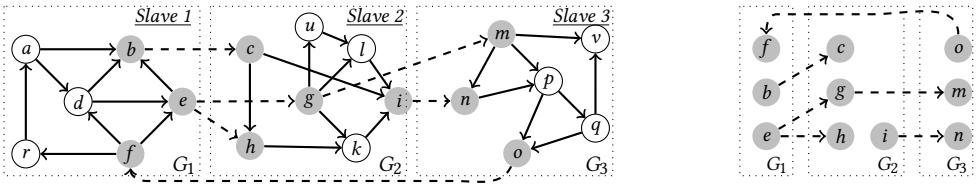
3.2.1 Data & Query Model

We consider a labeled directed graph model (see Section. 2.2.1.2) where vertices are (uniquely) labeled, and the edges are directed and unlabeled. Following Definition. 2.4, we denote the labeled directed graph as $G(V, E, \Sigma_V, \Phi)$ and refer to it as just "*graph*" for brevity.

Analogously to the partitioning of a labeled directed multi-graph (Definition. 2.3) described in Section. 2.2.1.3, we refer to $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ as the k vertex-disjoint partitioning, where each $G_i(V_i, E_i, \Sigma_V, \Phi)$ is a subgraph of graph G. We further refer to $C(V_C, E_C, \Sigma_V, \Phi)$ as the *cut*.

Distribution Function. We denote by $\rho : V \mapsto \mathbb{N}_0^+$ the *distribution function* that determines to which of the compute nodes (or *slaves*) in a cluster architecture each graph vertex $v \in V$ is distributed. Without loss of generality, and to simplify our notation for the following presentation, we assume a simple partitioning strategy by distributing every vertex $v \in V_i$ to slave i (i.e., $\rho(v) = i$ for each $v \in V_i$). We will thus refer to a graph partition and a slave interchangeably. To increase concurrent executions (e.g., when using multi-threading at the local compute nodes), an "overpartitioning" strategy may be employed instead, by assigning multiple graph partitions to each of the slaves.

Definition 3.1. *For a given graph partitioning $\mathcal{G}$ and an implied cut C of a graph G, we define the set of* **in-boundaries** *I_i for partition G_i as $I_i = \{v \mid v \in V_i, \exists(u, v) \in E_C,$*



(a) Graph G (b) Cut C

Figure 3.1: (a) Graph G with partitions $\mathcal{G} = \{G_1, G_2, G_3\}$ and (b) respective cut C

$u \in V_j$ and $i \neq j\}$, i.e., as the set of vertices in G_i that have an incoming edge from the cut C.

Conversely, we define the set of **out-boundaries** *$O_i = \{v \mid v \in V_i, \exists(v, u) \in E_C, u \in V_j$ and $i \neq j\}$ as the set of vertices in G_i that have an outgoing edge into the cut C.*

Example 3.1. *Figure. 3.1(a) shows an example graph G with three partitions $\mathcal{G} = \{G_1, G_2, G_3\}$ which are stored at three slaves. Its corresponding cut C is shown in Figure. 3.1(b). In- and out-boundaries are $I_1 = \{f\}$, $O_1 = \{b, e\}$, $I_2 = \{c, g, h\}$, $O_2 = \{i\}$, and $I_3 = \{m, n\}$, $O_3 = \{o\}$, respectively.*

Revisiting Definition. 2.5, a distributed set reachability (**DSR query**) takes a pair of sets as an input S, T and partitioning $\mathcal{G}$ of G, where $S \subseteq V$ ("source set") and $T \subseteq V$ ("target set"), and returns all pairs (s, t), with $s \in S$ and $t \in T$, where s, t are reachable in G, i.e., $s \rightsquigarrow t$.

Example 3.2. *For the graph G shown in Figure. 3.1, a DSR query $S \rightsquigarrow T$ with $S = \{a, d, g\}$ and $T = \{l, p\}$ returns the following reachable pairs: $\{(a, l), (a, p), (d, l), (d, p), (g, l), (g, p)\}$.*

3.2.2 Graph Partitioning Strategies

Graph partitioning plays an important role in that it significantly influences the performance of DSR query processing. Here, we discuss two graph partitioning strategies, one based on partitioning the vertices of a graph, called as *edge-cut partitioning*, and the other based on partitioning the edges, called as *vertex-cut partitioning*.

- *Edge-Cut Partitioning.* Given a partitioning $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ and cut $C(V_C, E_C, \Sigma_v, \Phi)$ of a graph $G(V, E, \Sigma_v, \Phi)$, we call $\mathcal{G}$, an *edge-cut partitioning*, if it holds that $V_i \cap V_j = \emptyset$, where $i \neq j$ and $1 \leq i, j \leq k$. In other words, each partition (or subgraph G_i) holds a non-overlapping subset of vertices from V. This further implies that each partition comprises of a unique subset of edges, such that an edge $(u, v) \in E_i$, if and only if, both $u, v \in V_i$. Moreover, the cut graph C comprises of the inter-partition edges and the induced vertices, i.e., $(u, v) \in E_C$ and $u, v \in V_C$, if and only if, $(u, v) \in E$, $u \in V_i$, and $v \in V_j$ for $i \neq j$.

An edge-cut partitioning can be done using either a *hash-based* or a *min-cut* strategy. In a hash-based partitioning, a hash function is used to place vertices on to the non-overlapping partitions. An ideal hash function should uniformly distribute the vertices such that each partition has an equal number of vertices, which can result in better load balancing. On contrary, hash-based methods incur high communication costs which are proportional to the number of cut-edges, i.e., $|E_C|$. Gonzalez et al. (Gonzalez et al., 2012) provided a theoretical formulation on the expected fraction of cut-edges for the case of random hash-based partitioning. Given a random hash-based partitioning $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ and cut $C(V_C, E_C, \Sigma_V, \Phi)$ of G, expected fraction of cut-edges is:

$$\mathbb{E}\left[\frac{|E_C|}{|E|}\right] = 1 - \frac{1}{k}$$

Such a partitioning can significantly impact the performance of both indexing and querying steps, which can be mitigated with a *min-cut* based partitioning strategy. The goal of the min-cut based partitioning is to minimize the expected fraction of cut-edges, which by itself is a NP-Complete problem (Orlin, 1977). Several approximate approaches have been proposed (Buluc et al., 2013) and software tools like METIS (Karypis and Kumar, 1998) provide a very good approximations for minimizing the expected fraction of cut-edges, while scaling to very large graphs.

- *Vertex-Cut Partitioning.* Alternatively, a graph can be partitioned using vertex-cut partitioning. In this strategy, each edge, rather than a vertex, is hashed on to a unique partitioning. That is, a partitioning $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ is called a vertex-cut partitioning of graph G if for each edge $(u, v) \in E$ then there exists a partition i such that $(u, v) \in E_i$. A vertex-cut partitioning may lead to partitions having overlapping vertex sets, i.e., $|V_i \cap V_j| \geq 0$ for $i \neq j$. Since each edge belongs to a specific partition, it is to be noted that the cut graph comprises only cut-vertices and empty cut-edges. Like in the case of edge-cut partitioning, vertex-cut partitioning can be done based on either edge hashing or optimizing on the number of vertex replications (i.e., vertex-cut). Apache GraphX (Gonzalez et al., 2014) facilitates using vertex-cut partitioning to partition graphs, for achieving better load-balance.

Unless otherwise stated, we consider only edge-cut partitioning as the partitioning strategy in our setting. We empirically compare both the hash-based and min-cut strategies.

3.3 Related Work

Having introduced the set reachability problem, in this section, we list some of the existing approaches which are related to or aid in solving set reachability queries.

We start with the centralized approaches and then discuss some of the popular distributed graph frameworks that can be programmed to support set reachability queries.

- *Centralized Approaches.* The reachability problem in directed graphs is one of the most fundamental graph problems and thus has been tackled by a plethora of *centralized indexing techniques* (Cohen et al., 2003; Gao and Anyanwu, 2013; Jin et al., 2011; Kyrola et al., 2012; Prabhakaran et al., 2012; Seufert et al., 2013; Trißl and Leser, 2007; Schaik and Moor, 2011; Veloso et al., 2014; Yildirim et al., 2010). All of these aim to find better tradeoff between query time and indexing space which, for a directed graph $G(V, E)$, are in between $\mathcal{O}(|V| + |E|)$ for both query time and space consumption when no indexes are used, and $\mathcal{O}(1)$ query time and $\mathcal{O}(|V|^2)$ space consumption when the transitive closure is fully materialized.

  Recently, Gao et al. (Gao and Anyanwu, 2013) proposed a suitable, but centralized, indexing strategy, based on a notion of *equivalence sets* of graph vertices that have the same reachability properties. (Then et al., 2014), on the other hand, focused on the *query-time optimization* of multi-source BFS searches. However, there exist hardly any works so far on distributed reachability queries (Fan et al., 2012). Fan et al. (Fan et al., 2012) recently discussed distributed, but single-source, single-target reachability algorithms for partitioned graphs and also provided performance guarantees. For a directed graph and given cut, (Fan et al., 2012) uses a partially iterative and partially indexing-based evaluation to find the reachability for a given pair of vertices over a classical master-slave architecture. In Section 3.4.1, we therefore provide a detailed review of the techniques proposed in (Fan et al., 2012), while the query-time processing we perform based on equivalence sets to a large extent resembles also the techniques described in (Gao and Anyanwu, 2013; Then et al., 2014) for a centralized setting. However, unlike in Gao and Anyanwu (2013), we do not enumerate the actual path sequences.

- *Distributed Graph Engines.* Distributed graph engines such as Pregel (Malewicz et al., 2010), GraphX (Xin et al., 2013), GraphLab (Low et al., 2010, 2012), Trinity (Shao et al., 2013), PowerGraph (Gonzalez et al., 2012), Giraph (Martella et al., 2015) and Giraph++ (Tian et al., 2013) are either based on MapReduce (Malewicz et al., 2010; Xin et al., 2013; Gonzalez et al., 2012), or they implement their own, proprietary communication protocols via Message Passing (Gonzalez et al., 2012; Low et al., 2012; Shao et al., 2013). Giraph, for example, offers the sendMessage(.) and compute(.) methods as generic API functions to implement various kinds of graph algorithms (including BFS and DFS). To implement a *single-source, single-target reachability query* over a directed graph, each iteration over the compute(.) method (as it is required for a single BFS/DFS step), however, results in a new call of the Map function or so-called "superstep". Among two such supersteps, messages are communicated among all compute nodes, which is a

strategy that—due to the a-priori unknown amount of iterations—usually does not permit for interactive query response times. For *multi-source, multi-target queries*, on the other hand, this approach scales well with the query size due to the possibility to implement shared computations in the compute(.) method.

A similar observation holds for GraphLab (Low et al., 2012), Trinity (Shao et al., 2013) and PowerGraph (Gonzalez et al., 2012) which implement asynchronous protocols based on the Message Passing Interface (MPI) (MPI et al., 2009). PowerGraph, for example, which is specifically tuned for skewed graphs, implements a judiciously chosen schedule of exchanged messages, but also here the worst-case amount of iterations remains equal to the diameter of the graph. The very recently proposed Giraph++ (Tian et al., 2013), Blogel framework (built on top of Giraph) provides further optimizations by shifting from a purely node-centric to either a graph-centric ("think like a graph") or block-centric compute paradigm. All (local) messages among the vertices within the same graph partition are performed inside a superstep, while other messages are processed only between two such supersteps. This significantly improves the performance by minimizing the number of messages, but also gives a much higher degree of freedom in the implementation of various graph algorithms.

Thus, on the one hand, the generic abstraction layers of these distributed graph engines make it difficult to exploit shared computations and yet to fully benefit from the underlying distribution scheme. On the other hand, these engines generally do not support graph-indexing techniques known from the centralized approaches, which could ideally be employed to even completely avoid iterative communication rounds among the compute nodes. Since Giraph++ offers the most flexible API among the aforedescribed engines, we extensively compared our approach against two principle implementations of DSR queries in the very recent Giraph++ framework (including one native Giraph version).

3.4 Distributed Reachability

Here, we discuss two paradigms, a non-iterative and an iterative approach, for processing a distributed reachability query, i.e., *single-source single-target reachability* query. We start with a non-iterative approach, which forms the basis for our approach to process DSR queries. We also discuss iterative approaches which we compare against our preferred non-iterative approach.

3.4.1 Non-iterative Approach

Fan et al. (Fan et al., 2012) recently proposed a non-iterative approach for processing a distributed reachability query $s \rightsquigarrow t$ over a master-slave architecture and is evaluated as follows. The master receives the query $s \rightsquigarrow t$ and communicates it to all slaves. At partition G_i, containing the source s, a local evaluation of the reachability of s to each vertex in the set of out-boundaries O_i is computed first.

Similarly, at partition G_j, containing the target t, a local evaluation of the reachability of each vertex in the set of in-boundaries I_j is computed. Additionally, a local computation of the reachability between all in-boundaries I_i and out-boundaries O_i is computed at each partition $G_1, \ldots, G_k$, and hence, at all slaves $i = 1..k$ in the compute cluster *in parallel*.[1]

The resulting local reachability information is then encoded into a bipartite graph with in-boundaries and out-boundaries forming the vertex set and the edge set represents the local connectivities between the in-boundaries I_i (including the source s, if present) and the out-boundaries O_i (including the target t, if present) at each partition G_i. All of these local bipartite graphs are communicated back to a single master node for the final evaluation. A *query-specific global dependency graph* is then constructed at this master node, for $s \rightsquigarrow t$, by merging the bipartite graphs and the static cut C. A reachability algorithm is then run over the dependency graph to answer $s \rightsquigarrow t$.

Algorithm 1 depicts the pseudo code of the distributed reachability approach proposed by (Fan et al., 2012) for a master-slave architecture. The master node, for a given pair of vertices s, t, invokes a partial evaluation, $\mathsf{localDG}(G_i, s, t)$, on each slave and constructs a global dependency graph G_{dep} by merging the local bipartite graphs G^i_{loc} returned from each slave i and the cut C. On this query specific global dependency graph G_{dep}, a reachability query $s \rightsquigarrow t$ is evaluated by running a BFS/DFS algorithm. The above algorithm can be implemented with a single round of communication (Fan et al., 2012) and, for example, can be processed in a single MapReduce iteration.

Complexity. The above algortihm performs the following sequence of operations for a reachability check – 1) partial evaluations at all slaves in parallel, 2) communication of local bipartite graphs G^i_{loc} and the construction of global dependency graph G_{dep}, and 3) reachability evaluation of $s \rightsquigarrow t$ over G_{dep}. Step 1, i.e., partial evaluation, constitutes the reachability computation from sets I_i to O_i at each slave i. Let m be the index of the largest graph partition in $\mathcal{G}$, the time complexity of this step is $\mathcal{O}\big((|V_m| + |E_m|) \cdot \min(|I_m|, |O_m|)\big)$, i.e., time taken to perform one BFS traversal from each member in either the set I_m or the O_m, whichever has the minimum size. The second step involves communication of locally computed bipartite graphs and the construction of global dependency graph. The cost of this step is proportional to the size of all the locally constructed bipartite graphs and the cut C, which is equal to $\mathcal{O}\big(\sum_{i=1}^{k}(|I_i| \cdot |O_i|) + |E_C|\big)$ in worst case. Finally, the third step involves a reachability evaluation of a pair s, t over G_{dep}, which can be performed in $\mathcal{O}(|V_{dep}| + |E_{dep}|)$ time by using a BFS/DFS graph traversal algorithm (Cormen et al., 2009).

Example 3.3. *Consider the distributed reachability query $d \rightsquigarrow q$ over the graph partitioning shown in Figure 3.1. The local evaluation at each partition results in the*

[1]Note that we follow a slightly different definition of in- and out-boundaries than in (Fan et al., 2012). However, the algorithm in (Fan et al., 2012) directly translates to the one outlined above.

Input: Partitioning $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ of graph G, Query: $s \rightsquigarrow t$
Output: Boolean value: *true* / *false*

1 **Master:**
2 $result := \emptyset$
3 **foreach** G_i *in* $\mathcal{G}$ **do**
4 $\quad G_{dep} := G_{dep} \cup$ localDG(G_i, s, t) ▷ `Local dependency graph evaluation`
5 $G_{dep} := G_{dep} \cup C$
6 **if** reachable(G_{dep}, s, t) **then return** *true* ;
7 **else return** *false* ;

8 **Slave** i**:** localDG(G_i, s, t)
9 initialize $G^i_{loc}(V^i_{loc}, E^i_{loc}, \Sigma_V, \Phi) := \emptyset$ ▷ $V_{loc} := \emptyset, E_{loc} := \emptyset$
10 $IS := I_i$ ▷ `in-boundaries set`
11 $OS := O_i$ ▷ `out-boundaries set`
12 **if** s *in* V_i **then** $IS := IS \cup \{s\}$;
13 **if** t *in* V_i **then** $OS := OS \cup \{t\}$;
14 **foreach** b_i *in* I_i **do**
15 $\quad$ **foreach** b_o *in* O_i **do**
16 $\quad\quad$ **if** reachable(G_i, b_i, b_o) **then**
17 $\quad\quad\quad V^i_{loc} := V^i_{loc} \cup \{b_i, b_o\}$
18 $\quad\quad\quad E^i_{loc} := E^i_{loc} \cup (b_i, b_o)$

19 **return** G^i_{loc}

Algorithm 1: Distributed reachability evaluation (Fan et al., 2012)

following representation of partial reachability information:

- $G_1 : \{d \rightsquigarrow b, d \rightsquigarrow e, f \rightsquigarrow b, f \rightsquigarrow e\}$,
- $G_2 : \{c \rightsquigarrow i, g \rightsquigarrow i, h \rightsquigarrow i\}$,
- $G_3 : \{m \rightsquigarrow q, m \rightsquigarrow o, n \rightsquigarrow q, n \rightsquigarrow o\}$.

By including the edges in the cut C (Figure 3.1(b)), the global dependency graph (Figure 3.2) is constructed at the master node to finally resolve $d \rightsquigarrow q$. By running a reachability algorithm (such as backward DFS) over the dependency graph, one can find that $d \rightsquigarrow q$ is indeed true *(the red path in Figure 3.2).*

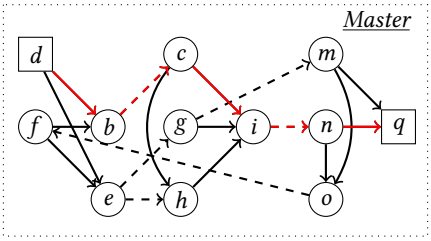
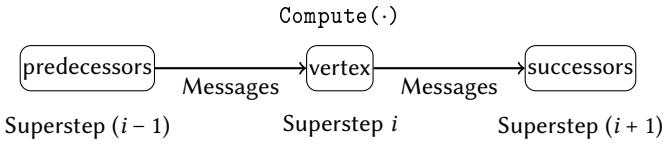


Figure 3.2: Dependency graph as constructed in (Fan et al., 2012) for a single reachability query

3.4.2 Iterative Approach

Vertex-centric approaches such as Pregel (Malewicz et al., 2010), Apache Giraph (Martella et al., 2015), GraphX (Gonzalez et al., 2014) based on Apache Spark (Zaharia et al., 2010), GraphLab (Low et al., 2010) are popular distributed graph processing frameworks, which can be programmed to process reachability queries and can easily scale well to very large graphs. A typical vertex-centric approach uses the following processing model.



Each vertex, in superstep i (i.e., iteration i), receives messages from its predecessors that were sent in the superstep $i - 1$, processes the messages in its local compute(.) function, updates the vertex's value, and finally sends messages, if any, to its successors which are subsequently received in superstep $i + 1$.

Using the above model, a reachability query $s \rightsquigarrow t$ is processed as follows. In superstep 0, source s, marked as visited, sends a message (s's ID) to all its successors in $succ(s)$ (see Section 2.1.1.3). In the next superstep, each vertex $v \in succ(s)$ receives this message, if v is not visited, marks itself as visited and forwards the received message (s's ID) to $succ(v)$. This process continues iteratively until either $v = t$ or v is visited, then v stops forwarding messages to its successors and halts, reporting $s \rightsquigarrow t$ if $v = t$. An iterative approach may take at most d iterations to process a single reachability query, where d is the diameter (see Section 2.1.1.3) of the input graph.

3.5 Non-iterative Approaches

In this section, we discuss three non-iterative approaches for processing a DSR query. We start with a naïve approach (Section 3.5.1.1) and then present an improved solution (Section 3.5.1.2), both of which are based on the non-iterative distributed reachability solution that uses a query specific global dependency graph, as discussed in Section 3.4.1. Next, we present our new approach (Section 3.5.2) based on the principles of *precomputation* and *indexing* of partial reachability information, and *partial evaluation* based query processing to mitigate the shortcomings of naïve and improved approaches.

3.5.1 Dependency Graph based Approaches

3.5.1.1 Naïve Approach

A *naïve approach* to extend the distributed reachability problem (Fan et al., 2012) to sets of vertices S, T would be to simply invoke a separate reachability query

```
Input: Partitioning 𝒢 = {G_1, G_2, ..., G_k} of graph G, Query S ⇝ T
Output: R {(s, t) | s ∈ S, t ∈ T and s ⇝ t}

1  Master:
2  result := ∅
3  foreach s in S do
4      foreach t in T do
5          G_dep(V_dep, E_dep, Σ_V, Φ) := ∅                    ▷ V_dep := ∅, E_dep := ∅
6          foreach G_i in 𝒢 do
7              G_dep := G_dep ∪ localDG(G_i, s, t)
8          G_dep := G_dep ∪ C
9          if reachable(G_dep, s, t) then result := result ∪ {(s, t)} ;
10 return result
```

Algorithm 2: A naïve approach to process a DSR query using dependency graph approach (Fan et al., 2012)

$s \rightsquigarrow t$ for every pair (s, t), with $s \in S$ and $t \in T$. Pseudo code for processing a DSR query $S \rightsquigarrow T$ using this naïve approach is depicted in Algorithm 2. The master node receives the query and, for each pair (s, t), a dependency graph G_{dep} is constructed using the local bipartite graphs G^i_{loc} returned by the slaves and the cut C. A local bipartite is constructed at each slave using the localDG(·) function (see Algorithm 1). On G_{dep}, which is constructed for every pair (s, t), a reachability query $s \rightsquigarrow t$ is invoked. If $s \rightsquigarrow t$ is `true`, i.e., reachable, then the pair (s, t) is added to the result R.

However, an obvious reason for the limited efficiency of this approach, even for reasonably-sized sets S and T, is that this approach tends to repeatedly perform the expensive global dependency graph computation for each subquery, albeit the significant portions of the global dependency graph are the same. Consequently, this approach can also not reuse any intermediate computations and thus likely to perform many redundant computations, leading to very serious performance problems in proccesing DSR queries.

In the next subsection, we propose an improved approach to holistically solve a DSR query that avoids multiple global dependency graphs construction for a single DSR query.

3.5.1.2 Improved Approach

An *improved approach* to extend the distributed reachability algorithm provided in (Fan et al., 2012) to sets is as follows. Let $S \rightsquigarrow T$ be the query received at the master. First, we partition $S \rightsquigarrow T$ into subqueries $S_1 \rightsquigarrow T_1, S_2 \rightsquigarrow T_2, \ldots, S_k \rightsquigarrow T_k$, where k is the number of graph partitions, such that each $S_i \subseteq V_i$ and $T_i \subseteq V_i$ contains only vertices that are local to partition G_i. Next, a local evaluation at each slave i involves finding the reachability among all pairs of vertices from the sets $S_i \cup I_i$ and $O_i \cup T_i$, respectively. These can again be run *in parallel* across all slaves. The resulting reachability information, again represented as a bipartite graph, is

Input: Partitioning $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ of graph G, Query: $S \rightsquigarrow T$
Output: R $\{(s, t) \mid s \in S, t \in T$ and $s \rightsquigarrow t\}$

1 **Master:**
2 $result := \emptyset$
3 partition S, T into $\{(S_1, T_2), (S_2, T_2), \ldots, (S_k, T_k)\}$
  ▷ `where` $S_i \subseteq V_i$ `and` $T_i \subseteq V_i$
4 **foreach** G_i *in* $\mathcal{G}$ **do**
5 $G_{dep} := G_{dep} \cup$ localDG(G_i, S_i, T_i)
6 $G_{dep} := G_{dep} \cup C$
7 **foreach** s *in* S **do**
8 **foreach** t *in* T **do**
9 **if** *reachable*(G_{dep}, s, t) **then** $result := result \cup \{(s, t)\}$;
10 **return** *result*

11 **Slave** i**:** localDG(G_i, S_i, T_i)
12 initialize $G_{loc}(V_{loc}, E_{loc}, \Sigma_V, \Phi) := \emptyset$ ▷ $V_{loc} := \emptyset, E_{loc} := \emptyset$
13 $IS := I_i \cup S_i$ ▷ `in-boundaries set`
14 $OS := O_i \cup T_i$ ▷ `out-boundaries set`
15 **foreach** b_i *in* IS **do**
16 **foreach** b_o *in* OS **do**
17 **if** *reachable*(G_i, b_i, b_o) **then**
18 $V_{loc} := V_{loc} \cup \{b_i, b_o\}$
19 $E_{loc} := E_{loc} \cup (b_i, b_o)$
20 **return** G_{loc}

Algorithm 3: An improved approach to process a DSR query using dependency graph approach (Fan et al., 2012)

then communicated from all slaves to the master node for the final evaluation. At the master node, the query-specific global dependency graph for the sets S, T is constructed as described in Section 3.4.1, and a local reachability algorithm is then used to emit all reachable pairs (s, t), with $s \in S$ and $t \in T$.

Pseudo code for the *improved approach* is depicted in Algorithm 3. At the master node, the set reachability query $S \rightsquigarrow T$ is first partitioned in to k subqueries $S_1 \rightsquigarrow T_1$, $S_2 \rightsquigarrow T_2$, $\ldots$, $S_k \rightsquigarrow T_k$ (Line 3). Analogous to the distributed reachability approach (Algorithm 1), a query specific global dependency graph G_{dep} is constructed from the local bipartite graphs and the cut C (Lines 4-5). Unlike in Algorithm 1, the local bipartite graphs are constructed using the reachability information from the sets $I_i \cup S_i$ to the sets $O_i \cup T_i$ (Lines 11-19). On G_{dep}, the DSR query $S \rightsquigarrow T$ is processed using a plain BFS/DFS traversal algorithm, which is abstracted by the reachable(·) function.

Example 3.4. *Consider the DSR query $S \rightsquigarrow T$ with $S = \{a, d, g\}$ and $T = \{l, p\}$ over the cut C shown in Figure 3.1(b). The sets of Boolean formulas obtained after the local evaluation at each slave are as follows:*

- $G_1 : \{a \rightsquigarrow b, a \rightsquigarrow e, d \rightsquigarrow b, d \rightsquigarrow e, f \rightsquigarrow b, f \rightsquigarrow e\}$,

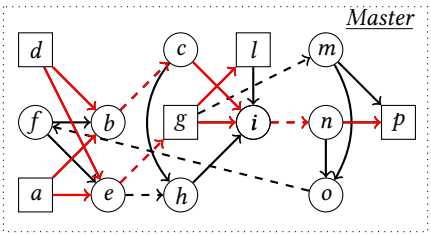


Figure 3.3: Dependency graph as constructed in (Fan et al., 2012) for a DSR query

- $G_2 : \{c \rightsquigarrow i, g \rightsquigarrow i, g \rightsquigarrow l, h \rightsquigarrow i\}$,
- $G_3 : \{m \rightsquigarrow p, m \rightsquigarrow o, n \rightsquigarrow p, n \rightsquigarrow o\}$.

At the master node, after evaluating $S \rightsquigarrow T$ over the global dependency graph shown in Figure 3.3, we obtain the following reachable pairs of source and target vertices: $\{(a, l), (a, p), (d, l), (d, p), (g, l), (g, p)\}$.

3.5.1.3 Discussion

Although the second algorithm provides a more viable solution of the DSR problem than the naïve approach, it still leaves a number of disadvantages that limit both its efficiency and scalability.

- First, the query-dependent, global dependency graph is generated "from scratch" for each query $S \rightsquigarrow T$, although both the cut C and the local reachability information $I_i \rightsquigarrow O_i$ among the in- and out-boundaries at each graph partition G_i are in fact static.
- Second, the approach does not leverage any distributed computation in its second step, as the final reachability computation $S \rightsquigarrow T$ over the global dependency graph is performed only by a single master node.
- Third, since the global dependency graph is generated dynamically for each query $S \rightsquigarrow T$, a local reachability index for the static cut C and the local $I_i \rightsquigarrow O_i$ components cannot be constructed, which restricts the final reachability computation to either a simple BFS or DFS strategy over the global dependency graph.

3.5.2 Our Approach

In *our approach*, instead of computing the global dependency graph for each incoming query from scratch at the master node, we precompute a partition-specific variant thereof, called the "boundary graph", only once and store this boundary graph in the form of a static reachability index at each slave. This strategy provides multiple benefits. First, it avoids repeated computations of the boundary

graph for each query. Second, since each slave has the complete reachability information among the boundary vertices of all other slaves available, finding the reachability of any two vertices (s, t) in the entire data graph G resolves to a local reachability computation at at most two slaves, which is irrespective of the diameter of the graph and the distribution of the source and target vertices of a set-reachability query (see Theorems 3.1 and 3.2). Additionally, an index can be built over the static boundary graph to accelerate this processing. Third, storing a (compacted version of the) boundary graph at each slave allows for a fully distributed processing of a set-reachability query and thus avoids the single-node bottleneck of previous approaches. We next formally define how we generate the boundary graph and its derived index structures.

3.5.2.1 Boundary Graph

A *boundary graph* is a directed graph that represents the reachability information among the in- and out-boundaries of all graph partitions $\mathcal{G} = \{G_1, \ldots, G_k\}$ with respect to a given cut C.

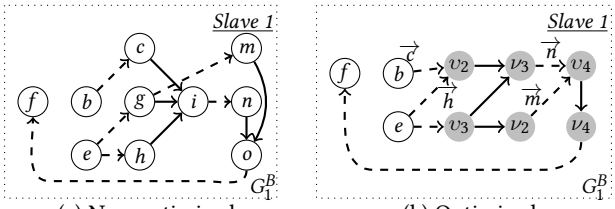
Definition 3.2. *Let $G_i^B(V_i^B, E_i^B, L, \phi)$ denote the **boundary graph** we compute for partition G_i, such that the following holds:*

- *The vertices $V_i^B = \bigcup_{i=1..k} I_i \cup O_i$ consist of the union of all in- and out-boundaries of all partitions $G_1, \ldots, G_k$.*
- *There exists an edge $(u, v) \in E_i^B$, iff*
  - *$(u, v) \in E_C$, or*
  - *$u \in I_j$ and $v \in O_j$, for $j \neq i$, and $u \rightsquigarrow v$ (i.e., u and v are both located at another partition G_j and there exists a path from u to v in G_j).*

That is, the boundary graph for partition G_i merges the static cut C with the static reachability information $I_j \rightsquigarrow O_j$ among all the remaining graph partitions G_j (for $i \neq j$) into a new, precomputed graph G_i^B. The resulting boundary graphs are thus partition-specific.

Example 3.5. *For our graph G with partitions G_1, G_2, G_3 and respective cut C as shown in Figure 3.1, the boundary graph G_1^B for partition G_1 is shown in Figure 3.4(a). Here, the dashed edges refer to edges in the cut C, while the solid edges denote the transitive pairwise reachability $I_j \rightsquigarrow O_j$ (for $j \neq 1$).*

Complexity. The construction of the boundary graph requires us to materialize the pairwise reachability $I_i \rightsquigarrow O_i$ among the in- and out-boundaries for each partition G_i. Using a simple BFS/DFS-based approach, the worst time complexity of this computation is $\mathcal{O}((|V_i|+|E_i|)\cdot \min(|I_i|, |O_i|))$ per partition. This can be further improved to $\mathcal{O}(1 \cdot |I_i| \cdot |O_i|)$ when using a sophisticated, local reachability index for this operation. On the other hand, the (worst-case) space complexity for storing the boundary graph at partition i is $\mathcal{O}(\sum_{j=1}^{k} |I_j| \cdot |O_j| + |E_C|)$, for $j \neq i$. From this, one can deduce that both the time and space complexity of the boundary graph



(a) Non-optimized (b) Optimized

Figure 3.4: Boundary graph G_1^B for partition G_1

computation strongly depend on the amounts of in- and out-boundaries we obtain based on the cut C.

Min-k-Cut Partitioning. A standard approach to reduce this number of boundary vertices is to reduce the number of edges in the cut C, while trying to keep the sizes of the partitions $G_1, \ldots, G_k$ balanced. Although finding an optimal such *min-k-cut* partitioning is a well-known NP-complete problem (Cormen et al., 2009), current graph libraries such as METIS (Karypis and Kumar, 1998) are capable of achieving very good approximations even for graphs with hundreds of millions of edges.

Equivalence Sets. Even for a given cut C, we can further reduce the size of the boundary graph by grouping the in- and out-boundary vertices into *equivalence sets*, thus continuing the idea presented in (Gao and Anyanwu, 2013) to a distributed setting. Specifically, we achieve this by grouping the boundary vertices into *forward-* and *backward-equivalent sets* according to the following definition.

Definition 3.3. *Two in-boundaries b_1, b_2 are called **forward-equivalent** with respect to subgraph G_i, i.e., $b_1 \equiv^f b_2$, iff for any vertex $v \in V_i - I_i$ and $b_1 \rightsquigarrow v$, it holds that $b_2 \rightsquigarrow v$.*

*Conversely, two out-boundaries b_1, b_2 are called **backward-equivalent** with respect to subgraph G_i, i.e., $b_1 \equiv^b b_2$, iff for any vertex $v \in V_i - O_i$ and $v \rightsquigarrow b_1$, it holds that $v \rightsquigarrow b_2$.*

That is, once the forward- and backward-equivalent sets of vertices are identified for each subgraph G_i, each such set is replaced by a new *in-virtual vertex* υ (for a forward-equivalent set) and a new *out-virtual vertex* ν (for a backward-equivalent set), respectively.

Example 3.6. *Following the above definition of forward/backward equivalence for the partitioning $\mathcal{G} = \{G_1, \ldots, G_3\}$ of G shown in Figure 3.1(a), we can obtain the following, partition-specific equivalence sets:*

- *At G_1, $\{\upsilon_1 = \{f\}\}$, $\{\nu_1 = \{b, e\}\}$,*
- *At G_2, $\{\upsilon_2 = \{c, h\}, \upsilon_3 = \{g\}\}$, $\{\nu_2 = \{g\}, \nu_3 = \{i\}\}$*
- *At G_3, $\{\upsilon_4 = \{m, n\}\}$, $\{\nu_4 = \{o\}\}$*

Next, the in- and out-boundaries are redefined with respect to the new virtual vertices. That is, I_i comprises of all in-virtual vertices and O_i comprises of all out-virtual vertices. For example, the optimized boundary graph for partition G_1 is shown in Figure 3.4(b). Note that we attach additional labels to the cross-edges in the boundary graph to obtain a loss-less representation of the boundary graph with respect to the partitions $G_1, \ldots, G_k$. For example, the cross-edge (b, c) is represented by connecting the vertex b and the in-virtual vertex υ_2 with the label $\overrightarrow{c}$ to denote that b is connected to only c in υ_2. The forward arrow denotes that this connection is valid only for a forward exploration. This is required, since vertices c, h are forward-equivalent, i.e., $c \equiv^f h$, with respect to partition G_2 only.

Computing Equivalence Sets. According to Definition 3.3, two in-boundaries $b_1 \in I_i$ and $b_2 \in I_i$ are forward-equivalent if they are reachable to exactly the same set of vertices in $V_i - I_i$. To determine the sets of forward-equivalent boundaries, we need to (1) compute all reachable pairs from I_i to $V_i - I_i$ and then (2) group the vertices in I_i into these equivalence sets. For large sets I_i and $V_i - I_i$, this computation may be prohibitively expensive. To address (1) and thus reduce the input that needs to be considered for (2), we apply the following optimizations.

- b_1, b_2 can only be forward-equivalent with respect to partition G_i if both belong to the same *strongly connected component* (SCC) in G_i. We thus condense each G_i into a more compact DAG by computing the SCCs over G_i.
- Instead of considering all target vertices $V_i - I_i$, we consider only the *direct successors* $S(I_i)$ of I_i, and hence $S(I_i) - I_i$, to check for forward-equivalence. The intuition for considering only successors is that if two boundaries b_1, b_2 are reachable to the same set of vertices in $S(I_i) - I_i$, by induction, b_1, b_2 also are reachable to the same set of vertices in $V_i - I_i$.

A similar construction then holds also for backward-equivalence, except that *predecessors* $P(O_i)$ are considered instead.

Example 3.7. *Consider partition G_3 with in-boundary set $I_3 = \{m, n\}$ to compute the sets of forward-equivalent vertices in I_3. In this case, this requires us to only verify whether $m \equiv^f n$, since m, n are the only in-boundaries in I_3. First, we run the SCC algorithm to condense G_3 into the DAG G_3'. In this example, $G_3' = G_3$, and we see that m, n do not belong to the same SCC. We then check their forward-equivalence based on the sets of vertices in $V_3 - I_3$ that are reachable from both m and n. To compute these reachable sets of vertices, we consider only the direct successors $S(I_3) - I_3 = \{p, v\}$ instead of considering all of $V_3 - I_3 = \{p, o, q, v\}$. Thus, the reachable set of vertices of both m and n is $\{p, v\}$, and hence we have $m \equiv^f n$.*

Algorithm 4 computes the forward-equivalent sets of vertices in each graph partition G_i as follows. Given a graph partition G_i with in-boundaries I_i, the forward-equivalent sets EQ_i^f are computed as follows. First, the graph is condensed into its DAG representation G_i' by computing the strongly connected components of G_i. Next, the target vertices $S(I_i) - I_i$ are chosen as the successors of the

```
   Input: Subgraph G_i, In-boundaries I_i
   Output: Forward-equivalent sets EQ_i^f
1  EQ_i^f := ∅
2  G'_i := condense(G_i)            ▷ graph condensation via SCC computation
3  S(I_i) := successors(I_i, G'_i)
4  rep[1..|I_i|] := true
5  rset[k] := ∅
6  for l = 1 ... |I_i| do
7      if rep[l] then
8          v := {b_l}
9          if rset[b_l] = ∅ then
10             rset := localSetReachability({b_l}, S(I_i) − I_i)
11         for m = l + 1 ... |I_i| do
12             if scc(b_l) = scc(b_m) then
13                 v := v ∪ {b_m}
14                 rep[m] := false
15             else
16                 rset := localSetReachability({b_m}, S(I_i) − I_i)
17                 if rset[b_l] = rset[b_m] then
18                     v := v ∪ {b_m}
19                     rep[m] := false
20         EQ_i^f := EQ_i^f ∪ v
```

Algorithm 4: Computing forward-equivalent sets

in-boundaries I_i. We define a Boolean array rep[] (for "representative"), whose truth value for a given boundary b_m denotes whether a forward-equivalent set (i.e., an in-virtual vertex) v is formed with any other boundary b_l, for $m > l$. The rep[] array is initially set to "true" for all boundaries. A boundary b_l is equivalent to b_m, iff either b_l and b_m belong to same SCC (Lines 11-14) or have the same reachability set rset (Lines 17-19). rset[j] for the jth boundary denotes the set of vertices from $S(I_i) - I_i$ that are reachable from b_j. The computed equivalence set v starting at boundary b_l, where rep[b_l]:=true, is added to EQ_i^f at the end of each iteration (outer loop – Lines 6-20).

With minor modifications from $S(I_i) - I_i$ to $P(O_i) - O_i$ (thus using predecessors instead of successors), the algorithm can similarly be adapted to compute the backward-equivalent sets EQ_i^b of out-boundaries.

3.5.2.2 Compound Graph

After compacting the partition-specific boundary graphs G_i^B by replacing both the forward- and backward-equivalent sets of vertices with their in- and out-virtual counterparts, we perform one more step to obtain our final graph index for evaluating DSR queries. To do so, we merge the partition-specific boundary graphs

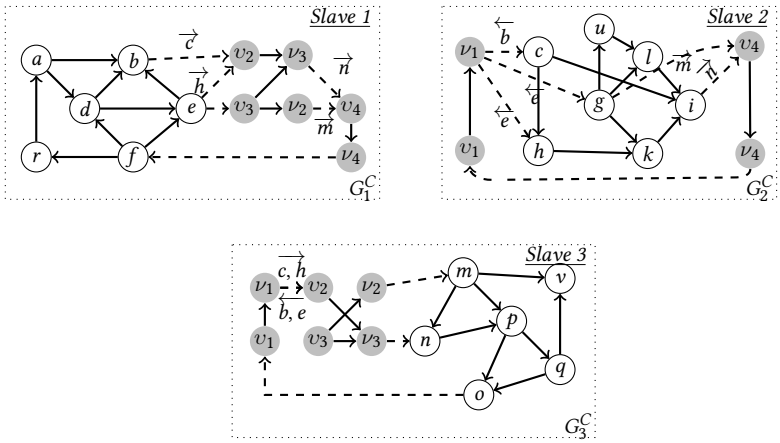


Figure 3.5: Final compound graphs G_1^C, G_2^C, G_3^C constructed for graph G with cut C of Figure 3.1

with the local partitions into a *compound graph* G_i^C for each partition G_i. These compound graphs will facilitate the processing of DSR queries via a combination of local reachability computations and a single filtering step among these local results.

Definition 3.4. *Let $G_i^C(V_i^C, E_i^C, L, \phi)$ denote the* ***compound graph*** *we compute for partition G_i, such that the following holds:*

- *The vertices $V_i^C = V_i \cup V_i^B$ consist of the union of vertices in the local subgraph G_i and boundary graph G_i^B.*
- *The edges $E_i^B = E_i \cup E_i^B$ consist of the union of edges in the local subgraph G_i and boundary graph G_i^B.*

Figure 3.5 shows the compound graphs for the initial data graph G from Figure 3.1(a).

3.5.2.3 Forward- and Backward-Lists

Our last precomputation step consists of storing the forward- and backward-lists, F_i and B_i, of boundaries which are non-local to each partition G_i. These will serve for routing messages to only those partitions G_j which are connected to G_i. Specifically, the *forward-list*,

$$F_i = \bigcup_{j \neq i} \{ v \mid v \text{ is in-virtualvertex of } G_j \},$$

is the set of all vertices that are non-local to G_i and are in-virtual vertices of another partition G_j. Similarly, the *backward-list*,

$$B_i = \bigcup_{j \neq i} \{ \nu \mid \nu \text{ is out-virtual vertex of } G_j \},$$

consists of all out-virtual vertices that are non-local to G_i.

For instance, for partition G_1 shown in Figure 3.5, we have $F_1 = \{\upsilon_2, \upsilon_3, \upsilon_4\}$ and $B_1 = \{\nu_2, \nu_3, \nu_4\}$.

3.5.2.4 Evaluating DSR Queries

Given these precomputed index structures, i.e., the compound graphs G_i^C and respective forward- and backward-lists, F_i and B_i, evaluating a DSR query now becomes straightforward. We again begin with a discussion of the single-source, single-target case and then explain how it generalizes to the multi-source, multi-target case.

A. Single Reachability. Consider the reachability query $s \rightsquigarrow t$. The algorithm for processing the query is shown in Algorithm 5. Given a data graph G with partitioning $\mathcal{G}$, we evaluate the query as follows. If both s and t belong to same partition G_i, then the reachability $s \rightsquigarrow t$ is confined to only partition i which stores the compound graph G_i^C. Since the compound graph G_i^C augments each G_i with the global reachability information among all boundary vertices, we can safely evaluate the reachability of $s \rightsquigarrow t$ on G_i^C by calling any centralized reachability algorithm via the function localSetReachability(.) (Lines 11-13). A formal justification for this is provided by the following theorem.

Theorem 3.1. *Let s, t both be local vertices of partition i, i.e., $s, t \in V_i$. Then the evaluation of the reachability $s \rightsquigarrow t$ over graph G can be answered entirely locally over the compound graph G_i^C without requiring any message exchange among the partitions.*

Proof. Let $P = \{(s, u_1), \ldots, (u_m, u), (u, v), (v, v_n), \ldots, (v_1, t)\}$ denote the set of edges along a path from source s to target t. Then the following holds: edge $(u, v) \in P$, with $u \in V_i$, $v \in V_j$, can either be (1) a *cut edge* iff $i \neq j$, (2) a *local edge* in partition i iff $i = j$, or (3) a *non-local edge* with respect to partition k iff $i = j$ and $i \neq k$.

<u>*Case A:*</u> Let all edges in P be either local to partition i (1) or be a cut edge (2) among partitions i, j. Then, from Definition 3.4 of the compound graph $G_i^C = (V_i^C, E_i^C)$, it follows that $P \subseteq E_i^C$. That is, the reachability $s \rightsquigarrow t$ can be computed entirely locally at partition i using E_i^C.

<u>*Case B:*</u> Let $(u, v) \in P$ such that (u, v) is a non-local edge (3) to partition i but a local edge (1) to another partition j, with $i \neq j$. That is, $(u, v) \in E_j^C$ but $(u, v) \notin E_i^C$. From this, it follows that $\exists\, p, q$ such that the edges (u_{p-1}, u_p), $(v_q, v_{q-1}) \in P$ are cut edges (2), where u_p, $v_q \in V_j$ with $1 \leq p \leq m$ and $1 \leq q \leq n$. Next, we choose (u_{p-1}, u_p), $(v_q, v_{q-1}) \in P$ as the edges with the largest indices of p, q for which this property holds. This choice ensures that a path from u_p to v_q via (u, v) resides entirely in partition j. Then, vertex u_p forms an in-boundary while vertex v_p forms an out-boundary of partition j, and the edges of the sub-path $\{(u_{p-1}, u_p), \ldots, (u_m, u), (u, v), (v, v_n), \ldots, (v_q, v_{q-1})\} \subseteq P$ reside in partition j. In this case, by the construction of the boundary graph $G_i^B = (V_i^B, E_i^B)$, we added

```
Input: Compound graphs: {G_1^C, G_2^C, ..., G_k^C}, Query: s ⇝ t
Output: true/false
1  Master:
2  rank_s := ρ(s)                    ▷ i.e., s ∈ V_i and G_i is at Slave ρ(i)
3  rank_t := ρ(t)                    ▷ i.e., t ∈ V_j and G_j is at Slave ρ(j)
4  result := false
5  foreach rank do
6      result := result ∨ compute(s, rank_s, t, rank_t)
       ▷ invokes parallel computations at all ranks
7  return result
8  Slave i:
9  method compute(s, rank_s, t, rank_t) :
10 rset := ∅
11 if rank_s = i and rank_t = i then
       ▷ invoke local reachability evaluation
12     if localSetReachability({s}, {t}) ≠ ∅ then
13         return true
14 else if i = rank_s then
15     j := rank_t
16     ϒ_s^j := localSetReachability({s}, F_i^j);
       ▷ F_i^j ⊆ F_i is the set of in-virtual vertices local to j
17     rset[s] := ϒ_s^j
18     sendMessage(j, rset)
19     return false
20 else if i = rank_t then
21     receiveMessage(i, rset)
22     ϒ_s^i := rset[s]
23     for υ in ϒ_s^i do
24         b := υ.rep                ▷ b is a member vertex in eqset υ
25         if localSetReachability({b}, {t}) ≠ ∅ then
26             return true
27 return false
```

Algorithm 5: Distributed reachability processing

a reachability edge (u_p, v_q) to E_i^B (see Definition 3.2). This means, that in the optimized E_i^B, we add an edge (υ, ν), where $u_p \in \upsilon$ is an in-virtual vertex and $v_q \in \nu$ is an out-virtual vertex. Since $E_i^B \subseteq E_i^C$ (see Definition 3.4), there exists a path $\{(s, u_1), (u_1, u_2), \ldots, (u_{p-1}, \upsilon), (\upsilon, \nu), (\nu, v_{q-1}), \ldots, (v_1, t)\}$ in partition i, thus again ensuring that the reachability of $s \rightsquigarrow t$ can be computed entirely locally at partition i using E_i^C. □

Example 3.8. *Consider the query $b \rightsquigarrow f$. Both vertices b, f are local to partition G_1. By considering only the subgraph G_1, one cannot find that f is reachable from b. But by considering the whole graph G, we see that $b \rightsquigarrow f$ is* true *via the path $b \to c \to i \to n \to p \to o \to f$. However, using the local compound graph G_1^C (see Figure 3.5), we can indeed find that $b \rightsquigarrow f$ is* true *via the path $b \to \upsilon_2 \to \nu_3 \to \upsilon_4 \to \nu_4 \to f$.*

If, on the other hand, s and t are located at two different partitions G_i, G_j, with $i \neq j$, the evaluation of a reachability query works as follows (Lines 14-25). Starting at partition G_i, we find the reachability from s to all the forward-boundaries $\upsilon \in F_i^j \subseteq F_i$ (Line 15) which are located at another partition G_j. Let $\Upsilon_s^j \subseteq F_i$ be the set of in-virtual vertices located at partition G_j (and hence stored by partition j as per our assumption) which are reachable from s. The message rset[s]:=$\langle s, \Upsilon_s^j \rangle$ is then communicated to partition j. At partition j, we consider each $\upsilon \in \Upsilon_s^j$ and replace it with any one of its members b, after which we evaluate the reachability from b to the local target vertex t. If there exists one such $b \in \upsilon \in \Upsilon_s^j$ with $b \rightsquigarrow t$, we report that $s \rightsquigarrow t$ is *true* (Lines 22-25).

Theorem 3.2. *Let $s \in V_i$ and $t \in V_j$, with $i \neq j$. Then, the evaluation of the reachability $s \rightsquigarrow t$ can be answered over the two compound graphs G_i^C and G_j^C by using a single step of message exchange from partition i to partition j.*

Proof. The proof is simple and leverages the result of Theorem 3.1. Let $P = \{(s, u_1), (u_1, u_2), \ldots, (u_{p-1}, u_p), \ldots, (u_n, t)\}$ denote the set of edges along the path from source s to target t, where $s \in V_i, t \in V_j$. If $i \neq j$, then there exists an edge $(u_{p-1}, u_p) \in P$ which is a cut edge (1). That is, $u_{p-1} \in V_k$, $u_p \in V_j$ and $k \neq j$. Next, we choose the largest index p such that the subpath $\{(u_p, u_{p+1}), \ldots, (u_n, t)\}$ resides entirely at partition j. Since (u_{p-1}, u_p) is a cut edge and $u_p \in V_j$, u_p forms an in-boundary of partition j. We next choose the smallest q such that $\{(s, u_1), \ldots, (u_{q-1}, u_q)\}$ resides entirely at partition i. Note that in the optimized boundary graph, we actually use virtual vertices instead of the regular ones, which we omit here for simplicity.

Path P can be thus written as a concatenation of subpaths $P_1 = \{(s, u_1), \ldots, (u_{q-1}, u_q)$, $P_2 = (u_q, u_{q+1}) \ldots, (u_{p-1}, u_p\}$, and $P_3 = \{(u_p, u_{p+1}), \ldots, (u_n, u_t)\}$. According to Theorem 3.1, P_1 and P_3 can be computed entirely at partition i and j, respectively. u_q and u_p thus are an out- and in-boundary of partition i and j, respectively, i.e., $u_q, u_p \in V_C$. As per the construction of the local compound graphs (see Definition 3.4), P_2 can be evaluated at either partition i or j. Thus, the

reachability problem $s \rightsquigarrow t$ is reduced to two reachability problems: (a) $s \rightsquigarrow u_p$ at partition i and (b) $u_p \rightsquigarrow t$ at partition j. To find such a u_p, we iterate over all in-boundaries b of partition j residing at partition i. We then compute the reachability $s \rightsquigarrow b$ and communicate the reachable in-boundaries to partition j. Thus, answering $s \rightsquigarrow t$, where $s \in V_i$ and $t \in V_j$, with $i \neq j$, requires a local processing at two partitions i, j and involves only a single step of message exchange from partition i to partition j. □

Example 3.9. *Consider the query $a \rightsquigarrow q$, where a is located at partition G_1 and q is located at partition G_3. At partition G_1, we compute the reachability from a to the single forward-boundary $\{v_4\}$ which is located at G_3. From the compound graph G_1^C (shown in Figure 3.5), we have $\Upsilon_a^3 = \{v_4\}$ since $a \rightsquigarrow v_4$. Υ_a^3 is then communicated to partition 3. At partition 3, we expand the actual vertices represented by the virtual vertex v_4 (say m, since $m \in v_4$) and find the reachability from m to q. Since $m \rightsquigarrow q$, we thus find that $a \rightsquigarrow q$ is* true.

B. Set Reachability. An actual DSR query $S \rightsquigarrow T$, which is received by the master node, is processed in our approach as shown in Algorithm 6. First, $S \rightsquigarrow T$ is partitioned into subqueries $S_1 \rightsquigarrow T_1, S_2 \rightsquigarrow T_2, \ldots, S_k \rightsquigarrow T_k$, where k is again the number of graph partitions. The partitioning of the query into these subqueries is determined such that each source vertex $s_i \in S_i$ and target vertex $t_i \in T_i$ resides locally at partition G_i (Line 2).

Step 1. (Lines 13-19) A local evaluation at partition G_i involves processing the pairwise reachability among the vertices from S_i to T_i and from S_i to F_i at *all partitions $i = 1..k$ in parallel.* This operation generates two types of reachable pairs: (s_i, t_i) and (s_i, v_j). The first type denotes the reachability between both a local source $s_i \in S_i$ and a local target $t_i \in T_i$. The second type denotes the reachability between a local source $s_i \in S_i$ and a forward-boundary $v_j \in F_i$, which is represented by an in-virtual vertex located at partition j.

Step 2. (Lines 21-32) The communication of the remotely reachable pairs, each of the form (s_i, v_j), is performed from partition i to partition j among *all pairs of partitions $i, j = 1..k$ in parallel.* In order to reduce the overhead of communicating individual pairs, each partition buffers its partial reachability information and communicates this buffer at once. Each buffer sent from partition i to partition j is of the form $\{\langle s_i, \Upsilon_{s_i}^j \rangle\}$ for all $s_i \in S_i$. For easier processing, the messages received at partition i from all other partitions are stored in an inverted index $\mathcal{I}_i(\Upsilon_*^i, L_i)$, where Υ_*^i is the aggregated set of in-virtual vertices (local to partition i). For each in-virtual vertex $v \in \Upsilon_*^i$, its aggregated non-local source set $S_v \subseteq S$ is stored in L_i. That is, for $s \in S_v$ and $v \in \Upsilon_*^i$, we already know that $s \rightsquigarrow v$.

Step 3. (Lines 34-39) A final local evaluation involves processing the set reachability $\Upsilon_*^i \rightsquigarrow T_i$ from the in-virtual vertices Υ_*^i to the target sets T_i at *all partitions $i = 1..k$ in parallel.* For each in-virtual vertex $v \in \Upsilon_*^i$ and original vertex b represented by v, we evaluate the reachability from b to all targets $t \in T_i$. If $b \rightsquigarrow t$ is *true*, then for each $s \in S_v$, we report that $s \rightsquigarrow t$ is *true*.

```
Input: Compound graphs {G_1^C, G_2^C, ..., G_k^C}, Query: S ⇝ T
Output: R {(s, t) | s ∈ S, t ∈ T and s ⇝ t}
1  Master:
2  partition S, T into {(S_1, T_2), (S_2, T_2), ..., (S_k, T_k)}
   ▷ where S_i ⊆ V_i and T_i ⊆ V_i
3  result := ∅
4  for i = 1 ... k do
5      result := result ∪ compute(S_i, T_i)
6  return result
7  Partition i:
8  method compute(S_i, T_i) :
9  local_rset := ∅
10 remote_rset := ∅
11 result := ∅
12 // Step 1:
13 local_rset := localSetReachability(S_i, T_i)
14 remote_rset := localSetReachability(S_i, F_i)
15 for s in S_i do
16     for t in local_rset[s] do
17         result := result ∪ {(s, t)}
18     for υ in remote_rset[s] do
19         ϒ_s^j := ϒ_s^j ∪ υ
           ▷ υ is an in-virtual vertex of partition j
20 // Step 2:
21 for j = 1 to k do
22     if j ≠ i then
23         msg := ∅
24         for s in S_i do
25             msg := msg ∪ {⟨s, ϒ_s^j⟩}
26         sendMessage(j, msg)
27 I_i(ϒ_*^i, L_i) = ∅
28 for j = 1 to k do
29     receiveMessage(j, msg)
30     for ⟨s, ϒ_s^i⟩ in msg do
31         for υ in ϒ_s^i do
32             I_i[υ] := I_i[υ] ∪ {s}
33 // Step 3:
34 for υ in ϒ_*^i do
35     b := υ.rep
36     local_rset := localSetReachability({b}, T_i)
37     for s in I_i[υ] do
38         for t ∈ local_rset[b] do
39             result := result ∪ {(s, t)}
40 return result
```

Algorithm 6: Distributed set reachability Processing

Example 3.10. *Consider again the graph G with partitions G_1, G_2, G_3 in Figure 3.1(a). The respective compound graphs G_1^C, G_2^C, G_3^C are shown in Figure 3.5. Let $S = \{d, l, p\} \rightsquigarrow T = \{a, k, q\}$ be the DSR query received at the master node. The query is partitioned into $\{d\} \rightsquigarrow \{a\}$, $\{l\} \rightsquigarrow \{k\}$, $\{p\} \rightsquigarrow \{q\}$. At partition G_1, we find the set-reachability (Step 1) between $\{d\}$, $\{v_2, v_3, v_4, a\}$, thus returning the reachable pairs $\{(d, v_2), (d, v_3), (d, v_4), (d, a)\}$. We perform the same operation in parallel at partitions 2 and 3 and communicate the results to all other partitions (Step 2). At partition 1, we receive the following reachability information: $\{(v_1, [l, p])\}$. Similarly, at partition 2, we receive $\{(v_2, [d, p]), (v_3, [d, p])\}$; and at partition 3, we receive $\{(v_4, [d, l])\}$. At the end of the local evaluation from boundaries to the final targets (Step 3), by replacing virtual vertices with each of their represented vertices (at partition 1, v_1 is replaced with f), the following sets of reachable pairs are generated at the partitions.*

- *At G_1, $\{(d, a), (l, a), (p, a)\}$*
- *At G_2, $\{(d, k), (l, k), (p, k)\}$*
- *At G_3, $\{(d, q), (l, q), (p, q)\}$*

Local Reachability Evaluation. Algorithms 5 and 6 both require partial reachability processing at each partition via the function localSetReachability(.). For this, any centralized reachability index (see, e.g. (Cormen et al., 2009; Seufert et al., 2013; Gao and Anyanwu, 2013; Yildirim et al., 2010)) can be plugged into our framework. We abstract this by calling "black-box" function localSetReachability(.) in our algorithms whenever a local (set-)reachability operation is invoked.

Forward vs. Backward Processing. Our above discussion focused on starting from the source vertices and ending at the target vertices. If there are less targets than sources, one may also start from the target vertices and search backwards to the source vertices to arrive at the same results. We therefore maintain both forward- and backward-lists, F_i and B_i, to facilitate these two directions of searching.

3.5.2.5 Incremental Updates

Insertions. Insertions over the SCC-condensed compound graphs G_i^C can be implemented *without* storing the original (i.e., uncondensed) compound graphs.

Let (u, v) denote a new edge that is to be inserted into the graph G. First, assume both u and v belong to the same graph partition i. Further, if u, v belong to the same SCC, then adding (u, v) to G_i would not change the local compound graph G_i^C (nor any other) at all and thus can be safely ignored. If, on the other hand, u, v belong to two different SCCs, then a series of update actions are required. First, we add the new edge to the local compound graph G_i^C and locally recompute the SCCs and equivalence sets. Next, new connections among the local in- and out-boundaries, I_i and O_i, are communicated to all other partitions j (for $j \neq i$) as additional edges. These can be incrementally merged into all the

compound graphs G_j^C by updating their SCCs as well. Second, if u and v belong to two different partitions i and j, then this means we have a new edge in the cut C, which however does not affect the reachability within partitions i and j. Thus, (u, v) can directly be merged into the distributed compound graphs as described above.

Let n, m denote the number of vertices and edges in the condensed compound graph G_i^C, and let $|I_i|$, $|O_i|$ be the number of in- and out-boundaries for partition i, respectively. By adding a local edge to partition i, a partial or full recomputation of the connections among vertices from I_i to O_i is required. Thus the worst-case time complexity of this step is $\mathcal{O}((n + m) \cdot |I_i| \cdot |O_i|)$, which is asymptotically optimal (Demetrescu and Italiano, 2006). The SCC recomputation at each compound graph has a time complexity of $\mathcal{O}(n' + m')$, where n' and m' are the numbers of vertices and edges in the new G_i^C's.

Deletions. Deletions over the SCC-condensed compound graphs G_i^C, on the other hand, result in a decremental maintenance of the SCCs, which requires either storing the original (i.e., uncondensed) compound graphs or organizing the SCCs in a hierarchical manner (Roditty and Zwick, 2008). In our implementation, we resort to storing the uncondensed compound graphs along with the condensed compound graphs G_i^C, albeit approaches like (Roditty and Zwick, 2008) may be employed for further optimizations.

A deletion of a local edge (u, v) in partition i is processed over the condensed compound graph G_i^C as follows. If the vertices u, v belong to the same SCC, then we expand this SCC into its original edges and reconnect these edges to the remaining SCCs in G_i^C. Moreover, in case of deletions, some of the existing boundaries may not be connected anymore. We identify such pairs of boundaries and communicate these to the other slaves. After receiving this list of deleted boundary edges, we reconstruct the local compound graphs G_j^C (for $j \neq i$) analogously to the insertion case. If, on the other hand, the vertices u, v belong to two different SCCs, then we expand both of them.

Here, the worst-case time complexity to maintain the local boundary edges is $\mathcal{O}((|V_i| + |E_i|) \cdot |I_i| \cdot |O_i|)$, which is the same as for rebuilding the local boundary graphs (see Section 3.5.2.1). The new compound graphs are condensed via SCC computation, whose worst-case time complexity is $\mathcal{O}(n' + m')$, where n' and m' again are the numbers of vertices and edges in the new G_i^C's.

3.6 Iterative Approaches

Next, we discuss two iterative-based solutions to process DSR queries using a general purpose distributed graph processing frameworks. We start with the popular vertex-centric approach (Section 3.6.1) and then discuss a more efficient graph-centric approach (Section 3.6.2) to process DSR queries. Furthermore, we extend the graph-centric approach by leveraging the equivalence-sets to further improve the performance.

3.6.1 Vertex-Centric Approach

As discussed in Section 3.4.2, vertex-centric approaches follow an iterative model to perform operations on graphs and define a compute(·) function on each vertex. Called *supersteps*, each vertex v receives messages from its predecessors in a superstep i that are sent in previous superstep $(i-1)$, process the messages in compute(·) function, and sends the messages to v's successors in the next superstep $(i+1)$.

A DSR query $S \rightsquigarrow T$ is processed using the vertex-centric approach as follows. Each vertex v in the graph maintains a vector $v.R$, initialized to $\emptyset$, that holds the set of source vertices that are reachable to v. In superstep 0, each source vertex $s \in S$, update its corresponding vector $s.R$ to $s.R \cup \{s\}$, and sends $s.R$ as a message to its successors. In the next superstep, a vertex v receives and aggregates messages sent by its predecessors into a temporary vector $v.R'$. Vertex v, then sends the differential information, i.e., $v.R' - v.R$, to its successors and subsequently updates its vector $v.R$ to $v.R \cup v.R'$. If $v.R' - V.R = \emptyset$. Then vertex v is *halted* and *activated* only when v again receives a message from one of its predecessors. This process continues, and at any superstep i, if all the vertices are *halted*, then the process stops. The values of vertices $t \in T$, i.e., vector $t.R$ contains all the reachable sources s in S, which are then used to generate reachable pairs of the form (s, t).

Appendix A.1.1 provides an implementation of the aforedescribed approach in Giraph (Martella et al., 2015), a popular distributed graph processing framework.

3.6.2 Graph-Centric Approach

Analogously to vertex-centric approaches, graph-centric approaches follow an iterative model to perform graph operations. On the contrary, graph-centric approaches considers a subgraph (or partition) in each superstep rather than an individual vertex. A compute(·) function is defined for each subgraph. In each superstep, zero or more vertices in a subgraph G_i (a partition of G) receive messages from its predecessors that are located in another subgraph G_j $(j \neq i)$. The received messages are then internally communicated to all the vertices in G_i, thus updating their corresponding vertex values. At the end of the superstep, only the vertices that have successors, that belong to another subgraph, send messages which are subsequently processed in the next superstep.

DSR queries using the graph-centric approach are processed as follows. First, using the graph partitioning $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ information, a DSR query $S \rightsquigarrow T$ is partitioned into k subqueries: $S_1 \rightsquigarrow T_1$, $S_2 \rightsquigarrow T_2$, ..., $S_k \rightsquigarrow T_k$. Additionally, like in the vertex-centric approach, each vertex maintains a vector $v.R$, initialized to $\emptyset$, which holds the set of source vertices that are reachable to v. In superstep 0, on subgraph G_i, we invoke a local reachability evaluation starting from all source vertices in S_i. A local reachability evaluation, using on a typical BFS/DFS traversal strategy (Cormen et al., 2009), finds, for all the vertices v in V_i, the list of sources vertices in S_i that are reachable to v, i.e., $v.R = \{s \mid s \in S_i \text{ and } s \rightsquigarrow v := \textit{true}\}$. At

the end of the local reachability evaluation, all vertices $v \in V_i$ at each partition, send the vector $v.R$ as message to their remote successors. In the subsequent superstep, vertices receive messages from their remote predecessors and store them in a temporary vector $v.R'$. If $(v.R' - v.R) \neq \emptyset$, v is treated as a new source and the local reachability evaluation is repeated. If, at any superstep, there are no new sources, the computation on the partition G_i is halted. If all the partitions are halted, the process is exited and the values of vertices $t \in T$, i.e., the vector $t.R$ contains all the reachable sources s in S, which are then used to generate reachable pairs of the form (s, t)

We implemented the above graph-centric approach in Giraph (Martella et al., 2015) by exposing the partition information similar to the implementation of Giraph++ (Tian et al., 2013). Appendix A.1.2 provides the implementation details.

Equivalence-Sets Optimization. In addition, we extended the graph-centric approach further to exploit the equivalence-sets optimization proposed for non-iterative approaches (see Section 3.5.2.1). More specifically, we precompute and group the boundary vertices of each partition, in to equivalence-sets groups. Then the input graph is modified as follows. For each vertex v, say belonging to partition i, i.e., $v \in V_i$, we augment a vector EQ_v, an equivalence set of vertices for the corresponding neigbors of v. The vector EQ_v is built as follows, for each remote successor r of v, r'(=rep[r]) is added to EQ_v, where r' is a representative member and r, r' both belong to the same equivalent set. For instance, consider a vertex v and its list of local successors $(l_1, l_2, \ldots, l_x)$ and remote successors $(r_1, r_2, \ldots, r_y)$. The adjacency list representation of v, i.e.,

$$v \rightarrow l_1, l_2, \ldots, l_x, r_1, r_2, \ldots, r_y$$

is converted to

$$v \rightarrow l_1, l_2, \ldots, l_x, r'_1, r'_2, \ldots, r'_z$$

where, r'_p is a representative member of r_q, $p \leq z$, $q \leq y$, and $z \leq y$.

DSR query processing with equivalence sets is much alike the one without equivalence sets except that instead of sending messages to the remote successors, we send messages to the remote representative members (r'). Appendix A.1.3 provides the implementation details of the DSR query processing that leverages the equivalence sets optimization in Giraph++, which we coin as "Giraph++wEq" in our experiments.

3.7 Evaluation

We next present a detailed empirical evaluation of our proposed index processing and updating strategies for DSR queries.

DSR Implementation. We implemented our DSR approach in TriAD engine (which will be discussed in Chapter 4). TriAD follows a master-slave architecture and uses MPICH2 asynchronous communication protocol for communication

among slaves. We added *Graph Parser* module to the master node of the TriAD architecture to parse graph datasets represented in edge list format 2.1.1.7. We used the TriAD custom partitioner to partition the input graph into a locality based partitioning using METIS. At each slave, we used the triple index structures (SPO, OPS) to store the compound graphs, local graphs and any intermediate graphs during preprocessing step. In addition, off-the-shelf centralized indexes maintain their own custom indexes in our implementation in TriAD. For more details about the architecture, please refer to Section 4.3.

Variants & Competitors. Specifically, we compare the following approaches:

- our DSR approach with a static reachability index implemented in TriAD (coined "*TriAD*"), as described in Section 3.5.2;
- a naïve enumeration of all pairs of source and target vertices (coined "*TriAD-Naïve*"), as described in Section 3.5.1.1;
- a generalization of the algorithm described by Fan et al. (coined "*TriAD-Fan*") (Fan et al., 2012) to sets of source and target vertices, as described in Section 3.5.1.2;
- an implementation of DSR queries in Apache Giraph (coined *"Giraph"*) (Martella et al., 2015), as depicted in Appendix A.1.1;
- an implementation of DSR queries in Giraph++[2] (coined *"Giraph++"*) (Tian et al., 2013), as depicted in Appendix A.1.2;
- an extended version of Giraph++ with equivalence sets (coined "*Giraph++wEq"*), as depicted in Appendix A.1.3.

Further, for our DSR approach, we report the results in combination with the following local reachability indexes:

- a plain depth-first-search (DFS) strategy which requires no additional index structures except for those described in Section 3.5.2 (coined *"TriAD-DFS"*);
- the multi-source, breath-first-search (MS-BFS) algorithm described by Then et al. (Then et al., 2014) which also requires no additional index structures except for those described in Section 3.5.2 (coined *"TriAD-MSBFS"*);
- using FERRARI (Seufert et al., 2013) as a local reachability index that is generated on top of the compound graphs described in Section 3.5.2 (coined *"TriAD-FERRARI"*).

Unless stated otherwise, we report the combination of our DSR index with DFS as the default local search strategy.

[2] https://issues.apache.org/jira/browse/GIRAPH-818

Small Graphs	|V|	|E|	Large Graphs	|V|	|E|
Amazon	0.4M	3.3M	LiveJ-68M	4.8M	68.9M
BerkStan	0.7M	7.6M	Twitter-1.4B	41.7M	1,468.4M
Google	0.9M	5.1M	Freebase-500M	97.3M	499.9M
NotreDame	0.3M	1.5M	Freebase-1B	156.6M	999.9M
Stanford	0.3M	2.3M	LUBM-500M	115.6M	500.0M
LiveJ-20M	2.5M	20.0M	LUBM-1B	222.2M	961.4M

Table 3.1: Graph datasets and sizes

Datasets. The list of graph collections we consider for our evaluation is shown in Table 3.1. All the smaller graphs (including the two Live Journal versions) are obtained from the Stanford Snap[3] project. For our evaluation over larger graphs, we used the real-world Freebase[4] and Twitter[5] snapshots. In addition, we also used the widely popular synthetic LUBM RDF benchmark, which we generated using the UBA 1.7[6] data generator.

General Setup. We used MPICH2-1.4.1 for communication among the compute nodes, using a cluster of 10 nodes which are connected via a 10GBit LAN. Each node has 64GB of RAM and an Intel X5650@2.67GHz quadcore CPU with HT enabled. Giraph and its variants are implemented in Java, where we used Hadoop v0.20 for running Giraph (Martella et al., 2015). Appendix A.1 depicts our actual implementation of DSR queries for the three Giraph variants.

3.7.1 Efficiency

For this experiment, we considered several real-world data graphs (both small and large) and the synthetic LUBM graph. We fixed the compute cluster to 6 nodes (i.e., to 5 slaves and 1 master). We randomly selected 10 source and 10 target vertices from all datasets (except LUBM-1B) as queries, thus resulting in 100 reachability comparisons. For LUBM-1B, which is very sparsely connected, we randomly chose 1,000 sources and 1,000 targets, of which only 131 pairs turned out to be reachable.

Table 3.2 shows the maximum (i.e., per node) uncondensed ("*Original*") and SCC-condensed ("*DAG*") compound-graph sizes as well as the total byte size ("*Size*") for our TriAD in comparison to the dependency-graph sizes for TriAD-Fan and TriAD-Naïve. In TriAD-Fan, for a given DSR query $S \rightsquigarrow T$, all of S and T are used "at once" to generate the dependency graph. In TriAD-Naïve, which generates the dependency graph per (s, t) pair, the sizes represented in Table 3.2 are the average dependency-graph sizes over 100 pairs. SCC compression, which is not feasible for the dynamically generated dependency graph, drastically reduces the

[3] http://snap.stanford.edu
[4] http://freebase.com
[5] http://an.kaist.ac.kr/traces/WWW2010.html
[6] http://swat.cse.lehigh.edu/projects/lubm/

Graphs	TriAD Compound graph Original (#edges)	DAG (#edges)	Size (MB)	TriAD-Fan Dep.graph (#edges)	TriAD-Naïve Dep.graph (#edges)
Amazon	1.0M	34.7K	206	622.3M	622.2M
BerkStan	2.1M	0.5M	383	2.2M	2.1M
Google	1.2M	0.2M	302	43.6M	43.6M
NotreDame	0.8M	68.9K	123	4.7M	4.7M
Stanford	0.8M	41.2K	122	1.2M	1.2M
LiveJ-20M	13.7M	1.0M	1,553	861.4M	n/a
LiveJ-68M	44.1M	0.3M	928	n/a	n/a
Freebase-1B	460.4M	241.6M	64,141	n/a	n/a
Twitter-1.4B	1,285.0M	8.2M	20,053	n/a	n/a
LUBM-1B	891.8M	891.3M	107,608	n/a	n/a

Table 3.2: Index sizes for DSR variants implemented in TriAD

sizes of the compound graphs stored at each slave. For example, for the Twitter-1.4B graph, which is highly connected, the size of each compound graph stored at the slaves initially is comparable to the size of the original graph. Applying SCC compression condenses these graphs by a factor of about 150. Also for LiveJ-68M, the SCC compression leads to a much smaller DAG size than for LiveJ-20M, such that our query times are actually lower for LiveJ-68M than for LiveJ-20M.

Table 3.3 shows our query-processing results. For both the small and large graphs, our approach clearly demonstrates efficiency improvements of several orders of magnitude when compared to the three Giraph variants as well as to TriAD-Fan and TriAD-Naïve. Even with a single round of communication, TriAD-Fan and TriAD-Naïve exhibit a considerable overhead in generating the dynamic dependency graph for each query. Specifically, we observed that for LiveJ-20M, TriAD-Fan generates a dependency graph of about 861 million edges even when the data graph is partitioned by METIS (Karypis and Kumar, 1998) in order to minimize the cut. Our TriAD approach, which benefits from the optimizations we apply when constructing the compound graphs, avoids the repeated generation of a large dependency graph at the master node and therefore is able to achieve very significant performance gains over TriAD-Fan and TriAD-Naïve. Giraph++ and Giraph++wEq, on the other hand, perform better than the native Giraph implementation, as the former benefit from their local updates of neighboring vertices. This drastically reduced the number of supersteps required for processing a set-reachability query. The equivalence-sets optimization for Giraph++wEq further reduced the communication but only marginally improved the query processing times.

3.7.2 Scalability

Next, we evaluated our approach in comparison to the Giraph variants under both strong and weak scaling. We dropped TriAD-Fan and TriAD-Naïve from these

Graphs	*Indexing Time* TriAD	*Query Size* |S|	|T|	*Query Time* TriAD	Giraph++	Giraph++wEq	Giraph	TriAD-Fan	TriAD-Naïve
(a) Small Graphs (times in seconds)									
Amazon	2.380	10	10	**0.008**	12.250	11.348	55.034	72.111	855.159
BerkStan	3.048	10	10	**0.009**	44.180	5.680	779.006	2.219	38.036
Google	3.194	10	10	**0.060**	60.154	11.426	53.614	25.210	114.078
NotreDame	1.089	10	10	**0.057**	11.085	12.320	94.787	1.800	50.598
Stanford	1.511	10	10	**0.008**	7.808	8.922	341.976	0.468	6.211
LiveJ-20M	44.536	10	10	**0.227**	19.888	19.262	28.075	521.569	n/a
(b) Large Graphs (times in seconds)									
LiveJ-68M	144.981	10	10	**0.090**	64.728	61.940	93.253	n/a	n/a
Freebase-1B	1,938.670	10	10	**67.849**	1,371.423	1,014.442	1,857.124	n/a	n/a
Twitter-1.4B	6,963.730	10	10	**1.119**	3,065.483	3,046.450	n/a	n/a	n/a
LUBM-1B	2,083.190	1,000	1,000	**1.340**	146.864	142.142	154.407	n/a	n/a

Table 3.3: Efficiency evaluation (indexing and query times) of DSR approaches for small and large graphs

comparisons, as they could not scale to the larger graphs anymore. We considered LiveJ, Freebase, Twitter and the synthetic LUBM graph for our scalability evaluation. We used METIS to partition the graphs and distributed the partitions to up to 10 compute nodes (one of which used as master), and we considered 10 random source and 10 random target vertices as queries. Figures 3.7(d)(h)(l)(p) also show the robustness of our approach with respect to larger query sets.

- *Live Journal:* Figures 3.7(a)-(d) depict the scalability evaluation of our approach and the Giraph variants for LiveJ-68M. Figure 3.7(a) shows the results for a strong scaling. Here, we can observe that TriAD scales very well and performs significantly better than the Giraph variants. We also observe that Giraph^{++} and Giraph^{++}wEq perform slightly better than Giraph by leveraging the node locality and equivalence-sets optimization, respectively. This observation is confirmed further by Figure 3.7(b), where Giraph communicates about two orders of magnitude more messages compared to Giraph^{++}and Giraph^{++}wEq. Figure 3.7(c) shows the weak scalability for the 10 by 10 DSR queries.

- *Freebase:* The scalability results for Freebase-1B are shown in Figures 3.7(e)-(h). We can observe that our approach scales well on average, even when the graph sometimes is rather unevenly partitioned as a result of using METIS. This uneven partitioning also is the reason for the runtime increase from 7 to 8 slaves. By leveraging node locality in Giraph^{++} and the equivalence-sets optimization in Giraph^{++}wEq, both approaches continue to show performance gains over Giraph, but this time with a more visible difference in the communication costs among the three variants as shown in Figure 3.7(f). Figure 3.7(g) shows the weak scalability.

- *Twitter:* We next performed a similar scalability evaluation over Twitter, consisting of more than 1.4 billion edges and more than 41 million vertices. METIS this time resulted in a very skewed partitioning, with one partition containing almost half of the edges and almost one third of the edges being cut edges. This constituted a challenge for our approach, because we compute the boundary graph along with the cut edges. However, since vertices in Twitter are densely connected, the resulting compound graphs at all slaves can very well be condensed using SCC compression, which led to very small graph indexes at the slaves (with a compression factor of more than 150). Without this optimization, Giraph was able to load the Twitter graph but failed to process the set-reachability query, returning an "out of memory" exception. The strong and weak scalability of our approach and the Giraph variants are shown in Figures 3.7(i) and 3.7(k).

- *LUBM:* As the final scalability experiment, we considered the synthetic LUBM-1B dataset whose results are shown Figures 3.7(m)-3.7(p). Most of the RDF-based LUBM graph is acyclic and sparsely connected. Thus, our SCC condensation for the compound graphs has very low effect on the overall query processing. Figure 3.7(m) shows the strong scalability of our approach versus the

Giraph variants. Figure 3.7(n) shows the communication costs for different variants of Giraph and our approach. Again, our DSR approach (TriAD), which evaluates a set-reachability query in a single round of communication, exchanges a very low amount of messages compared to the iterative Giraph variants.

3.7.3 Updates

We considered the six smaller graphs plus the LiveJ-68M dataset (see Table 3.1) for updates. We distinguish two principal kinds of incremental update workloads, which we call *bulk updates* and *progressive updates*, respectively.

- For bulk insertions, we start with 60% of randomly chosen edges of the original graph and then increment the graph by 5% of the remaining edges, until we reach the original graph. For bulk deletions, we start with the original graph and decrement the graph in 5% steps.
- For progressive insertions, we randomly pick x% (say, 5%) of edges from the original graph and measure the time to insert these into an index built over the remaining $(100 - x)$% (say, 95%) of edges. We increment x in 5% steps. For progressive deletions, we decrement the original graph by a progressive amount of edges.

We used the same queries as described in Section 3.7.1 to measure the effect of these update steps on the query times.

Insertions. Figures 3.8(a)(e) show the update and respective query times for our bulk insertions. It can be observed that the time needed for bulk insertions remains almost constant for each 5% step. Query performance, which depends on the final DAG size, however varied considerably with each update. Next, we considered progressive insertions. From Figure 3.8(b), it can be clearly seen that the update times are only a fraction of the total rebuild time (see Table 3.2). Query performance, shown in Figure 3.8(f), increased marginally at each step, as expected, although also this depends on the final DAG size as a result of the update operation.

Deletions. Deletions are generally more costly in our setting and took almost the same time as building the index from scratch (see Table 3.2) for both bulk and progressive updates. Figures 3.8(c)(g) depict the update and respective query times for bulk deletions. While deletion times show a downward trend, query times tend to increase as the graphs become more sparsely connected, thus leading to larger DAG sizes. This is especially visible for the LiveJ-68M dataset. For the case of progressive deletions, as shown in Figures 3.8(d)(h), we observe similar trends in terms of update and query times.

3.7.4 Parameters

A. Local Reachability Indexes. We next measured our DSR approach (TriAD) in conjunction with three centralized strategies. For all three cases, we condense

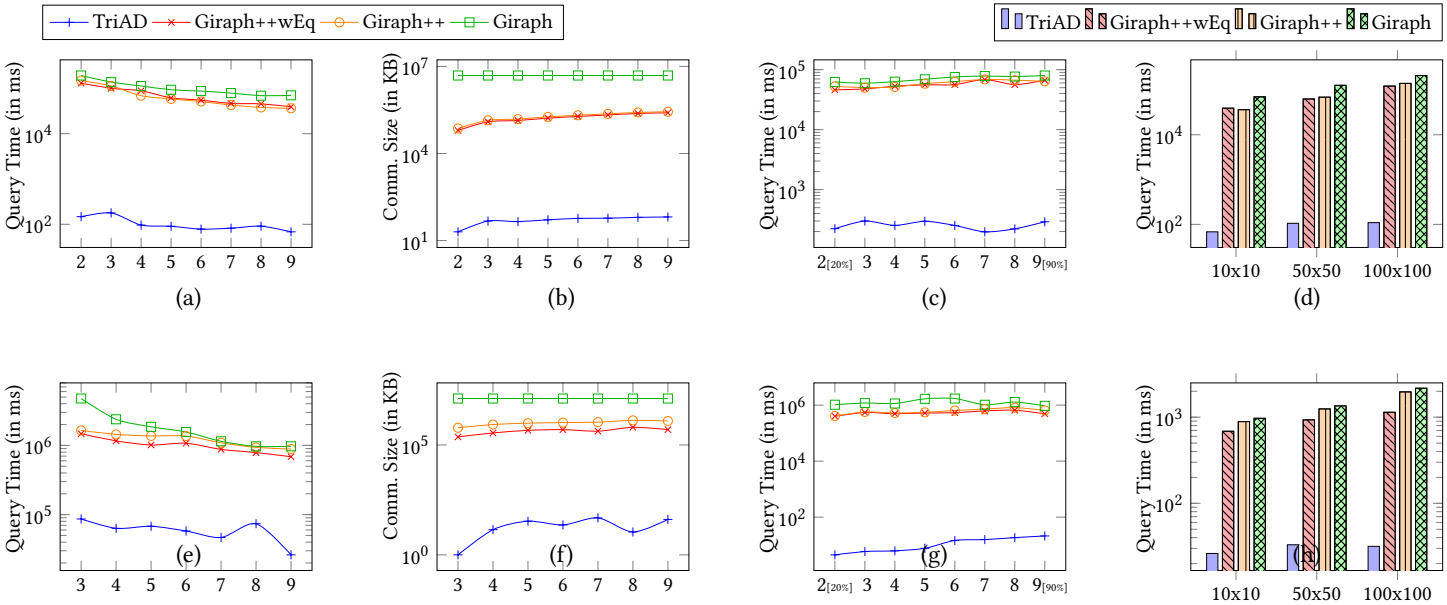


Figure 3.6: Scalability evaluation for LiveJ-68M (a-d) and Freebase-1B (e-h)

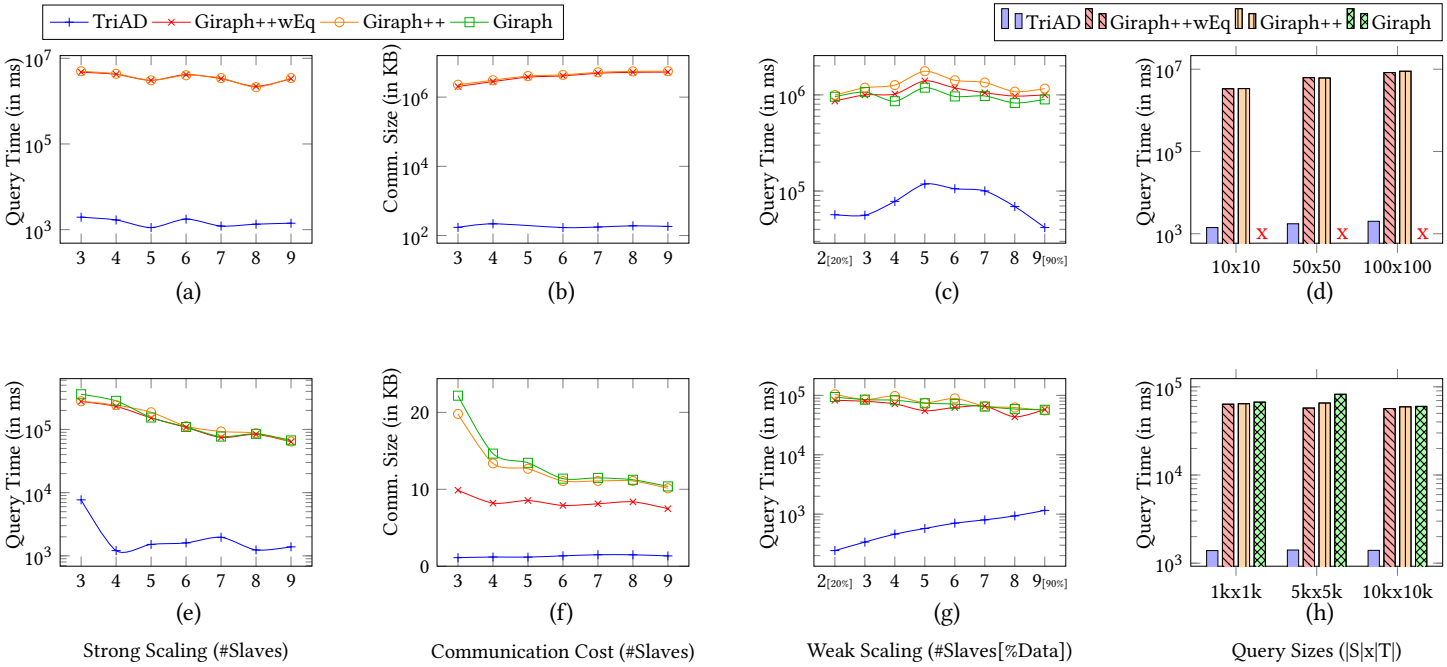


Figure 3.7: Scalability evaluation for Twitter-1.4B (a-d) and LUBM-1B (e-h)

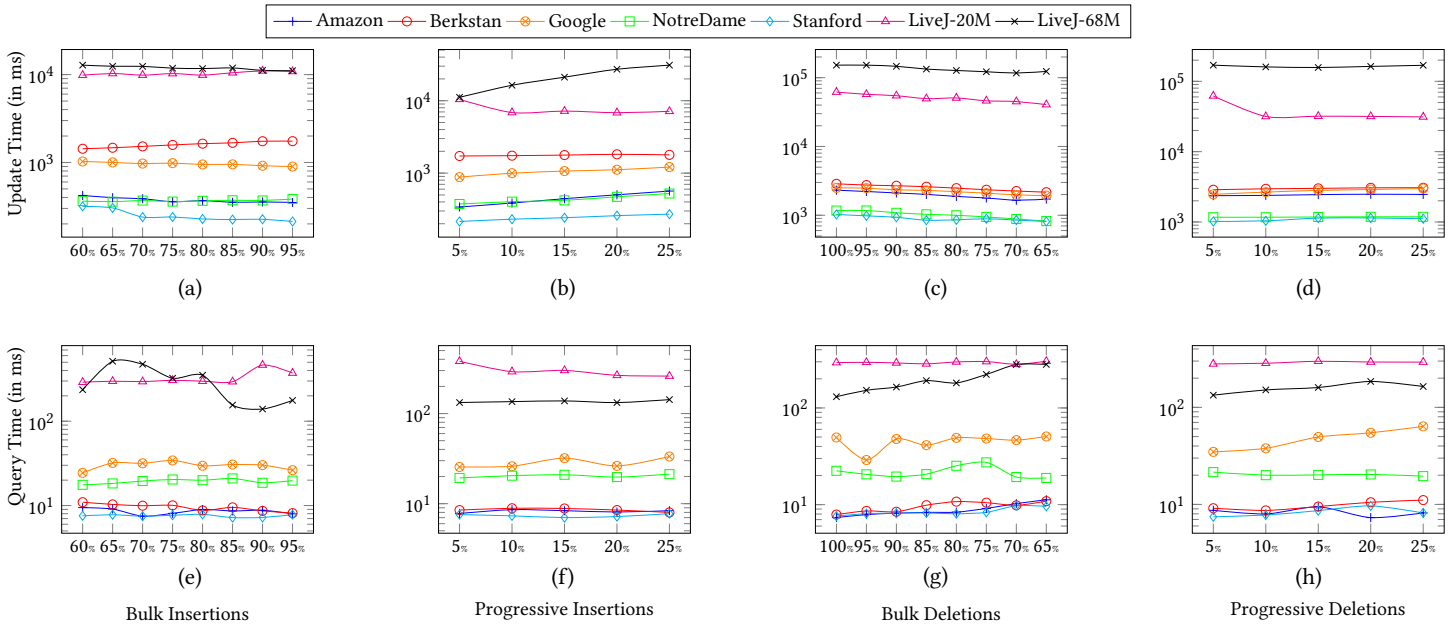
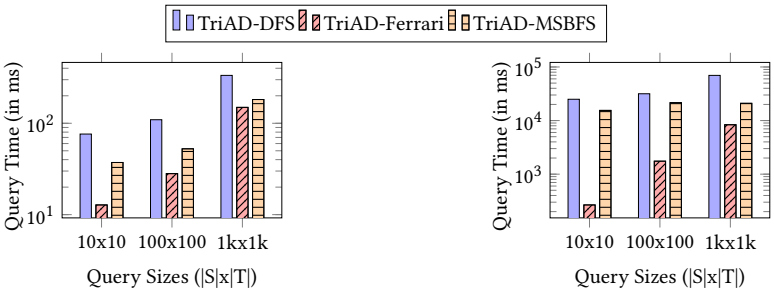


Figure 3.8: Update evaluation (both insertions and deletions) for various graph collections



(a) LiveJ-68M (b) Freebase-1B

Figure 3.9: Comparison of local reachability indexes

the local compound graphs via computing SCCs. TriAD-DFS uses a standard DFS strategy (Cormen et al., 2009) for processing a DSR query, where no additional index is built over the compound graphs. For each source s and target t in the given DSR query, we perform a DFS to evaluate the reachability $s \rightsquigarrow t$. MS-BFS (Then et al., 2014) caches, for each vertex v that is visited during a graph traversal, the reachability of v to all its targets. Thus, if v is another source in the query, we avoid recomputing the reachability and thus save a graph traversal. FERRARI (Seufert et al., 2013) finally provides a tunable tradeoff between index size and query performance. We set both the number of intervals per vertex and the number of seed vertices to 1,000.

Figure 3.9 shows the effects on query performance when using different local search strategies. For this experiment, we again considered 10 nodes of which one was the master node. We used two real-world datasets, LiveJ-68M and Freebase-1B, for this evaluation. We considered different query sizes to demonstrate the strengths and weaknesses of the three approaches. Figure 3.9(a) shows the results for LiveJ-68M. We can observe that TriAD-DFS takes longer compared to the other two baselines as it requires one graph traversal (in the worst case) for each source. TriAD-FERRARI, with its compact reachability index, demonstrates significant performance gains over the other two baselines for different query sizes. On the other hand, for large query sizes, the TriAD-MSBFS approach benefits from its memoization and less redundant graph traversal and tends to close the gap to FERRARI. The three strategies show similar trends also for the larger Freebase-1B dataset (see Figure 3.9(b)).

B. Equivalence-Sets Optimization. By computing equivalence sets among in- and out-boundaries, we are able to reduce both the boundary-graph sizes as well as the number of reachability computations required per slave. Table 3.4 shows the benefits of this optimization. Figure 3.10 shows a comparison for the query performance and communication costs with and without equivalence sets in Giraph.

C. Partitioning Strategy. We next considered the effect of the partitioning strategy on the performance of our approach. For this, we used two partitioning strategies: a random hash-partitioning and METIS (Karypis and Kumar, 1998). We used

Graph	Query Time (times in sec.) Non-Opt.	Opt.	Boundary-Graph Sizes (#forward; #backward) Non-Opt.	Opt.
Amazon	0.101	0.008	900; 530	18; 5
BerkStan	0.157	0.110	20,750; 49,462	3,916; 4,981
Google	1.416	1.003	47,822; 98,955	3,759; 6,287
NotreDame	1.085	0.768	16,771; 6,899	2,481; 37
Stanford	0.061	0.038	5,411; 13,942	183; 475

Table 3.4: Equivalence-sets optimization in TriAD

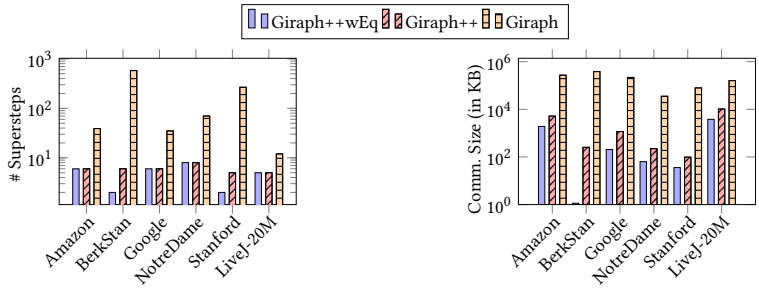


Figure 3.10: Equivalence-sets optimization in Giraph

a cluster of 6 nodes (one of which was dedicated as the master) and evaluated the strategies using a set-reachability query with 10 sources and 10 targets. Table 3.5 shows the performance comparison of the two partitioning strategies over several real-world graphs. It can be clearly observed that the choice of the partitioning strategy influences the performance of our approach. Hash partitioning (i.e., "random sharding") usually results in a drastic increase of cut edges and thus in a lower query performance. METIS partitioning in turn helps in minimizing this cut, which significantly improves the query performance.

Graph	Partitioning (query times in sec.) Hash	METIS
Amazon	0.009	0.008
BerkStan	0.016	0.009
Google	0.330	0.060
NotreDame	0.085	0.057
Stanford	0.009	0.008
LiveJ-20	0.524	0.227
LiveJ-68	0.188	0.090

Table 3.5: Impact of hash vs. METIS partitioning

3.7.5 Applications

A. SPARQL 1.1 with Property Paths. For this experiment, we considered the LUBM-500M and Freebase-500M datasets (both in RDF format). We augmented a distributed RDF store (Gurajada et al., 2014a) with our DSR approach (TriAD) by modifying its query processor to handle property paths via our new index structures. To evaluate the performance of our approach in processing SPARQL 1.1 queries, we compared against the commercial Virtuoso RDF store (Erling and

Mikhailov, 2010). The results are shown in Table 5.1, while the customized SPARQL queries we used for this evaluation are depicted in Appendix A.2

(a) LUBM-500M *(query times in sec.)*					
	#Slaves	**L1**	**L2**	**L3**	**Geo. Mean**
TriAD	1	6.437	0.331	42.681	4.497
TriAD	5	**1.250**	**0.162**	**8.516**	**1.199**
Virtuoso (cold)	1	10.050	12.624	57.776	19.425
Virtuoso (warm)	1	4.963	5.452	56.603	11.527
(b) Freebase-500M *(query times in sec.)*					
	#Slaves	**F1**	**F2**	**F3**	**Geo. Mean**
TriAD	1	1.084	1.568	0.677	1.048
TriAD	5	**0.356**	0.642	**0.423**	0.459
Virtuoso (cold)	1	6.590	4.112	13.809	7.206
Virtuoso (warm)	1	1.196	**0.002**	5.601	**0.238**

Table 3.6: SPARQL 1.1 queries with property paths

B. Social-Network Communities. As another DSR application, we detected connectivities among communities in a social network. This problem is a basic step in many graph-analytics tasks. That is, given two communities C_1 and C_2 together with a set of representative members for each community $S \subseteq C_1$, $T \subseteq C_2$, find all pairs s, t, with $s \in S$ and $t \in T$, such that $s \rightsquigarrow t$. We considered two social network datasets, LiveJ-68M and Twitter-1.4B, for this experiment. We employed the iterative community-detection algorithm by Blondel et al. (Blondel et al., 2008) to identify communities. We then randomly picked two communities, and from each we picked 10 to 1,000 members as representatives. We then ran our DSR approach (TriAD) to identify all reachable pairs among these representatives. The results are shown in Table 3.7.

	LiveJ-68M *#Communities: 5,032*		**Twitter-1.4B** *#Communities: 17,121*	
Query Size (|S|x|T|)	**Query Time** *(in sec.)*	**#Pairs**	**Query Time** *(in sec.)*	**#Pairs**
10x10	0.065	81	1.339	63
100x100	0.164	8,184	2.476	8,526
1kx1k	0.717	784,947	10.175	712,725

Table 3.7: Community connectedness using TriAD

3.7.6 Summary of Results

Our experiments confirm the significantly improved efficiency (with a gain in query times of several orders of magnitude) of our DSR index implemented in TriAD compared to iterative approaches such as Apache Giraph and variants of (Fan et al., 2012).

Moreover, we are also able to demonstrate the good update support of our in-

dex structure, which—in particular for insertions—behaves much better in practice than suggested by the worst-case bounds we provide in Section 3.5.2.5. We believe that insertions are the much more likely use-case for managing large, dynamic graphs (e.g., Twitter streams), while deletions, which are costly to handle for any kind of graph-compression technique, are much more uncommon in practice. Also, there is also hardly any support for updates in the centralized approaches (such as (Seufert et al., 2013)), which restricts our local search strategy to a simple DFS or BFS in this case. Further experiments demonstrate the robustness of our approach under different parameters and show its viability for various large-scale graph-analytics tasks.

3.8 Summary

In this chapter, we investigated a generalized form of the well-known reachability problem in directed graphs, which (1) considers both sets of source and target vertices as queries, and (2) allows the underlying graph to be partitioned and hence be distributed across multiple compute nodes. The DSR problem is a basic building block and thus has a plethora of applications in graph analytics and query-processing tasks. Thus, we presented as our core contribution, an efficient and scalable framework for processing DSR queries and also studied its formal properties. By precomputing and materializing the reachability information among vertices along the cut of a partitioned data graph, our approach is guaranteed to require at most one round of communication among the compute nodes to resolve any DSR query. Our approach exhibits a very good support for incremental vertex and edge insertions, while our current implementation resorts to just a basic support for respective deletions. Moreover, any state-of-the-art centralized reachability index may be applied to the local graph partitions to further accelerate query-processing times. In addition, we also discussed DSR query processing in iterative approaches, which though clearly less efficient for selective queries are scalable to large graphs. Our evaluation over both real-world and synthetic graphs and in comparison to iterative approaches also empirically demonstrated the viability of our approach.

Chapter **4**

Basic Graph Patterns

Basic graph patterns (BGP) is an important querying model belonging to the class of pattern matching queries (see Section 2.1.3.2). A graph pattern (or simply a pattern) is a basic unit in BGP queries, and typically constitutes a triple of the form $\langle u, e, v \rangle$, where each of u, e, v can be either a *variable* or a *constant*. An answer to a BGP query is the set of all vertex bindings to the variables in the query, such that each vertex binding along with the constants collectively form a subgraph of an input graph. Because of its varied importance in many application scenarios, BGP queries are a popular choice of querying graphs and are predominantly supported by many graph query languages such as SPARQL, Cypher, etc. and by current graph database systems.

In this chapter, we look at the distributed processing of *BGP queries* on labeled directed multi-graphs. Specifically, we focus on conjunctive BGP queries, where the only allowed set operation is the "conjunction (AND)" among the patterns. We consider labeled directed multi-graphs and conjunctive BGP queries as the underlying data and query models of our choice, as they provide enough expressivity to represent many existing real-world datasets and query needs. Furthermore, with the wide adoption of semantic knowledge graphs across multiple domains, we chose the Resource Description Framework (RDF) and SPARQL 1.0, both W3C recommendations, as the representative languages for our data and query model. We propose a novel distributed architecture and a working prototype system, coined "TriAD (for *Triple Asynchronous and Distributed*)", that can handle conjunctive BGP queries in an efficient and scalable manner. The TriAD system is built on the duality of graph-based and relational concepts to process BGP queries. With our empirical analysis on multiple real-world and synthetic datasets, TriAD performs significantly better than the existing state-of-the-art systems.

4.1 Introduction

4.1.1 Motivation

Along side connectivity queries discussed in Chapter. 3, BGP queries belong to an important querying model that has profound significance in many applications. These queries come with varying degrees of complexity from being as simple as selecting an edge to a more complex queries like matching a subgraph in an input graph. A triple of the form $\langle u, e, v \rangle$ constitutes a typical pattern in BGP queries, where each of u, e, v can be either a variable or constant. Processing a single triple pattern over an input graph resolves to a selection of edges whose labels match the constants in the given query. For instance, processing a pattern $\langle$*person*, bornIn, Honolulu$\rangle$ matches the directed edges such as the edge (Barack_Obama, bornIn, Honolulu) in the input graph, and the vertex label "Barack_Obama" is one of the instances for the query variable *person*. In reality, BGP queries often comprise of a set of patterns, which together form a query graph. Processing a multi-pattern BGP query resolves to finding all the subgraphs of an input graph that are isomorphic to the query graph. Chandra et al. (Chandra and Merlin, 1977) discuss an equivalency between the graph isomorphism and conjunctive queries on databases and further states that the processing a conjunctive queries has polynomial data complexity and non-polynomial expression (query) complexity. We further refer the reader to (Chandra and Merlin, 1977; Vardi, 1982) for more details on the complexity of processing conjunctive queries in a relational model, which we rely on for this work.

Applications. Querying Resource Description Framework (RDF) graphs is one of the many interesting applications where BGP queries are frequently used. RDF is a W3C recommended language for representing linked data on the web. Most knowledge graphs, biological datasets, and to some extent many social networks can be expressed as an RDF graph. Some of the real world examples include knowledge graphs such as DBpedia (Bizer et al., 2009), YAGO (Suchanek et al., 2007), Google's Knowledge Graph (Singhal, 2012), Microsoft Bing's Satori Knowledge Base (Qian, 2013), etc., biological datasets such as Uniprot (EMBL et al., 2013), Bio2RDF (Belleau et al., 2008), and social networks such as LiveJournal [1] support RDF representation of user profiles and their relationships. SPARQL (Prud'hommeaux et al., 2013), a W3C recommended, is the de facto language for querying RDF graphs. With the increasing number of both commercial and non-commercial organizations, which actively publish and query RDF data, the amount and diversity of openly available RDF repositories is growing at an unprecedented pace. This attracted a lot of research attention in development of efficient and scalable RDF stores, which is also the focus of the current chapter.

Other applications that rely on BGP query model include finding network motifs in complex networks (Milo et al., 2002), analysis in biological networks (Eck-

[1] snap.stanford.edu

man and Brown, 2006; Aittokallio and Schwikowski, 2006), network traffic analysis (Natarajan, 2000), graph simulation (Fan et al., 2011, 2014b), etc.

4.1.2 State-of-the-art

In this section, we explore some of the prior state-of-the-art systems that existed before our approach and support BGP queries. Specifically, we focus on RDF-based systems as they extensively support SPARQL query language which is largely based on the BGP querying model.

In response to the explosion of RDF data that is available on both the surface and the deep web, much research effort has been invested recently in the development of scalable, both centralized and distributed, techniques for indexing RDF data and for processing SPARQL queries.

Centralized Architectures. Among the centralized approaches, native RDF stores like Jena, Sesame, HexaStore (Weiss et al., 2008), SW-Store (Abadi et al., 2009), MonetDB-RDF (Sidirourgos et al., 2008), RDF-3X (Neumann and Weikum, 2010a,b), BitMat (Atre et al., 2010), gStore (Zou et al., 2011), and TripleBit (Yuan et al., 2013) have been carefully designed to keep pace with the growing scale of RDF collections. Efficient centralized architectures employ various forms of techniques to accelerate query processing and reduce database foot print. These include multi-permutation indexing (Neumann and Weikum, 2010a; Weiss et al., 2008) to facilitate low-cost merge join operations, vertical partitioning schemes (Abadi et al., 2009; Sidirourgos et al., 2008) to reduce the look up and join costs, and sophisticated bit encoding schemes (Atre et al., 2010; Yuan et al., 2013; Zou et al., 2011) to keep large portions of the index in main memory. Moreover, gStore (Zou et al., 2011) further relies on a novel hierarchical synopsis index structures to efficiently process BGP queries.

Shared-nothing Distributed Architectures. With the increasing popularity of shared-nothing architectures based on the MapReduce paradigm (Dean and Ghemawat, 2008), systems like SHARD (Rohloff and Schantz, 2011), (Huang et al., 2011) (an offspring of SW-Store, in the following referred to as "H-RDF-3X"), and EAGRE (Zhang et al., 2013) have been proposed for the scalable, distributed evaluation of SPARQL queries. While MapReduce allows for an easy adaptation of parallel (both Map- and Reduce-side (Lin and Dyer, 2010)) join algorithms on top of RDF-specific index structures, MapReduce frameworks are known to incur a non-negligible overhead due to their iterative, synchronous communication protocols and fault-tolerant job scheduling strategies. Even with the currently fastest, openly available MapReduce implementations, such as Hadoop++ (Dittrich et al., 2010) and Spark (Zaharia et al., 2010), this typically renders sub-second query response times for distributed joins infeasible. Systems like H-RDF-3X (Huang et al., 2011) and EAGRE (Zhang et al., 2013) thus make use of aggressive data replication to avoid iterative joins in Hadoop and to restrict query executions to the local RDF stores as much as possible. However, with longer-diameter queries or unexpected

workloads, there is no alternative to running joins via Hadoop, which often slows down query response times by two or more orders of magnitude.

Trinity.RDF (Zeng et al., 2013) is the first distributed RDF engine that employs a custom communication protocol based on the Message Passing Interface (MPI) standard (The MPI Forum, 1993). Instead of joining index lists, Trinity.RDF follows a graph-exploration strategy on top of a distributed, in-memory key-value store. Although Trinity.RDF is only single-threaded in its final join phase, it often allows for faster response times compared to the Hadoop-based RDF engines, especially when queries are selective and the graph exploration starts from just a few initial nodes. For non-selective queries, however, the generic architecture of Trinity.RDF, which is based on the Trinity graph engine (Shao et al., 2013), does not allow for the integration of parallel join techniques, as they are common, on the other hand, in Hadoop (Huang et al., 2011; Zhang et al., 2013).

Limitations with the State-of-the-art. Centralized approaches, though efficient, suffer from the scalability point of view. Because of their natural hardware limitations, centralized approaches cannot handle today's large RDF datasets, which typically comprises of more than 1 billion triples, while linked open data (LOD) [2] comprises of more than 130 billion triples [3]. On the other hand, shared-nothing distributed architectures, though scalable to large graphs, are not efficient in providing a sub-second query time performance which is crucial in many graph applications. In the next, we summarize our analysis of RDF systems by highlighting the following limitations that all existing, distributed RDF engines currently face.

1. Synchronous vs. Asynchronous Join Executions. Although Hadoop-based joins allow for the execution of multiple join operators in parallel, they need to *synchronize at each level of the query plan* before they can continue to process the plan with the next iteration of joins. These synchronization steps are heavily dominated by a few stragglers or imbalanced query plans.

2. Graph Exploration vs. Relational Joins. Parallel graph exploration is very efficient for queries that aim to select just a few subgraphs out of the RDF data graph. For a row-oriented output format, as it is required by the SPARQL 1.0 and 1.1 standards, *graph exploration is not sufficient to generate the final join results.* Thus, the parallel execution of joins remains a crucial factor for the efficiency and scalability of a SPARQL engine.

3. Sideways Information Passing vs. Join-ahead Pruning. Sideways information passing (SIP) is a run-time pruning technique employed to prune irrelevant tuples during query processing. Though effective in centralized systems, one of the main disadvantages of SIP is that it requires synchronization across multiple operators, which

[2] http://http://linkeddata.org
[3] http://stats.lod2.eu

significantly hinders the performance. On the other hand, join-ahead pruning, a compile time pruning technique, can be directly embedded inside the operators and provides support for asynchronous executions in a distributed setting.

Current Scenario. Recently, several efforts have been made, after our work (Gurajada et al., 2014a), in the distributed setting front. In (Peng et al., 2016), authors proposed a *partial evaluation and assembly* based strategy — popular in XML, graph simulation — for processing SPARQL queries on distributed RDF graphs. The queries are first evaluated locally at each partition to identify all partial matches, which are then later assembled using either a centralized or a distributed assembly framework. In another line of work, (Harbi et al., 2016) proposed an *adaptive hashing* strategy where the triples are adaptively partitioned based on a query workload. Authors of (Peng et al., 2016; Harbi et al., 2016) demonstrated empirical effectiveness of this approach over state-of-the-art systems including our approach.

4.1.3 Our Approach & Contributions

4.1.3.1 Our Approach

To mitigate the above problems and to process BGP queries in an efficient and scalable manner on labeled directed multi-graphs, such as RDF graphs, we propose a distributed system that is built on the concepts of relational and graph models. Specifically, we propose a novel, shared-nothing, main-memory architecture in combination with an asynchronous Message Passing (MPI et al., 2009) protocol. Our engine, coined TriAD (for "Triple-Asynchronous-Distributed"), aims at closing the gap between current relational shared-nothing Hadoop engines (Huang et al., 2011; Rohloff and Schantz, 2011; Zhang et al., 2013), on the one hand, and pure graph-based exploration strategies based on Message Passing (Shao et al., 2013; Zeng et al., 2013), on the other hand. TriAD is designed to achieve higher parallelism and less synchronization overhead during query executions than the Hadoop engines by adding an additional layer of multi-threading for entire paths of a query plan that can be executed in parallel. TriAD is the first distributed engine that employs asynchronous join executions (using a custom MPI protocol), which are coupled with a lightweight join-ahead pruning technique for the distributed processing of SPARQL queries. Specifically, TriAD builds on the following principles.

Parallel and Asynchronous Join Executions. TriAD in principle follows a classical master-slave architecture. During query execution, however, the slave nodes operate largely autonomously and communicate directly via *asynchronously exchanged messages* to run multiple join operators along the query plan in parallel. Our form of communication is asynchronous, because sibling execution paths of a query plan can be processed in a freely multi-threaded fashion and only need to be merged (i.e., be synchronized) once the intermediate results of entire such

execution paths are joined.

Distributed Indexes with Join-Ahead Pruning. Indexing is a primal factor in the success of efficient RDF systems such as RDF-3X, Hexastore, etc. On the similar lines, we employ six *permutation indexes* which are encoded into a distributed main-memory data structure that consists only of simple integer structs and vectors. Each index permutation list is first hash-partitioned ("sharded") according to its join key and then locally sorted in lexicographic order. Thus, even in its basic configuration without any multi-threaded execution of the query plan, TriAD can perform efficient, distributed merge-joins over the hash-partitioned permutation lists. In addition to the primary indexes, we employ a form of join-ahead pruning via an additional *summary graph* at the master node, in order to prune entire partitions of triples from the index lists that cannot contribute to the results of a given BGP query.

Distribution-Aware Query Optimizer. Similar to (Neumann and Weikum, 2008, 2010a), TriAD employs a bottom-up dynamic programming (DP) algorithm for *join-order optimization.* In addition to (Neumann and Weikum, 2008, 2010a), we also consider the locality of the index structures at the slave nodes, the shipping cost of intermediate join results, and the option to execute sibling paths of the query plan in a multi-threaded fashion, in order to determine the plan with the overall least cost estimate. This enables the optimizer to take much better advantage of the actual hardware capabilities, by taking the network latency and bandwidth, the CPU capacity for merging and hashing, and parallel query executions via multi-threading and distribution into account.

4.1.3.2 Contributions

We summarize the novel aspects of this chapter as follows.

- We investigate a new approach to the design of distributed engines with a goal to process BGP queries in an efficient and scalable manner and achieving a sub-second query performance. TriAD exploits both *intra-node multi-threading* and *asynchronous inter-node communication* to run multiple join operators of a query plan in a distributed and parallel way.
- We propose a novel form of graph summarization for labeled, directed multi-graphs, such as RDF, to facilitate *join-ahead pruning* in a distributed environment. In contrast to sideways information passing, the graph summary is directly merged into the distributed relational-based indexes and thus allows us to perform this kind of join-ahead pruning in combination *with an asynchronous execution* of the join operators.
- TriAD employs two stages of query optimization (and execution) over both the summary graph and the data graph (a labeled directed multi-graph). Our distribution-aware query optimizer employs detailed summary- and data-graph statistics to determine the *best exploration-order* for the summary graph and the

best join-order for the data graph, respectively. Both optimization steps are implemented via an efficient DP algorithm.

- Each individual join operator runs against a *distributed, horizontally partitioned index*, such that even for a single join or path-like queries TriAD benefits from the distributed evaluation of these joins. In addition, for a more "bushy" query plan, consisting of multiple root-to-leaf paths (called "execution paths"), the execution of the joins runs in multiple threads at each compute node, which allows us to evaluate multiple operators in the query plan in parallel and asynchronously along these execution paths.
- We provide an *extensive experimental comparison* of TriAD to no less than nine state-of-the-art RDF, DBMS and Hadoop engines. We achieve the —to our knowledge— so far fastest query response times, in comparison to the prior state-of-the-art systems, for the LUBM, BTC and WSDTS benchmarks reported for a mid-range server and regular Ethernet setup.

4.2 Background & Preliminaries

In this section, we briefly review the key concepts that form the basis for the design of TriAD. We start with defining the data and query model used in the current chapter, and then provide a quick overview of related work in processing BGP queries.

4.2.1 Data & Query Model

We consider a labeled directed multi-graph (see Section 2.2.1.1), where vertices are uniquely labeled and allow more than one labeled, directed edge between a pair of vertices, as the data model for this work. Following the Definition 2.3, we denote the labeled directed multi-graph as $G(V, E, \Sigma_V, \Sigma_E, \Phi)$ and, henceforth, refer to it as just "*graph*".

As described in Section 2.2.1.3, we partition a graph G into k vertex-disjoint partitions $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ and refer to $\mathcal{G}$ as the partitioning of G.

As real-world instances of the graph G, we consider RDF graph datasets. An RDF dataset consists of a set of triples of the form $\langle$*subject*, *predicate*, *object*$\rangle$ (or $\langle s, p, o\rangle$, for short), where *subject* denotes a globally unique resource, *object* may denote either a unique resource or a literal (i.e., a string or a number), and *predicate* denotes a relationship between the subject and object.

Example 4.1. *Figure 4.1 shows an example RDF graph G with partitioning $\mathcal{G} = \{G_1, G_2, G_3, G_4\}$ and the corresponding triplet form (NT format) of G is shown in Table 4.1.*

We consider basic graph patterns (BGP), defined in Section 2.2.2.2, as the query model in this work. Re-describing the BGP query model, a BGP query $Q(V_Q, E_Q, \Sigma_V, \Sigma_E, \mathcal{V}, \Phi_Q)$ is a graph where edge set comprises of a set of triple patterns,

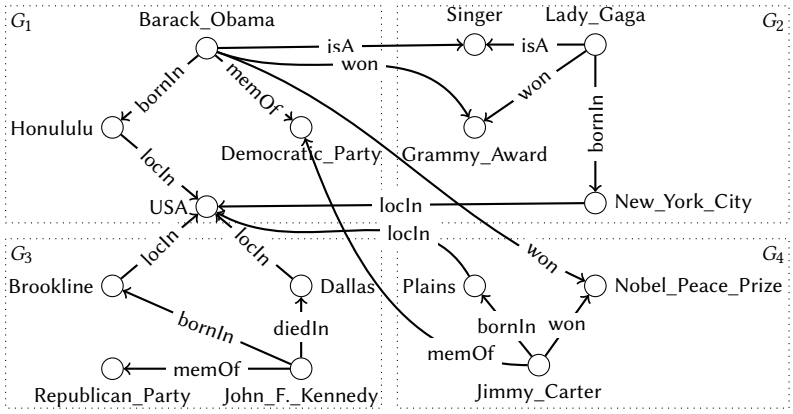


Figure 4.1: RDF graph G with a locality-based partitioning $\mathcal{G} = \{G_1, G_2, G_3, G_4\}$

Subject	*Predicate*	*Object*
Barack_Obama	bornIn	Honolulu .
Barack_Obama	won	Peace_Nobel_Prize .
Barack_Obama	won	Grammy_Award .
Honolulu	locIn	USA .
⋮	⋮	⋮

Table 4.1: Example RDF in NT format.

each of the form $\langle u, e, v \rangle$. Vertices $u, v \in V_Q$, query vertices, can be either a constant or variable, i.e., $u, v \in \{\Sigma_V \cup \mathcal{V}\}$. While, e denotes an edge label satisfying $e \in \{\Sigma_E \cup \mathcal{V}\}$.

Example 4.2. *An example query "Find all the people who are born in a city that is located in the "USA" and won a prize" can be expressed as a BGP query via SPARQL 1.0* [4] *query language and Cypher* [5] *as follows.*

SPARQL 1.0:

```
SELECT ?person ?city ?prize
WHERE { ?person bornIn ?city.
        ?city locIn USA.
        ?person won ?prize.}
```

Cypher:

```
MATCH (person) - [:bornIn] -> (city)
MATCH (city) - [:locIn] -> (USA)
MATCH (person) - [:won] -> (prize)
RETURN person, city, prize
```

Processing the above BGP query Q against an RDF graph G thus resolves to finding all subgraph isomorphisms between Q and G. The result—in analogy to SQL— is a set of rows, each containing a distinct set of bindings of query variables in $\mathcal{V}$ to constants in $\Sigma_V \cup \Sigma_E$.

[4] www.w3.org/TR/rdf-sparql-query/
[5] neo4j.com/developer/cypher-query-language/

Example 4.3. *For example, the result of the above SPARQL/Cypher query over our RDF graph (shown in Figure 4.1) is the following tuples.*

person	*city*	*prize*
Barack_Obama	Honolulu	Peace_Nobel_Prize
Barack_Obama	Honolulu	Grammy_Award
Lady_Gaga	New_York_City	Grammy_Award
Jimmy_Carter	Plains	Peace_Nobel_Prize

4.2.2 Related Work

In the next, we discuss a selection of RDF engines, which extensively support BGP queries via SPARQL, and we believe are most related to our approach. We also briefly discuss their differences to our architecture and refer the reader to (Cudré-Mauroux et al., 2013; Sakr and Al-Naymat, 2010; Sidirourgos et al., 2008) for a comprehensive overview of recent approaches.

4.2.2.1 Relational Approaches

The majority of the existing RDF stores, both centralized and distributed, follow a relational approach towards storing and indexing RDF graphs, and processing BGP queries via SPARQL. In a relational based approach, edges of an input RDF graph are stored in a relation table R. A BGP query, such as the one expressed in SPARQL, is then rewritten into an SQL query and processed over the relation R. A typical BGP query with m patterns thus requires m self-joins over the relation R. Systems like Apache Jena (Jena, 2007), Sesame (Broekstra et al., 2002), 3store (Harris and Gibbins, 2003) are some of the early RDF engines that rely on relational backed engines to efficiently process BGP queries expressed in SPARQL.

For instance, consider the graph shown Figure 4.1. In a relational approach, a relation $R(S, P, O)$ is built by storing each edge of the graph as a tuple in R as shown in Figure 4.2(a). The SPARQL query Q shown in Example 4.2 is then rewritten into an equivalent SQL query with three self-joins over R as shown in Figure 4.2(a)

One of the major drawbacks of the above technique is the expensive self-joins involved in query processing. On graphs with millions and billions of edges, which are common today, this naïve approach would hinder real-time performance which is essential in many applications. Recent approaches, such as SW-store by Abadi et al. (Abadi et al., 2009), vertically partition RDF triples into multiple property tables to mitigate this problem. On the other hand, Hexastore (Weiss et al., 2008) and RDF-3X (Neumann and Weikum, 2010a,b), still relying on a giant table, employ index-based solutions by storing triples directly in B^+-trees over multiple, redundant index permutations. Including all permutations and projections of the SPO attributes, this may result in up to 15 such B^+-trees (Neumann and Weikum, 2010a). Coupled with sophisticated statistics and query-optimization techniques, these centralized, index-based approaches still are very competitive as recently shown in (Tsialiamanis et al., 2012).

R

S	*P*	*O*
Barack_Obama	bornIn	Honolulu
Barack_Obama	won	Peace_Nobel_Prize
Barack_Obama	won	Grammy_Award
Honolulu	locIn	USA
⋮	⋮	⋮

(a)

```
SELECT R1.S, R1.O, R3.O
FROM R AS R1, R AS R2, R AS R3
WHERE R1.O = R2.O AND R1.S = R3.S
      AND R1.P ='bornIn' AND R2.P ='locIn'
      AND R2.O ='USA' AND R3.P ='won'
```

(b)

Figure 4.2: An example of RDF graph (a) represented as a relation R and SPARQL query (b) written as an SQL query

Join-Order Optimization. Determining the optimal join-order for a query plan is arguably the main factor that impacts query processing performance. RDF-3X (Neumann and Weikum, 2010a) thus performs an exhaustive plan enumeration in combination with a bottom-up DP algorithm and aggressive pruning in order to identify the best join order. In TriAD, we adopt the DP algorithm as it is described in (Neumann and Weikum, 2010a), and we adapt it to finding both the best exploration-order for the summary graph and the best join-order for the subsequent processing against the distributed indexes. Moreover, by including detailed distribution information and the ability to run multiple joins in parallel into the underlying cost model of the optimizer, we obtain query plans that are specifically tuned towards parallel execution than with a pure selectivity-based cost model.

Join-Ahead Pruning. Join-ahead pruning is a second main factor that influences the performance of a relational query processor. In join-ahead pruning, triples that might not qualify for a join, called as "*dangling triples*", are pruned even before the actual join operator is invoked. This pruning of dangling triples ahead of the join operators may thus save a substantial amount of computation time for the actual joins. Instead of the sideways information passing (SIP) strategy used in RDF-3X (Neumann and Weikum, 2010a,b), which is a runtime form of join-ahead pruning, TriAD employs a similar kind of pruning via graph summarization (Milo and Suciu, 1999; Picalausa et al., 2012; Zou et al., 2011). Graph summarization, discussed later in this section, serves as a preprocessing step to the actual query executions and thus has the crucial advantage that it can be adapted to an asynchronous execution of the join operators.

MapReduce. Based on the MapReduce paradigm, distributed engines like H-

RDF-3X (Huang et al., 2011) and SHARD (Rohloff and Schantz, 2011) horizontally partition an RDF collection over a number of compute nodes and employ Hadoop as a communication layer for queries that span multiple nodes. H-RDF-3X (Huang et al., 2011) partitions an RDF graph into as many partitions as there are compute nodes via METIS (Karypis and Kumar, 1998). Then, a one- or two-hop replication is applied to index each of the local graphs via RDF-3X (Neumann and Weikum, 2010a). Query processing in both systems is performed using iterative Reduce-side joins, where the Map phase performs selections and the Reduce phase performs the actual joins (Lin and Dyer, 2010). Although such a setting works well for queries that scan large portions of the RDF data graph, for less data-intensive queries the overhead of iteratively running MapReduce jobs and scanning all—or large amounts—of the RDF tuples during the Map phase is significant. Even recent approaches like EAGRE (Zhang et al., 2013) that focus on minimizing I/O costs by carefully scheduling Map tasks and utilizing extensive data replication cannot completely avoid Hadoop-based joins in the case of longer-diameter queries or unexpected workloads. Our experimental evaluation clearly shows that running joins via Hadoop should be avoided if interactive query response are desired.

4.2.2.2 Native Approaches

Recently, a number of approaches were proposed to store RDF triples in native graph format. These approaches typically employ adjacency lists as a basic building block for storing and processing RDF data. Moreover, by using sophisticated indexes, like gStore (Zou et al., 2011), BitMat (Atre et al., 2010) and TripleBit (Yuan et al., 2013), or by using graph exploration, like in Trinity.RDF (Zeng et al., 2013), these approaches prune many triples before invoking relational joins to finally generate the row-oriented results of a SPARQL query. We believe that with Trinity.RDF (Zeng et al., 2013), we provide a detailed experimental comparison to such graph approaches for RDF, which thus also represents a wider family of more generic graph engines such as Pregel (Malewicz et al., 2010) or Neo4j (Neo4j, 2012). Other kinds of graph queries, such as reachability, shortest-paths or random walks, are partly already included in the SPARQL 1.1 standard and required for RDF/S-style inferences. Such queries are targeted by various graph engines, such as FERRARI (Seufert et al., 2013) or GraphX (Xin et al., 2013), but we consider these to be beyond the scope of this chapter. Also beyond our current scope are workload awareness (Shang and Yu, 2013) and incremental updates (Neumann and Weikum, 2010b).

Graph Exploration vs. Joins. To avoid the overhead of Hadoop-based joins, Trinity.RDF (Zeng et al., 2013) is based on a custom protocol based on the Message Passing Interface (MPI) (MPI et al., 2009). In Trinity.RDF, however, intermediate variable bindings are computed among all slave nodes via graph exploration, while the final results need to be enumerated at the single master node using a single-threaded, left-deep join over the intermediate bindings. As an example, consider a SPARQL query with 3 variables $\langle ?x, ?y, ?z \rangle$, which each become bound

to 10 distinct constants during graph exploration. Assuming that each combination of the bindings generates a valid SPARQL result, the 30 bindings lead to 1,000 rows that need to be generated for the join. Thus, the ability to evaluate joins in parallel remains a crucial factor for scaling-out an RDF engine.

4.2.3 Graph Summarization

Graph summarization is an effective approach to prune dangling triples prior to the actual query processing. In graph summarization, a large data graph is first summarized into a smaller graph that retains the principal characteristics of the original RDF data graph in a compact way. The main intuition behind graph summarization is that processing a query over the summary graph allows us to remove large parts of the data graph that contain no relevant triples with respect to the query. Running a complex query against both the summary graph and subsequently against the pruned data graph may thus be faster than running the query against the original data graph. We formally define an summary graph as follows.

Definition 4.1. *A **summary graph** $G_S(V_S, E_S, \Sigma_S, \Phi_S)$ for a given RDF graph $G(V, E, \Sigma_V, \Sigma_E, \Phi)$ again is an labeled directed multi-graph (with only edge labeling) where each node $\mathrm{p} \in V_S$, with $\mathrm{p} \subseteq V$, called **supernode**, and each edge $\langle \mathrm{p}_1, e, \mathrm{p}_2 \rangle \in E_S \in V_S \times V_S \times \Sigma_S$, called **superedge**, connects two supernodes in $\mathrm{p}_1, \mathrm{p}_2 \in V_S$, Function Φ_S maps e to a label in Σ_S.*

4.2.3.1 Generating Graph Summaries

Here, we discuss two graph summarization techniques that are popular among RDF stores to summarize RDF graphs.

- *Bisimulation-based summaries.* Bisimulation (Park, 1981; Vaglini, 1991) is an important concept in concurrency theory and set theory that deals with the similarity between transition systems and majorly studied in the context of labeled transition systems (Sangiorgi, 2009), which are essentially labeled directed graphs. Bisimulation impose a binary relation defined on the vertices of the graph and can be defined as follows.

  Definition 4.2. *Given a graph $G(V, E, \Sigma_V, \Sigma_E, \Phi)$, a binary relation $u\mathcal{R}v$, where $u, v \in V$, is a bisimulation of G, if it holds that, i) for $\langle u, p, w \rangle \in E \Rightarrow \exists w' \in V$ such that $\langle v, p, w' \rangle \in E$ and $w\mathcal{R}w'$, and conversely for $\langle v, p, w \rangle \in E \Rightarrow \exists w' \in V$ such that $\langle u, p, w' \rangle \in E$ and $w\mathcal{R}w'$. Moreover, vertices u, v are called bisimilar.*

  A few centralized RDF stores have so far been proposed to perform join-ahead pruning via bisimulation-based graph summarization (Picalausa et al., 2012; Zou et al., 2011). These extend the idea of bisimulation (Milo and Suciu, 1999), which was originally employed for XML tree summarization, to RDF graphs. Bisimulation-based summaries (Picalausa et al., 2012) are particularly effective for join-ahead pruning if only the predicates of the query triple patterns are

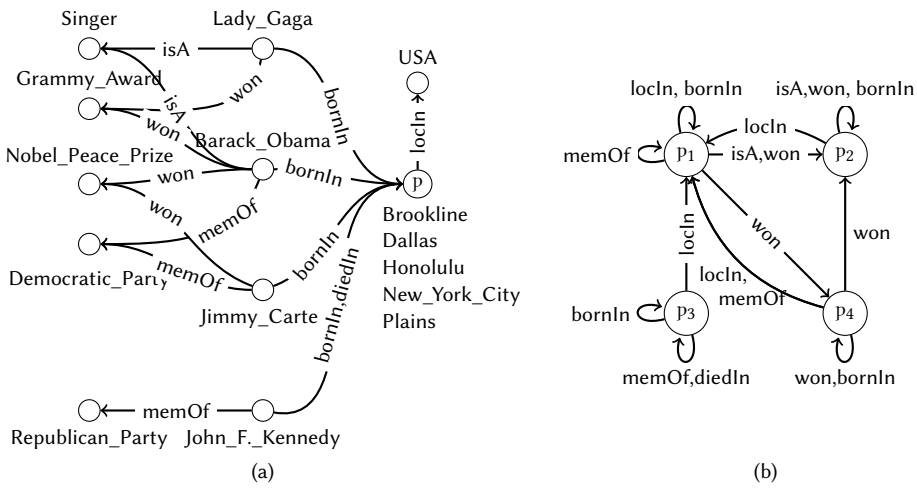


Figure 4.3: An example of (a) bisimulation-based summary and (b) locality-based summarization of RDF graph G shown in Figure 4.1

labeled with constants, such that multiple, possibly disconnected components of the data graph are merged into compact synopses for indexing.

Figure 4.3(a) shows a summary graph of the example RDF graph (shown in Figure 4.1) based on the bisimulation. It can be observed that the vertices Brookline, Dallas, Honolulu, New_York_City, Plains are pair-wisely bisimilar and are, thus, grouped to form single supernode vertex (p) in the summary graph.

Example 4.4. *Running the BGP query on the bisimulation summary graph shown in Figure 4.3(a) binds the variables* ?person, ?city, ?prize *as shown below.*

person	*city*	*prize*
Barack_Obama	p	Peace_Nobel_Prize
Lady_Gaga		Grammy_Award
Jimmy_Carter		

- *Locality-based summaries.* On the other hand, *locality-based summaries* (Zou et al., 2011) are similar to graph clustering, in which nodes of the data graph are partitioned such that the nodes within each partition share more neighbors than the nodes that are spread across the partitions. Since SPARQL typically involves finding connected components of the data graph, locality-based approaches are particularly effective in pruning if one or more of the subjects or objects in the query graph are labeled with constants. Such queries are very common in SPARQL. An example of the locality-based summary graph for the example RDF graph (shown in Figure 4.1) using the partitioning $\mathcal{G} = \{G_1, G_2, G_3, G_4\}$ is shown in Figure 4.3(b), where p_i denotes supernode for the partition G_i.

Example 4.5. *Running our BGP query against the summary graph G_S of Figure 4.3 binds partitions $\mathrm{p}_1, \mathrm{p}_2, \mathrm{p}_4$ to ?person, $\mathrm{p}_1, \mathrm{p}_2, \mathrm{p}_4$ to ?city and $\mathrm{p}_2, \mathrm{p}_4$ to ?prize. Thus, all RDF triples in G, which are associated with p_3, can safely be pruned when*

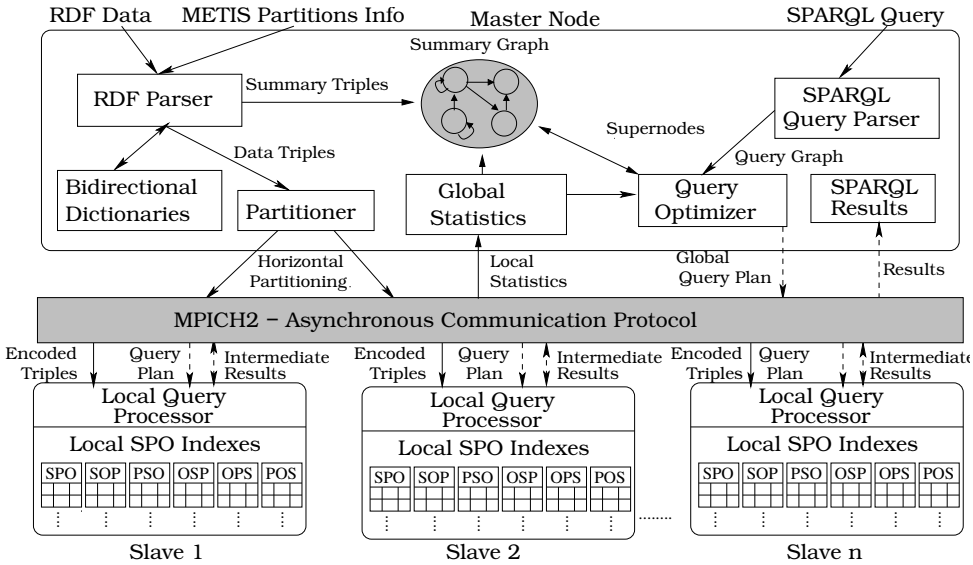


Figure 4.4: TriAD system architecture

processing the query against the data graph without introducing false negatives to the result. By processing the query against G, we replace these supernode bindings of the query variables with their actual RDF constants and thus remove also false positives from the results. Often, this form of join-ahead pruning allows us to detect empty join results without even touching the data graph at all.

4.3 System Architecture

In this section, we provide an overview of the TriAD system architecture designed to process BGP queries in a distributed manner. We consider the input labeled directed multi-graph to be represented in RDF and the BGP queries to be expressed in SPARQL query language. Figure 4.4 depicts the TriAD architecture. TriAD resembles a typical master-slave, shared-nothing model, in which each compute node manages its own main memory area and stores disjoint partitions of the RDF index structures. One designated compute node, the *master node*, stores all metadata about the indexed RDF facts and serves as the initial point of contact for all indexing and query processing tasks. The remaining *slave nodes* hold the local index structures and exchange intermediate query results via a direct, asynchronous communication protocol among each other. All communication is based on the Message Passing Interface (MPI) using the MPICH2[6] API.

Master Node

RDF Parser & Partitioner. This component takes care of parsing RDF files (provided in TTL/N3 format) and partitioning the complete set of incoming RDF

[6] http://www.mpich.org/

triples into the summary graph and the local SPO index structures (Section 4.4).

SPARQL Parser. The SPARQL parser is responsible for preprocessing incoming queries. Queries are turned into a graph representation, before the query optimizer compiles the query into a global join plan which is then sent to all slaves (Section 4.5).

Summary Graph. The initial processing of a SPARQL query pattern against the summary graph facilitates join-ahead pruning (using a locality-based summarization strategy) at the slaves by removing graph partitions that contain no matching triples for the graph pattern denoted by the query (Section 4.4.1).

Bidirectional Dictionaries. The RDF parsing step involves building bidirectional mappings for the incoming RDF triples in order to quickly convert strings to integer ids and vice versa. To accommodate our graph partitioning scheme for the summary graph, the forward dictionary maintains the combination of partition identifier (a node in the summary graph) and component id (Section 4.4.2).

Global Statistics. When indexing finishes, the master receives the local index statistics from the slaves and merges these into its own global statistics to be used for query optimization (Section 4.4.5).

Query Optimizer. In a second processing step, the query optimizer (Section 4.5.3) builds the global query plan based on the global statistics, the locality of the SPO indexes, and cardinality re-estimations after processing the query against the summary graph.

Slave Nodes

Local SPO Indexes. At each slave, a local indexer receives the id-formatted triples and builds its local index structures for each of the six primary SPO permutations (Sections 4.4.3 & 4.4.4).

Local Query Processors. Each slave receives a copy of the global query plan from the master, whereupon the local query processors initialize their own instances of the physical query operators in the plan. The slaves concurrently start executing the same plan but scan different partitions of their local SPO indexes. Along with the global plan, the master also communicates the join-head pruning information from the summary graph to the slaves (Section 4.5.4).

4.4 Index Organization

In this section, we provide a detailed description of the data partitioning and indexing strategies employed by TriAD.

4.4.1 Global Summary Graph

In order to avoid processing unnecessarily large SPO permutation lists at query time, we pursue a join-ahead pruning technique at the master node. Specifi-

cally, we employ a summary graph, denoted as $G_S(V_S, E_S, \Sigma_S, \phi_S)$, for this purpose, which is stored at the master and serves as a concise summary of the actual RDF graph $G(V, E, \Sigma_V, \Sigma_E, \phi)$ (see Definitions 2.3 & 4.1).

Partitioning. Incoming triples, as they are produced by the RDF parser, are of the form $\langle s, p, o\rangle$, where $s, o \in V$ and $p \in \Sigma_E$ is a distinct label for a given pair of vertices s, o. In order to create the summary graph, we first consider this set of RDF facts as one large graph G (using an intermediate dictionary for mapping node and edge labels to integer ids) and apply a non-overlapping graph-partitioning algorithm like METIS (Karypis and Kumar, 1998) to it. METIS pursues a min-k-cut graph partitioning strategy via a form of iterative refinement. At each iteration, the size of the graph is reduced by collapsing vertices and edges, which makes it easier to partition the resulting smaller graph and allows METIS to scale to large graphs with many millions of edges. In the resulting partitioning scheme, each distinct subject s or object o that occurs in an RDF triple is assigned to exactly one graph partition (i.e., supernode) $\mathrm{p} \in V_S$.

The resulting summary graph is treated as a new set of triples of the form $\langle \mathrm{p}_1, p, \mathrm{p}_2\rangle$, where $\mathrm{p}_1, \mathrm{p}_2 \in V_S$ are supernodes. For each original $\langle s, p, o\rangle$ triple that lies in the cut between two supernodes p_1,p_2, a new superedge $\langle \mathrm{p}_1, p, \mathrm{p}_2\rangle \in E_S$ is added to G_S. Within each $\mathrm{p}_i \in V_S$, the original edges of the RDF data graph form self-loop edges of p_i. Moreover, among each such pair of supernodes p_i, p_j $\in V_S$, the summary graph only stores edges with distinct labels p. Altogether, this reduces the size of the summary graph in comparison to the data graph drastically (see Figure 4.3(b)).

Indexing the Summary Graph. After partitioning the data graph, summary triples of the form $\langle \mathrm{p}_1, p, \mathrm{p}_2\rangle$ are indexed at the master node. To support an efficient exploratory search over the summary graph, we index edges in G_S in an adjacency-list-like format. These are stored as two large in-memory vectors holding the PSO and POS permutations of the summary triples for both forward (outgoing links) and backward (incoming links) lookups. Each of the two vectors is sorted in lexicographical order and processed via a combination of binary search and direct pointer accesses.

Optimal Number of Partitions. Determining the number of partitions that minimizes the combined query cost over both the summary and the (pruned) data graph purely empirically may be a very tricky and costly procedure by itself. In order to obtain an estimation of the best summary graph size, we formulate the following cost model as an optimization problem that takes both the centralized query execution at the summary graph and the subsequent distributed execution at the pruned data graph into account.

Let $|V|$ and $|E|$ be the number of nodes and edges in the data graph, respectively, and let d be the average degree of a node in the data graph, i.e.,

$$d := \frac{|E|}{|V|}$$

Further, let c_D denote the cost of executing a query against the graph in a centralized setting. Ideally, the cost $c_{D,n}$ for processing a query in a distributed setting linearly scales with the number of slaves n, i.e.,

$$c_{D,n} = \frac{c_D}{n}$$

Similarly, let $|V_S|$ be the targeted number of nodes in the summary graph. Then it is reasonable to assume that the cost c_S of processing the query against the summary graph is proportional to the summary graph size, i.e.,

$$c_S := \frac{|E_S|}{|E|} \cdot c_D = \frac{d\,|V_S|}{|E|} \cdot c_D$$

Finally, let $|V_P|$ and $|E_P|$ be the number of nodes and edges in the data graph pruned by preprocessing the query against the summary graph. Then the cost $c_{P,n}$ of processing the query against the pruned graph in a distributed setting is

$$c_{P,n} := \frac{|E_P|}{|E|} \cdot c_{D,n}$$

Assuming further that the size of the pruned data graph—at least for selective queries—is inversely proportional to the size of the summary graph, we can rewrite the latter cost as

$$c_{P,n} = \frac{\lambda}{|V_S|} \cdot c_{D,n}$$

Putting all these costs together, we obtain the total cost $c_{Q,n}$ of processing a query against the summary and subsequently against the data graph as follows.

$$\begin{aligned} c_{Q,n} &:= c_S + c_{P,n} \\ &= \frac{d\,|V_S|}{|E|} \cdot c_D + \frac{\lambda}{|V_S|} \cdot \frac{c_D}{n} \end{aligned} \tag{4.1}$$

This yields a cost function that is convex in $|V_S|$. Minimizing $c_{Q,n}$ thus gives an optimal number of nodes when

$$|V_S| := \sqrt{\frac{\lambda |E|}{d\,n}} \tag{4.2}$$

We remark that this result coincides with information-theoretic results for determining the optimal number of clusters in a data set (Sugar and Gareth, 2003).

Although this makes the number of summary graph partitions (e.g., for METIS) easy to compute, in practice, the best choice of partitions certainly depends on a multitude of parameters, including the particular characteristics of the given data set, the query workload, the hardware configuration, as well as the network bandwidth and latency. We project all these latent parameters into a single parameter λ in our cost model, which we need to measure (only once) empirically for a given hardware, query workload, and dataset setting.

Example 4.6. *We empirically verified how well a measured value of λ generalizes to different scales of a given data set and query workload as follows. Based on the LUBM-160 benchmark with queries Q1–Q7 (see Section 5.6), we first stepwisely adjusted the number of summary graph partitions to find the value of $|V_S|$ that minimized the geometric mean of the queries' runtimes. LUBM-160 consists of $|E| = 27.9 \times 10^6$ triples with an average node degree of $d = 3.6$, and by varying $|V_S|$, we determined the best number of summary graph partitions to lie at around $|V_S| = 17k$ partitions. Thus, plugging the above values into Equation (4.1) for a cluster of $n = 5$ slaves, we obtain a value of $\lambda = 187$. We next use this value of λ to predict the best number of partitions for the LUBM-10240 setting (using the same queries), which consists of $|E| = 1.7 \times 10^9$ triples. Equation (4.2) predicts $|V_S| = 136k$ partitions, which is very well within the range of the actual best number of partitions, which we again manually determined to lie in between $100k$–$200k$ partitions (see Figure 4.8.A.4).*

4.4.2 Encoding Triples

After determining the summary graph partitions that each subject s and object o in the RDF data graph belongs to, the master node encodes the partitioning information directly into these triples. For this, let $\langle s, p, o\rangle$ denote a triple in the RDF data graph, and let $\langle \mathrm{p}_1, p, \mathrm{p}_2\rangle$ be its corresponding triple in the summary graph. We then obtain the final encoding of triples in the RDF data graph as $\langle \mathrm{p}_1 || s, p, \mathrm{p}_2 || o\rangle$. The integer ids of s and o are obtained by maintaining one separate dictionary (a hash map) per summary graph partition; the ids of p and p_1, p_2 are available from the intermediate dictionary and the summary graph itself.

Example 4.7. *Following the summary graph shown in Figure 4.3(b), the input RDF triple* ⟨Barack_Obama, bornIn, Honolulu⟩ *is encoded as follows. The subject with label* Barack_Obama *is encoded as* $1||1$*, the predicate as* 1*, and finally the object as* $1||2$*, thus yielding* $\langle 1||1, 1, 1||2\rangle$ *as the final encoding for this triple.*

4.4.3 Horizontal Partitioning of Data Triples

As with any distributed system, we partition the set of encoded RDF data triples across the slaves. Our horizontal partitioning scheme aims to preserve the locality information obtained from the summary graph by hashing entire summary graph partitions into the grid-like distribution scheme shown in Figure 4.5. Since each combined $\mathrm{p}_1||s$ and $\mathrm{p}_2||o$ identifier contains information about both the summary graph partition and the actual subject and object identifiers, we can now "shard" these triples as follows. Let $\langle \mathrm{p}_1||s, p, \mathrm{p}_2||o\rangle$ be an encoded RDF triple, and let n be the number of slaves. Then each RDF triple is sharded twice, once by sending it to slave $(\mathrm{p}_1 \mod n)$ and once by sending it to slave $(\mathrm{p}_2 \mod n)$.

Example 4.8. *Consider the two triples* ⟨Barack_Obama, won, Nobel_Peace_Prize⟩ *and* ⟨Barack_Obama, bornIn, Honolulu⟩ *shown in Figure 4.3(b). Here,* Barack_Obama

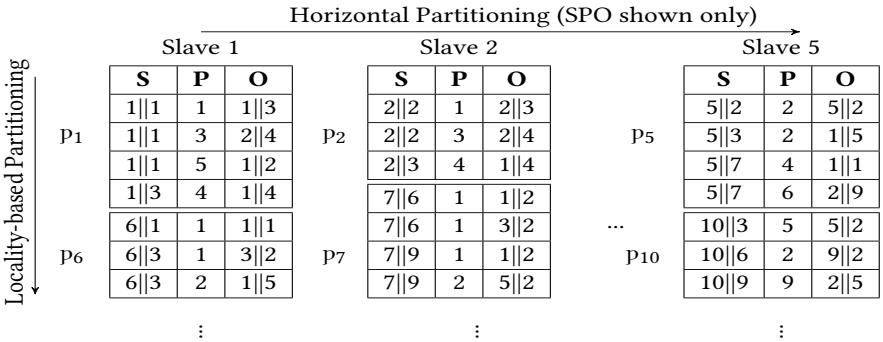


Figure 4.5: Locality-based & horizontal partitioning of triples

and Honolulu *belong to Supernode* 1 *and* Nobel_Peace_Prize *belongs to Supernode* 4. *Considering a cluster of 5 slaves, we distribute the first triple onto Slaves* 1 *and* 4, *whereas the second triple is hashed twice (but sent only once) to Slave* 1.

Locality-Based Sharding and Join-Ahead Pruning. As opposed to the random partitioning schemes used, e.g., in (Rohloff and Schantz, 2011), our hashing scheme aims to preserve the locality information provided by the summary graph. Triples belonging to the same supernode are placed on the same horizontal partition which facilitates join-ahead pruning of partitions that do not contain any triples that are relevant with respect to an entire query. From Example 4.5, assume we know that only partitions p_1, p_2, p_4 are relevant for processing the SPO permutation of the query triple ?*person* bornIn ?*city* because only p_1, p_2, p_4 are bound to the subject variable ?*person* after processing the entire example query against the summary graph shown in Figure 4.3(b). As shown in Figure 4.5 (and from Example 4.8), only the first block each at Slaves 1, 2 and 4 thus is relevant for scanning the SPO permutation for this triple in this query.

4.4.4 Local Permutation Indexes

Upon receiving the sharded triples from the master, the slaves start creating their local permutation indexes in parallel. Each slave creates six large, in-memory vectors of triples, which will serve as our primary index structure for processing queries. Each of the six vectors corresponds to one SPO permutation of the three encoded $\langle p_1||s, p, p_2||o\rangle$ fields. For fast lookups of a given query triple with a set of supernode ids selected from the summary graph, we define methods for random access (via binary search) and sequential access (in the form of iterators) on top of these vector-based SPO lists. Figure 4.5 depicts an example of these SPO indexes at the slaves.

SPO Indexes. At each slave, the six SPO permutations are arranged into two groups: i) the *subject-key* indexes (SPO, SOP, PSO), and ii) the *object-key* indexes (OSP, OPS, POS). All triples hashed onto a slave node via their subject field $p_1||s$ are added to the node's subject-key indexes. Likewise, triples hashed by their object field $p_2||o$ are added to the node's object-key indexes. This way, each encoded RDF

triple is replicated exactly six times across the compute cluster. At each slave, the three subject-key and the three object-key vectors have exactly the same size, respectively.

Sorting Triples. Each of the six triple vectors at a slave is sorted in lexicographic order with respect to its corresponding permutation of the $\langle \mathrm{p}_1 \| s,\ p,\ \mathrm{p}_2 \| o \rangle$ fields. The grid structure shown in Figure 4.5 thus preserves both locality information (i.e., the graph partitions) of the summary graph and guarantees coherence of triples with the same subjects, objects and predicates, respectively.

4.4.5 Local & Global Statistics

In order to create efficient join plans, we compute multiple statistics over both the data and the summary graph. These statistics include i) cardinalities of individual $p_1 \| s$ (subject), p (predicate), and $p_2 \| o$ (object) arguments in case of the data graph and ii) cardinalities of individual p_i (supernode), p (predicate) arguments in case of the summary graph.

In addition, as in (Neumann and Weikum, 2010a), we store cardinalities of iii) $(\mathrm{p}_1 \| s,\ \mathrm{p}_2 \| o)$ (subject, object), iv) $(p,\ \mathrm{p}_2 \| o)$ (predicate, object), v) $(p,\ \mathrm{p}_1 \| s)$ (predicate, subject), and vi) selectivities of (p_1, p_2) (predicate, predicate) pairs as part of the data graph statistics. We follow a similar approach for the summary graph and also store the cardinalities of individual vii) $(p,\ \mathrm{p}_i)$ (predicate, supernode) and viii) selectivities of (p_1, p_2) (predicate, predicate) pairs.

These statistics can only provide us with an exact cost for the first series of index scans, while cardinalities for joins need to be approximated. Estimating the cost of an entire query plan thus requires the recursive estimation of the cardinalities of intermediate relations obtained from joins, which can be formalized as

$$Card(R_{1,2}) := Card(R_1) \cdot Card(R_2) \cdot Sel(R_1, R_2) \tag{4.3}$$

where $Sel(R_1, R_2)$ denotes the selectivity of the pair of predicates (p_1, p_2) associated with the triple patterns R_1 and R_2, respectively. The selectivities for the entire RDF data graph are first aggregated locally at the slaves (in the form of absolute cardinalities) and then merged globally at the master, while the ones for the summary graph are aggregated at the master node, only.

Example 4.9. *For the triple patterns* $R_1 : \langle ?person, \mathsf{bornIn}, ?city \rangle$ *and* $R_2 : \langle city, \mathsf{locIn}, \mathsf{USA} \rangle$, we store the cardinalities $Card(R_1) = 4$ and $Card(R_2) = 5$ at the master node. Similarly, we store the selectivity $Sel(R_1, R_2) = 0.2$ for the pair of predicates (`bornIn`, `locIn`). From Equation (4.3), we thus obtain $Card(R_{1,2}) = 4$ as the estimated number of joined triples.

4.5 Query Optimization & Distributed Processing

In this section, we present a detailed description of the two-staged optimization and execution strategy we follow in processing BGP queries in TriAD.

4.5.1 Two-Staged Query Processing Overview

A BGP query expressed in SPARQL query is parsed and translated into a query graph of the form $Q(V_Q, E_Q, \Sigma_V, \Sigma_E, \mathcal{V}, \Phi_Q)$ (see Definition 2.6) by assigning a unique id to each distinct variable in $\mathcal{V}$, while constants in $\Sigma_V \cup \Sigma_E$ are replaced by ids obtained from the forward dictionary. In the following, we refer to $E_Q = \{R_1, R_2, \ldots, R_n\}$ as the set of query triple patterns that capture a conjunctive BGP query pattern.

Stage 1. The first stage, called "pruning stage", is performed entirely at the master node. We first process the query against the summary graph to find bindings of supernode identifiers to query variables. For this, we employ an exploratory algorithm (similar to the one described in (Atre et al., 2010; Zeng et al., 2013)) for finding these supernode bindings. The reason behind choosing an exploratory-based algorithm over conventional joins is that, here, our objective is to only find supernode bindings for each query variable to facilitate join-ahead pruning at the actual SPO permutation indexes. For an efficient graph exploration, we determine the best exploration order, the *exploratory plan*, using a first DP-based optimizer over the summary graph statistics. The supernode bindings obtained from the pruning stage are relayed to the physical operators at the second stage.

Stage 2. In the second stage, we process the query against the data graph which is distributed across all slaves. Here, we follow a relational style of processing, aiming to generate the final join results of the SPARQL query. We determine the best join order by using a second DP-based optimizer (see, e.g., (Neumann and Weikum, 2010a)) in combination with a distribution-aware cost model as objective function. The supernode bindings obtained at the pruning stage are also used to (re-)estimate the cardinalities of input relations and are fed into the cost model for optimization. This *global query plan* generated at the master is then communicated to all slaves. Along with the global plan, the supernode bindings from Stage 1 are passed on to the slaves for pruning dangling triples (i.e., entire summary graph partitions) from the SPO permutation indexes. At each slave, the local query processor executes the plan by asynchronously sending and/or receiving intermediate join results to/from the other nodes. When query processing terminates, each slave holds its own partial query results which are then finally merged at the master.

4.5.2 Generating Supernode Bindings

The first stage of processing generates supernode bindings via a graph exploration approach. However, unlike the simpler 1-hop graph exploration described

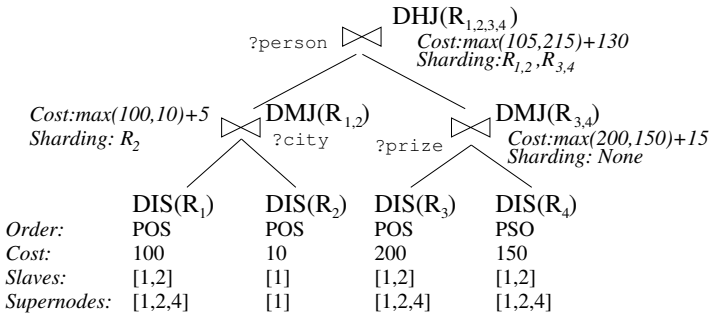


Figure 4.6: Global query plan for the query of Example 4.10

in (Zeng et al., 2013), we perform a full graph exploration with back-propagation. That is, we add a supernode binding to a join variable only if this binding satisfies the entire query also with respect to the remaining join variables.

Example 4.10. *Consider a SPARQL query consisting of the following four triple patterns R_1 to R_4.*

R_1 : ?*person* bornIn ?*city*.
R_2 : ?*city* locIn USA.
R_3 : ?*person* won ?*prize*.
R_4 : ?*prize* hasName ?*name*.

For the fixed exploration order $\langle R_1, R_2, R_3, R_4 \rangle$, we first find all possible bindings for variables ?person and ?city for the query pattern R_1. Next, for the second query pattern R_2, we prune those bindings for variable ?city that are not located in "USA". We propagate this information back to ?person and thus prune supernode bindings for ?person. Finally, with query patterns R_3, R_4, we filter out the bindings for variables ?person, ?city and accordingly add new bindings to ?prize and ?name.

Exploratory Plan Optimization

A random exploration order of query patterns might make the summary graph processing inefficient and sometimes even slower than processing the data graph. To avoid this, we estimate the best exploration order by leveraging the summary graph statistics. To do so, we employ a first bottom-up DP algorithm to determine the order of triple patterns that yields the overall least cost estimate. At each DP step, we calculate the cost of the partial plan considered so far and prune if the current branch cannot contribute to the plan with the least cost anymore.

Based on Equation (4.3), the cost of an entire exploration plan that is represented by a fixed order of triple patterns $R_1, \ldots, R_n$ can thus be estimated as follows.

$$Cost(\langle R_1, \ldots, R_n \rangle) \propto Card(R_1) + \sum_{i=2}^{n} \left(Card(R_i) \prod_{j=1}^{i} Sel(R_i, R_j) \right) \tag{4.4}$$

Here, $Card(R_i)$ denotes the precomputed cardinality of query pattern R_i, and $Sel(R_i, R_j)$ represents the join selectivity of pairs of predicates (p_i, p_j) associated with triple patterns R_i, R_j, respectively (Section 4.4.5). This selectivity is set to 1 if R_i and R_j do not share any join variable. We remark that this estimation again assumes independence among join patterns.

4.5.3 Querying the Data Graph

With the supernode bindings at hand, Stage 2 of the query evaluation is performed over the indexed and sharded RDF data graph. Since there exist six SPO permutations of the entire RDF data graph, which are distributed across n slaves, each individual query pattern R_i could potentially be scanned in six different ways, and each such scan can be done in parallel across the slaves.

Physical Operators. Inspired by the reduced set of query operators in RDF-3X (Neumann and Weikum, 2010a,b), we employ only three distributed operators to construct a query plan in TriAD:

- **Distributed Index Scan (DIS)**: Invokes a parallel scan over a permutation list that is sharded across n slaves.
- **Distributed Merge Join (DMJ)**: Invokes a distributed merge-join across n slaves when both input relations are sorted according to the join key(s) in the query plan.
- **Distributed Hash Join (DHJ)**: Invokes a distributed hash-join across n slaves otherwise.

Each physical DIS operator is aware of the *locality* of the sharded list it scans, the *permutation order* chosen by the optimizer, and the *pruned summary graph partitions* determined by Stage 1. Moreover, both the DMJ and DHJ operators are aware of the *locality* of their input relations and their *join conditions* (see Figure 4.6).

Query-Time Sharding. Both the DMJ and DHJ operators may require sharding a relation at query time. Due to our index layout, the DMJ operator requires sharding of only at most one input relation R_i obtained from a DIS operator when R_i's triples were previously sharded (Section 4.4.3) on a non-join key. For instance, consider the left-hand DMJ shown in Figure 4.6.

Here, using a DIS over the POS index yields all triples for R_2 whose objects are bound to "USA". Since R_2's object is not a join key for the left-hand DMJ, we need to shard R_2's triples according to the join key ?*city* (the subject of R_2). On the other hand, the right-hand DMJ operator requires no query-time sharding at all when scanning the POS and PSO indexes, respectively, since both R_3's and R_4's triples were sharded on the join key ?*prize*. Likewise, the upper DHJ operator requires sharding both of its intermediate input relations, since $R_{1,2}$ and $R_{3,4}$ are not sorted on their common join key ?*person* and thus are misplaced among the slaves with respect to this key.

Global Query Plan Optimization

The choice of a physical join operator strongly influences the cost function determined by the DP optimizer. To initialize the DP table for each pattern R_i and SPO permutation k, which is distributed across n slaves, we set the DIS cost as follows.

$$Cost(R_i^k) \propto \begin{cases} Card(R_i)/n \\ \quad \text{if permutation } k \text{ matches the binding pattern given by the constants in } R_i \\ |E_D|/n \\ \quad \text{otherwise} \end{cases} \tag{4.5}$$

For example, for the query pattern ⟨`Barack_Obama`, `?p`, `?o`⟩, the cost of scanning the matching triples over the SPO, SOP permutations is expected to be much lower compared to scanning them over the OPS, OSP, PSO and POS permutations. For calculating the actual costs $Cost(R_i^k)$ of an index scan, we multiply the basic cardinalities with a constant cost factor η^{DIS}.

After initializing the DP table with the first series of DIS costs, we continue to build a query plan that aims to reflect the optimal order of both *joining* and *shipping* intermediate results across the slaves. At each DP step, we join two subplans over two non-overlapping subqueries Q^{left} and Q^{right} into a new combined plan Q. The cost of Q is then recursively defined as follows.

$$Cost(Q) := \begin{cases} Cost(R_i^k) \\ \quad \text{if } R_i \text{ denotes a DIS over permutation } k; \quad (4.6.1) \\ Cost(Q^{left}) + Cost(Q^{right}) \\ +Cost(Q^{left} \bowtie^{op} Q^{right}) \\ +Cost(Q^{left} \rightleftharpoons^{op} Q^{right}) \quad \text{otherwise.} \quad (4.6.2) \end{cases}$$

Here, $Cost(Q^{left} \bowtie^{op} Q^{right})$ denotes the cost of joining the two subqueries via operator op, which depends on the cardinalities of both Q^{left}, Q^{right} times a constant cost factor η^{op} for the respective join operator $op \in \{\text{DMJ}, \text{DHJ}\}$. Conversely, $Cost(Q^{left} \rightleftharpoons^{op} Q^{right})$ denotes the cost of shipping intermediate relations for Q^{left}, Q^{right} across the slaves before executing the actual join. This is again computed from the cardinalities of Q^{left}, Q^{right}, which are each multiplied with the width of their intermediate relations and a constant factor $\eta^{\rightleftharpoons}$ for the communication cost.

Cardinality (Re-)Estimation. Equation (4.6.1) captures the scan costs for a basic triple pattern R_i to be proportional to the cardinality that is available from our precomputed global statistics. Preprocessing the query against the summary graph however lets us refine these cardinalities by the amount of summary graph partitions that are actually selected for each R_i after the initial graph exploration step. Thus, let $Card(R_i)$ be the precomputed cardinality of a query pattern R_i over

the RDF data graph, and let $|C_s|$, $|C_o|$ be the cardinalities of its subject s and object o, respectively, obtained from the precomputed summary graph statistics. Let $|C'_s|$, $|C'_o|$ be the number of supernode bindings obtained from Stage 1 of processing the query over the summary graph. We then (re-)estimate $Card'(R_i)$ via a simple linear interpolation as follows.

$$Card'(R_i) := \frac{|C'_s|}{|C_s|} \cdot \frac{|C'_o|}{|C_o|} \cdot Card(R_i) \tag{4.6}$$

These re-estimated cardinalities are plugged into Equation (4.6.1) and used by the optimizer when determining the global query plan.

Accounting for Parallel Operations. To accommodate for the parallel execution of two subplans Q^{left}, Q^{right} (Section 4.5.4), we further refine the cost function of Equation (4.6.2) as follows.

$$\begin{aligned} Cost(Q) \quad := \quad & \max\left(Cost(Q^{left}), Cost(Q^{right})\right) \\ & + Cost(Q^{left} \bowtie^{op} Q^{right}) \\ & + Cost(Q^{left} \rightleftharpoons^{op} Q^{right}) \end{aligned} \tag{4.7}$$

That is, at any DP step, the cost of the current (sub-)plan for Q is proportional to the cost of the concurrent execution of the subplans for Q^{left}, Q^{right}, rather than to the cost of their sequential execution. Another significant advantage of parallel executions—in addition to speeding up computations—is that it also better exploits the network bandwidth by sending more than one intermediate relation at a time via asynchronously exchanged messages.

Example 4.11. *Figure 4.6 shows an example of a global plan returned by the optimizer for a two-node distribution. One can observe that the plan explicitly includes the locality and pruning information that each DIS operator has at the leaves. For instance, the POS list chosen for pattern R_2 entirely resides at Slave 1, whereas the ones for R_1, R_3, R_4 are distributed across both slaves. The plan also shows how the parallel execution of subplans affects the cost estimates for the DMJ and DHJ operators.*

4.5.4 Distributed Query Execution

The global query plan generated at the master is communicated to all slaves along with the supernode bindings. Each slave receives this plan, initializes its own instances of the physical operators (but over different chunks of the sharded SPO lists), and then starts processing the plan concurrently. The protocol that is executed at each of the slave nodes concurrently is shown in Algorithm 7.

```
   Input: Global query plan with supernode bindings Plan;
   local SPO index Idx; number of slaves n;
   Output: Relation with partial query results Relation;
1  method Main(Plan, Idx, n, i) {
2    EP[1..l] ← CreateExecutionPaths(Plan); //plan with l leaf op's
3    for j = 1..l do
4      START_THREAD((EP[j], Idx) → Process);
5    Alive[i] ← SendSlaveStatusToMaster(i);
6    WAIT_ALL(EP[1..l]); //synchronize on execution paths
7  return EP[1].Relation; } //return partial result relation for this slave
8  method Process(EP, Idx) {
9    while Op ← NextOperator(EP) do
10     if Op is DIS then
11       SN[1..p] := GetSupernodeBindings(Op); //for join-ahead pruning
12       EP.Relation ← GetIterator(Op, Idx, SN[1..p]); //binary search
13     else
14       Alive[1..n] ← ReceiveSlaveStatusFromMaster();
15       if Op.Sharding then
16         Part[1..n] ← Shard(EP.Relation); //repartition relation
17         EP.Relation ← Part[i]; //keep partition i locally
18         for j ≠ i && Alive[j] do
19           Ack[j] ← MPI_Isend(Part[j], j, EP.Id);
20         for j ≠ i && Alive[j] do
21           Ack[n + j] ← MPI_Ireceive(Part[j], j, EP.Id);
22           EP.Relation ← Merge(EP.Relation, Part[j]);
23         WAIT_ALL(Ack[1..2n]); //synchronize on incoming messages
24       SibEP ← FindSiblingExecutionPath(Op);
25       R1 ← EP.Relation;
26       R2 ← SibEP.Relation;
27       if SibEP.Id < EP.Id then
28         STOP_THREAD(EP);
29       EP.Relation ← Join(R1, R2,Op); } // Op is DMJ or DHJ
```

Algorithm 7: Local query processor at Slave i

Multi-Threaded, Asynchronous Plan Execution. The key to allow for a parallel, asynchronous execution of the global query plan lies in executing the plan in a multi-threaded fashion at each slave. The Main method of Algorithm 7 invokes a new thread (using the C++ Boost API) for each sequential *execution path* (EP) of operators in the query plan. An EP is a path from a leaf of the operator tree up to its root (the vertical dashed lines in Figure 4.7). At each slave, we start a separate thread for each such EP (Line 4), and we later join (i.e., synchronize) two threads into one at the lowest common join operator that two such EPs share (Line 28).

As shown in the Process method (and in Figure 4.7), the execution of an EP always starts with the DIS operators. Each DIS operator obtains the respective supernode bindings for join-ahead pruning as part of the global query plan (Line 11).

Instead of building an intermediate relation, a DIS operator returns an iterator that directly points to the first qualifying tuple (obtained via binary search and the supernode bindings) in a sorted SPO permutation list (Line 12). These iterators are then passed to the parent DMJ operators to perform the joins directly on the raw indexes. Otherwise, if the operator is DMJ or DHJ and sharding is required, Slave i first shards the intermediate relation that it holds at its current EP (Line 16). Only in this case, Slave i needs to synchronize on incoming messages (Line 23) from the other $n-1$ slaves in order to merge the incoming (resharded) tuples into the current EP's intermediate relation (Line 22). Thus, each EP holds an intermediate relation which is iteratively passed on to a subsequent join operator (Line 29) in the execution path. Although the operators within an EP are executed sequentially, multiple such EPs (and thus operators) run in parallel and asynchronously at each slave, and across all slave nodes.

Asynchronous MPI Communication. Sharding an intermediate relation for a DMJ or DHJ operator at query time is a blocking operation that requires a synchronization step among all slaves. This may be a significant bottleneck, as it blocks the slaves from performing their partial join operations until all the slaves have received their corresponding chunks of tuples. We address this by using the asynchronous `MPI_Isend` and `MPI_Ireceive` methods of the MPICH2 API (Lines 19 & 21). Thus, without waiting for the entire sharding phase to finish among all slaves, a part of a DMJ or DHJ operation can be invoked locally on a slave as soon as this slave has received the $n-1$ messages with the chunks of tuples it is responsible for (denoted by the horizontal dashed lines in Figure 4.7). Conversely, once a slave finishes all its execution paths, it broadcasts its completion to all other slaves via the master.

In summary, if query-time sharding is required prior to a join, then this step is comparable to a "Shuffle&Sort" phase of a Map-side join in MapReduce (Lin and Dyer, 2010). In our case, shuffling is not always required, and sorting is avoided entirely. Due to the layout of our distributed index structures (Sections 4.4.2 & 4.4.3), we can always rely on efficient DMJ operators for the first level of joins. At this first level, we need query-time sharding only if we join the subject of one query pattern on the object of another query pattern (i.e., we have an S-O or O-S join) and at least one of the non-joining subjects or objects is a constant. Conversely, a DHJ operator requires query-time sharding of either one (or both) of its input relations. During plan generation, this is taken into consideration by the optimizer together with the constant cost factors of the operators, such that we favor merge joins over hashing whenever possible.

Example 4.12. *Figure 4.7 illustrates the distributed execution of the query plan depicted in Figure 4.6 (and shown in Example 4.10) for a two-node distributed setup. At the leafs, the DIS operators (e.g., for R_1) obtain the supernode bindings and each create an iterator over the pruned POS index (shaded partitions). Before invoking the left-hand DMJ on ?city, and since R_2 at Slave 2 is empty, we repartition the triples of R_2 at Slave 1 into two partitions, one of which is sent to Slave 2 (denoted by a hor-*

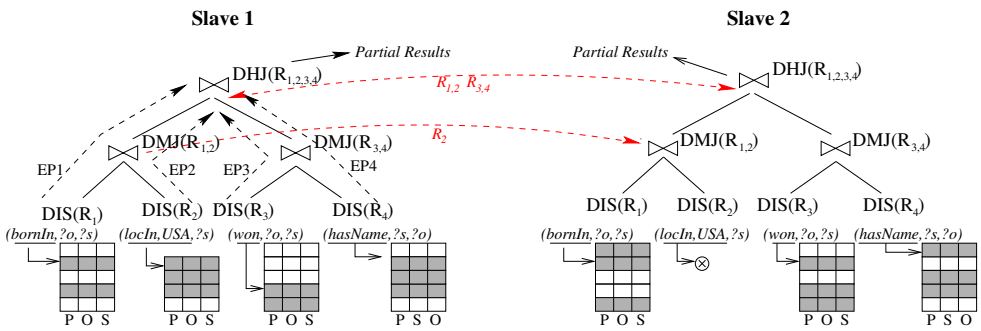


Figure 4.7: Distributed execution of the query shown in Example 4.10 with asynchronous communication (horiz. dashed lines)

izontal dashed line). The two right-hand DMJs on ?prize at Slaves 1 and 2 require no communication, as their input triples are already in-place. Since the two DMJs order tuples on different join keys for ?city and ?prize, only the final DHJ requires sharding and shipping for both $R_{1,2}$ and $R_{3,4}$ for the join on ?person. All of the DIS operators are performed in a fully distributed and multi-threaded fashion. Also the next level of DMJ runs in an asynchronous fashion. Only the final DHJ needs to wait until both of its incoming DMJ operators have finished generating their intermediate results.

4.6 Evaluation

We next present the detailed evaluation of BGP query processing using our TriAD system. For this, we evaluated TriAD in comparison to two centralized RDF engines, RDF-3X (Neumann and Weikum, 2010a) and BitMat (Atre et al., 2010), four distributed RDF engines, SHARD (Rohloff and Schantz, 2011), H-RDF-3X (Huang et al., 2011), 4store (Harris et al., 2009) and the very recent Trinity.RDF (Zeng et al., 2013) engine, one main-memory DBMS, MonetDB (Sidirourgos et al., 2008), as well as to Apache's Hadoop and Spark engines (the latter for comparing to their plain join performance). To study join-ahead pruning, we consider two variants of our system. The first, referred to as TriAD-SG, makes use of the summary graph, while the second, referred to as TriAD, performs a random partitioning of triples.

4.6.1 Datasets & Setup

4.6.1.1 Benchmarks

We used the widely popular LUBM[7] synthetic benchmark, the real-world BTC 2012[8] dataset, and the recent WSDTS[9] SPARQL diversity test suite. For LUBM, we employed the data generator using UBA 1.7 in N3 format. Concerning the

[7] http://swat.cse.lehigh.edu/projects/lubm/
[8] http://challenge.semanticweb.org/
[9] https://cs.uwaterloo.ca/~galuc/wsdts/

queries, we used the benchmark queries published in (Atre et al., 2010) and used by Trinity.RDF (Zeng et al., 2013). For BTC, we defined 8 queries of varying complexity similar to the ones published in (Neumann and Weikum, 2009), replacing only the operators that TriAD currently does not support (i.e., `DISTINCT` and `[]`).

4.6.1.2 TriAD Setup

We implemented TriAD in C++ using GCC-4.4 with -O3 optimization. We used MPICH2-1.4.1 and Boost 1.54.0 as external libraries. For TriAD-SG, we constructed our summary graph by partitioning the RDF data graph using the METIS 5.1 graph partitioner with a default configuration. To achieve a better performance during partitioning, we ignored edges connecting string literals, resulting in both time and space savings. We ran all experiments on a local compute cluster with 12 nodes (one of which was dedicated as the master node) which are connected via a 1GBit LAN connection. Each machine has 48GB of RAM, 16 quad-core CPUs of 2.4GHz, and runs Debian Linux 6.0.6.

4.6.1.3 Competitor Setup

To compare against Hadoop-based engines, we implemented H-RDF-3X (Huang et al., 2011) as our main competitor. For H-RDF-3X, we first partition the graph using METIS and assign each partition to a slave that runs RDF-3X 0.3.7 as its local RDF engine. For a fair comparison, and given that all LUBM queries have a diameter of less than 2, we employ a 1-hop replication. Moreover, since neither Trinity.RDF (Zeng et al., 2013) nor its underlying Trinity graph engine (Shao et al., 2013) are openly available, Tables 4.2 and 4.6 thus depict the running times reported in (Zeng et al., 2013) for the same benchmark setting but over a much stronger hardware configuration. Most notably, our available network bandwidth and main memory lie at 1GBit and 48GB as opposed to 40GBit and 96GB reported in (Zeng et al., 2013), respectively. All other competitors are off-the-shelf installations within our cluster.

4.6.2 Results

4.6.2.1 LUBM-10240 Dataset

Efficiency. In our first series of experiments, we use the LUBM-10240 dataset which consists of about 1.84 billion triples (amounting to a size of 730 GB in raw N3 format). Queries $Q1$–$Q7$ (Atre et al., 2010; Zeng et al., 2013) can be classified as non-selective ($Q2$), selective in both the input relations and output size ($Q4$, $Q5$, $Q6$), and selective in output size ($Q1$, $Q3$, $Q7$). This setup is identical to the one used for evaluating Trinity.RDF (Zeng et al., 2013), thus allowing us draw a careful comparison to their performance results. In Table 4.2, we depict the wall-clock processing times of both TriAD and TriAD-SG in comparison to all competitors. For TriAD-SG, we experimented with different summary graph sizes. Table 4.2

	TriAD	TriAD-SG	Trinity.RDF	SHARD	H-RDF-3X		4store		RDF-3X		BitMat	
		(200K)			*(cold)*	*(warm)*	*(cold)*	*(warm)*	*(cold)*	*(warm)*	*(cold)*	*(warm)*
Q1	7,631	**2,146**	12,648	6.9E5	2.3E6	1.7E5	aborted	aborted	1.9E6	1.8E6	17,339	11,295
Q2	1,663	2,025	6,018	2.1E5	5.3E5	4,095	1.1E5	15,113	6.3E5	17,835	2.4E5	1.8E5
Q3	4,290	**1,647**	8,735	4.7E5	2.2E6	1.3E5	aborted	aborted	1.7E6	1.7E6	8,429	2,679
Q4	2.1	**1.3**	5	3.9E5	166	1	1,903	12	243	3	aborted	aborted
Q5	**0.5**	0.7	4	97,545	85	1	2,429	12	99	1	472	338
Q6	69	**1.4**	9	1.8E5	5.8E5	23,440	3,572	9	913	287	7,796	5,377
Q7	14,895	16,863	31,214	3.9E5	2.3E6	2.1E5	aborted	aborted	6.5E5	46,262	71,157	36,905
Geo. Mean	249	**106**	450	3.0E5	91,378	2,406	-	-	31,345	2,991	-	-

Table 4.2: LUBM-10240 – Query processing times (in *ms*)

depicts our best setting, where 200,000 supernodes and 130,744,241 superedges reside at the master node.

Starting from the non-selective query $Q2$, which contains a single join that returns a large number of results, TriAD outperforms all competitors, thus taking advantage of its distributed join execution. Trinity.RDF is about 3 times slower, since here graph-exploration provides no benefit for non-selective queries, thus retaining many bindings and performing a single, centralized join at the master node. H-RDF-3X, due its use of local RDF stores, can execute the join in parallel and runs faster than Trinity.RDF (warm cache) but due to the unbalanced partition sizes across the local RDF stores, H-RDF-3X remains slower than TriAD. In addition, TriAD, which uses main-memory backed indexes, can perform fast random-access jumps over its indexes. Here, the use of the summary graph (see TriAD-SG) even slightly hurts performance, since $Q2$ does not benefit from join-ahead pruning either.

For the selective queries $Q1$, $Q3$, $Q7$, TriAD manages again to outperform Trinity.RDF. The slower performance of Trinity.RDF apparently is due to its 1-hop distributed graph exploration method without back-propagation (which we conclude from the observation that these queries are only selective in their final output but non-selective for the lowest level of joins). For both $Q1$ and $Q3$, the summary graph with a full graph exploration (including back-propagation) improves the performance of TriAD, since pruning is very effective for these selective queries. Especially for $Q3$, which has an empty result, the summary graph prunes many SPO partitions which leads to performance gains over Trinity.RDF. This impact of full graph exploration is also shown by the centralized BitMat system which is faster than TriAD but slower than TriAD-SG. For query $Q7$, the pruning stage in TriAD-SG is not as effective, thus retaining many SPO partitions and resulting in an inferior performance compared to TriAD due to the overhead of shipping and comparing the supernode identifiers for our index scans. 4store repeatedly crashed on queries $Q1$, $Q3$, $Q7$ (marked as "aborted").

Queries $Q4$, $Q5$, which are processed against many low-cardinality input relations, can be considered as the best cases for effective join-ahead pruning. For these queries, the centralized RDF-3X engine with join-ahead pruning is very efficient. TriAD is slightly faster than RDF-3X (warm cache) and Trinity.RDF by using distributed joins with skip-ahead jumps over the index lists based on the supergraph partitions. In the case of TriAD-SG, where the first-stage processing is negligible, it performs similarly to TriAD.

Trinity.RDF performs better than TriAD for $Q6$, where large intermediate relations hamper the performance of TriAD. The use of the summary graph in TriAD-SG however is almost 50 times faster, thus reducing the size of the intermediate results significantly and outperforming Trinity.RDF. H-RDF-3X performs significantly worse in this case, since it breaks the query into smaller subqueries and fails to capitalize on the SIP benefits of RDF-3X.

Scalability. We studied both the strong and weak scalability of TriAD by in-

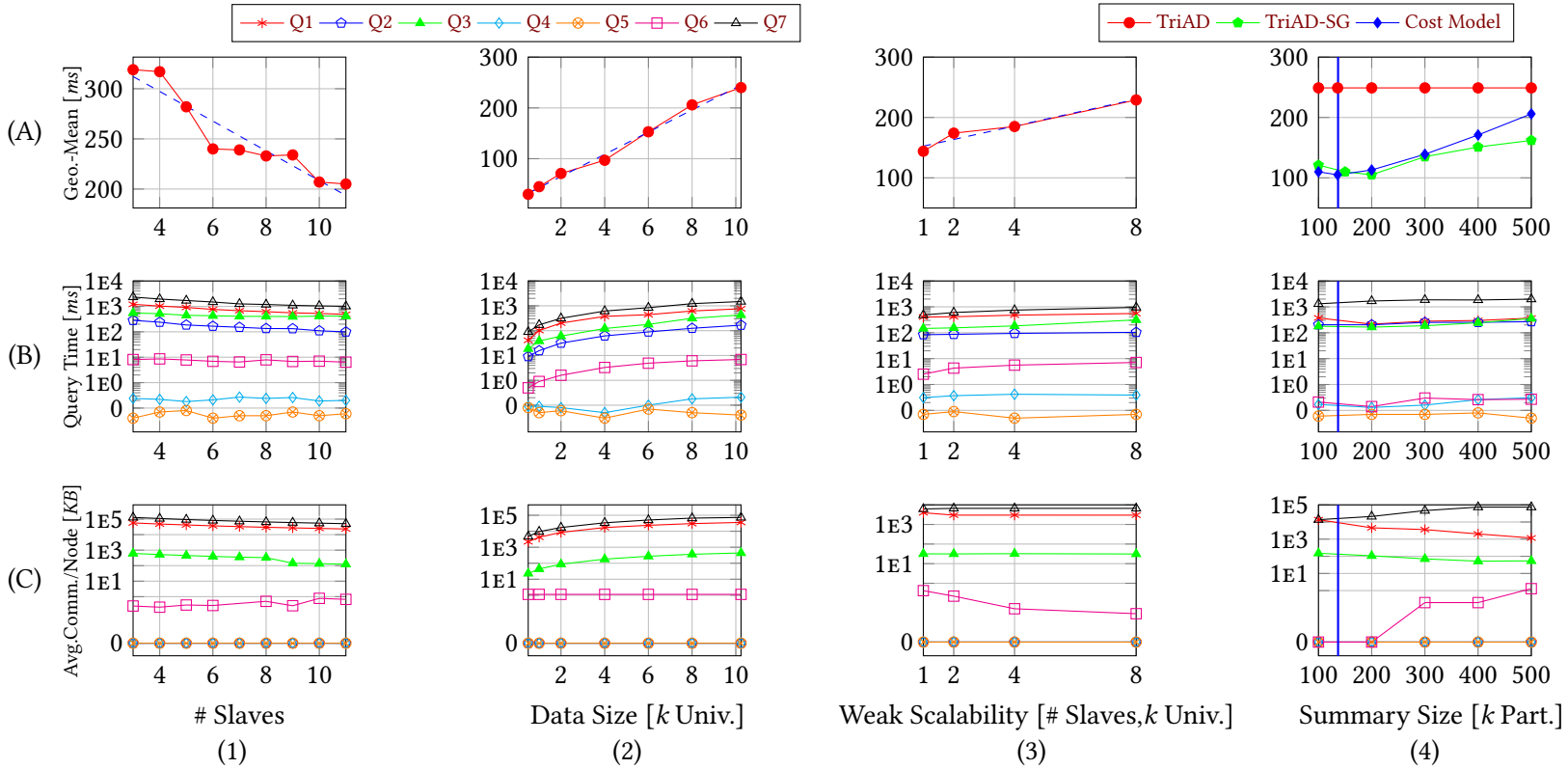


Figure 4.8: TriAD (Cols. 1–3) & TriAD-SG (Col. 4) scalability experiments for various configurations of the LUBM benchmark

creasing the number of available machines and the size of the data set. The results are shown in Figure 4.8. Figures 4.8.C.1 and 4.8.B.1 show the strong scalability in terms of query time when increasing the number of slaves from 2 to 11. Figure 4.8.C.1 shows the average communication costs per slave for this increasing number of slaves. For measuring strong scalability, we use the LUBM-10240 dataset. (We omit the setting with a single slave as our indexes and statistics do not fit into 48GB of RAM.) We observe that our processing time decreases linearly as the number of machines increases and, as expected, we see the average communication cost per slave decreasing while the total communication cost is increasing. We also studied how TriAD performs as we increase the size of the data while keeping the number of machines fixed. Results are shown in Figures 4.8.A.3, 4.8.B3 and 4.8.C.3 and imply a very good scalability for TriAD with respect to the data size. Similarly, Figures 4.8.A.2, 4.8.B.2, 4.8.C.2 depict the case when we increase both the size of the data and the number of available machines. From the geometric means in Figure 4.8.A.2, we can observe that the variance is very low, thus confirming the afore behavior also in terms of weak scalability. Notice that the join multiplicities for $Q1$–$Q7$ are larger than 1, such that the result sizes also grow super-linearly.

Communication Costs. With regard to the communication costs among slaves, our measurements for LUBM-10240 are shown in Table 4.3 (in *KB*). The use of the summary graph generally achieves a better query performance by reducing the size of the intermediate results via join-ahead pruning, thus decreasing both the communication costs and the computational costs of the joins. The maximum gains appear for selective queries $Q1$, $Q3$, and $Q7$.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
TriAD	35,720	0	439	<0.1	<0.1	1.13	73,141
TriAD-SG	4,587	0	107	0	0	0	21,051

Table 4.3: Communication size (in KB) for LUBM-10240

Impact of Summary Graph Size. The query times and the average communication costs for queries $Q1$–$Q7$ for different summary graph sizes are shown in Figures 4.8.A.4, 4.8.B.4 and 4.8.C.4. With increasing summary graph sizes, we generally observe increasing query times, which become dominated by processing the queries against the summary graph. We can also observe a decreasing trend for the communication costs (except for $Q7$) because of more pruning. Figures 4.8.A.4, 4.8.B.4 and 4.8.C.4 show the optimal number of partitions predicted by our cost model (blue vertical line). The TriAD baseline (red horizontal bar) is shown in Figure 4.8.A.4. The cost predicted by our cost model (green curve in Figure 4.8.A.4, see also Section 4.4.1) has been scaled linearly to fit this plot, which however does not affect the shape of the plot nor its minimum.

Impact of Multi-Threading. We evaluated the gain of multi-threading and its effect on plan generation for the LUBM-10240 dataset on a 10-node setup. Figure 4.9 shows the query times of the different variants of TriAD on a logarithmic

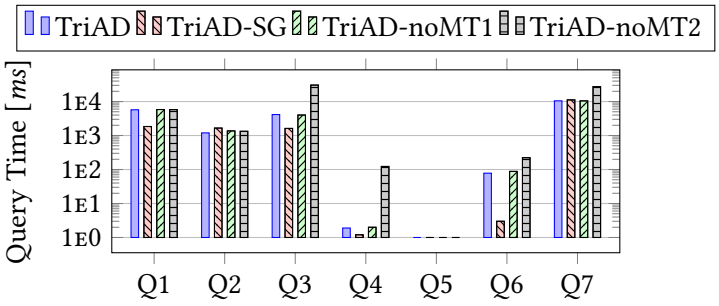


Figure 4.9: Impact of multi-threading in TriAD

Relation Sizes	**Dataset**	**Q5**	**Q2**
R_1 / R_2	LUBM-1000	10B / 3MB	9MB / 180MB
	LUBM-10240	70B / 29MB	103MB / 2GB
Query Time (in *sec*)		**Q5**	**Q2**
TriAD	LUBM-1000	<0.01	0.16
	LUBM-10240	<0.01	1.20
Apache Hadoop	LUBM-1000	21.17	29.69
	LUBM-10240	21.83	73.36
Apache Spark (cold / warm)	LUBM-1000	4.07 / 0.14	26.72 / 15.04
	LUBM-10240	9.36 / 0.48	116.25 / 96.12
MonetDB (cold / warm)	LUBM-1000	0.05 / 0.01	1.52 / 0.05
	LUBM-10240	0.11 / 0.02	26.83 / 0.23

Table 4.4: Single-join performance of various engines

scale. To measure the effectiveness of multi-threading on both plan generation and query execution, we defined two variants: i) TriAD-noMT1 (using our multi-threading-aware cost model for optimization but single-threaded executions), and ii) TriAD-noMT2 (using a single-threaded mode for optimization and execution). For queries $Q3$ and $Q4$, allowing multi-threaded operations achieves an order of magnitude better performance results. A main reason for this large difference—besides a better CPU and network utilization—are improved query plans generated by the optimizer when multi-threading is enabled.

Single-Join Performance. To evaluate the basic performance of joins in Apache Hadoop and Spark versus TriAD, we compared the built-in Map-side join function of Hadoop (over two sorted and key-partitioned input files) with the DMJ operator in TriAD. We ran the comparison over a 10-node cluster setup with two different LUBM scale factors. Table 4.4 shows the relation sizes and the query performance (this time in *seconds*) of Hadoop and Spark (Zaharia et al., 2010) for both a selective ($Q5$) and a non-selective ($Q2$) LUBM query, each consisting of just a single join operation. We can clearly observe that Hadoop-based joins should be avoided. MonetDB, in comparison, yields the by far best join performance when the input relations fit into the main memory of a single machine. It however degrades when optimizing complex SPARQL queries (see Table 4.6). For Apache Spark, we used a naïve implementation of Map-side joins without any caching,

for selective query $Q2$ the warm cache performance is significantly better than Hadoop as the relations fit into memory, while for non-selective query $Q5$, warm cache performance was observed to be lower than Hadoop.

Effectiveness of Dictionary Encoding in TriAD. Dictionary encoding is a crucial element in building efficient RDF systems. Most centralized and distributed RDF systems encode RDF strings into integer IDs before constructing any indexes over the triples. This strategy provides multiple benefits such as small index sizes, flexibility to compress indexes, and the faster query processing times. A naïve dictionary encoding strategy, which many current RDF systems follow, is to assign incremental integer IDs to RDF strings in the order they appear in the RDF file. This may lead to assigning larger IDs to frequently appearing RDF strings. Moreover, the naïve strategy disallows any locality-based encoding of RDF strings. In TriAD-SG, this is mitigated to a larger extent by using the locality-based partitioning and the encoding of RDF triples, but not use a statistical based approach proposed recently in *KOGNAC* (Urbani et al., 2016). *KOGNAC* investigates a novel approach to the dictionary encoding problem by assigning IDs based on the statistical (frequent and infrequent strings) and locality-based grouping of RDF triples. For more details, we refer the reader to our paper (Urbani et al., 2016). Here we briefly present the empirical evaluation on effectiveness of KOGNAC in TriAD. We considered LUBM-1000 and LUBM-8000 dataset comprising of 100 million and 1 billion triples respectively. Table 4.5 shows the performance of TriAD using three different encoding techniques – KOGNAC, Default (appearance order in file), Random (randomly assigned IDs). It can be observed that KOGNAC performs significantly better than Random order, but comparable to Default order.

Queries	**LUBM-1000**			**LUBM-8000**		
	(times in ms)					
	KOGNAC	*Default*	*Random*	*KOGNAC*	*Default*	*Random*
Q1	2,684	2,640	3,090	13,843	12,327	15,205
Q3	106	109	631	471	757	2660
Q4	2	2	3	3	5	3
Q5	1	1	2	1	1	2
Q7	2,558	2,458	3,067	12,107	11,532	16,708
Geo. Mean	68,014	67,628	129,113	188,314	221,904	332,345

Table 4.5: Effectiveness of dictionary encoding in TriAD

4.6.2.2 LUBM-160 Dataset

We also evaluated the performance of TriAD and TriAD-SG over a smaller dataset. For a fair comparison, we used a single slave node setup for this, and the results are shown in Table 4.6. We can observe that TriAD continues to perform well for selective queries $Q4$, $Q5$, $Q6$ and the non-selective query $Q2$. For the remaining selective queries $Q1$, $Q3$, $Q7$, the large intermediate relations hamper performance,

	TriAD	TriAD-SG *(17K)*	Trinity-.RDF	RDF-3X *(cold)*	RDF-3X *(warm)*	MonetDB *(cold)*	MonetDB *(warm)*	BitMat *(cold)*	BitMat *(warm)*
Q1	427	**97**	281	38,802	27,702	10,600	1,500	1,078	1,053
Q2	**117**	140	132	32,936	347	279	174	3,055	3,030
Q3	210	**31**	110	27,692	27,678	10,900	1,700	47	40
Q4	2	**1**	5	76	2	39	25	5,421	5,357
Q5	0.5	**0.2**	4	1	1	80	23	6	6
Q6	19	**1.8**	9	59	7	130	51	132	128
Q7	693	711	**630**	35,485	1,086	10,100	1,700	1,642	1,583
Geo. Mean	39	**14**	46	1,280	170	748	216	277	362

Table 4.6: LUBM-160 – Query processing times (in *ms*)

thus showing a negative impact on the centralized execution. Still, TriAD-SG benefits from join-ahead pruning and delivers a much better performance than the other systems except for query $Q7$. In this case, like in the LUBM-10240 dataset, TriAD-SG performs no pruning in the first stage and thus the overhead in the second stage marginally decreases its performance.

Impact of Summary Graph. The number of summary graph partitions directly affects the performance of the system as highlighted in Table 4.7. With a smaller number of partitions, each supernode comprises of many triples. Thus, even though the join-ahead pruning can be done quickly over smaller summary graphs, due to large supernode sizes, the overall number of pruned tuples remains low, thus making the second-stage query processing considerably more expensive. Thus, the right choice of the summary graph size has a crucial impact on the overall performance.

Summary size	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geo-.Mean
10K	153	141	30	0.8	0.5	1.5	692	16
17K	97	140	31	0.7	0.2	1.8	711	14
20K	86	140	36	0.9	0.5	1.5	702	16

Table 4.7: Impact of summary graph partitions for LUBM-160

Graph Exploration vs Relational Joins. Finally, we compared three approaches for processing the summary graph: (1) full graph exploration (Full GE), (2) 1-hop graph exploration (1-hop GE), (3) a conventional form of relational joins (RJ). Table 4.8 shows the runtime performance of the three approaches over a summary graph with 17K partitions for the LUBM 160 dataset. We can clearly observe that, in a relational approach, there is a penalty incurred for generating large intermediate relations. This is avoided entirely in graph exploration (Full GE) without increasing the number of false-positive bindings. On the other hand, as expected, the 1-hop exploration performs faster than the full exploration (Full GE) for the complex queries Q1,Q3,Q7, but it also retains a lot of false positives, which in turn makes the second stage of processing more costly.

Approach	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geo-.Mean
1st Stage (*times in ms*)								
Full GE	22	3	22	0.03	0.003	0.5	82	1.3
1-hop GE	13	4	13	1.4	0.3	1.8	17	3.9
RJ	312	6	312	1.4	0.4	1.6	767	16.3
1st + 2nd Stage (*times in ms*)								
Full GE	97	140	31	1	0.2	1.8	711	14
1-hop GE	412	139	75	3.1	0.8	2.5	712	29
RJ	312	137	321	2.3	0.9	3	1447	40

Table 4.8: Performance of 1-hop and full graph exploration (GE) vs. relational joins (RJ) in TriAD-SG for LUBM-160

4.6.2.3 BTC 2012 Dataset

Apart from the synthetic LUBM benchmark dataset, we evaluated TriAD over the real-world BTC benchmark. We considered queries $Q1$–$Q8$ published in (Neumann and Weikum, 2009). Queries $Q1$, $Q2$, $Q8$ (4 joins), $Q3$ (5 joins) are star queries with result sizes of 1, 2, 1, 292, respectively. Queries $Q4$, $Q7$ (6 joins) and $Q5$, $Q6$ (4 joins) are combinations of star and path queries. Table 4.9 shows the performance of TriAD against the available competitors. (We omit SHARD and BitMat from the table as they failed to finish the indexing step.) We can observe that TriAD consistently outperforms the competitors. In the case of $Q6$, which has an empty result, our summary graph returns no bindings and thus entirely avoids query processing against the data graph. Also, one can observe the high running times for H-RDF-3X compared to RDF-3X. The reason again lies in breaking the queries into smaller queries, such that the SIP gains of RDF-3X remain under-utilized.

	#Results	TriAD	TriAD-SG	H-RDF-3X		RDF-3X	
			(200K)	*(cold)*	*(warm)*	*(cold)*	*(warm)*
Q1	1	1.5	**0.3**	49	6	297	4
Q2	1	61	**3**	29	6	140	5
Q3	1	**1**	4	122	23	66	5
Q4	0	**0.6**	6	31,033	27,415	120	7
Q5	5	51	**5**	1.3E5	42,638	277	104
Q6	0	0.5	<**0.1**	5,476	153	53	24
Q7	0	50	**39**	89,922	34,906	2,900	2,386
Q8	292	128	**7**	1,338	**7**	4,590	31
Geo. Mean	–	7.4	**1.5**	2,145	280	299	25

Table 4.9: BTC 2012 – Query processing times (in *ms*)

4.6.2.4 WSDTS Dataset

We finally evaluated the performance of TriAD and TriAD-SG over the more diverse WSDTS dataset which consists of about 109 million triples. We generated 20 queries using the WSDTS query generator and categorize them into L (long path), S (star), F (snowflake) and C (complex). Table 4.10 shows the performance

	#Slaves	L1-L5 (Geo.-Mean)	S1-S7 (Geo.-Mean)	F1-F5 (Geo.-Mean)	C1-C3 (Geo.-Mean)
TriAD	1	**2**	**2**	94	494
TriAD-SG(75K)	1	8	4	35	767
TriAD	5	**2**	3	**29**	**270**
SHARD	5	3.2E5	5.8E5	7.1E5	7.7E5
RDF-3X (cold)	1	10,066	167	1,749	6,610
RDF-3X (warm)	1	18	**2**	41	354
MonetDB (cold)	1	3530	10,459	timeout	timeout
MonetDB (warm)	1	171	744	timeout	timeout

Table 4.10: WSDTS-1000 – Query processing times (in *ms*)

over TriAD and TriAD-SG against the available competitors. We can observe that TriAD continues to perform well for all query categories, especially for long path (L) and complex queries (C). On the other hand, TriAD-SG with summary-based pruning performs well for class F queries. For L, S, C class queries, TriAD-SG indeed shows some overhead due to its additional summary-graph processing. The performance dip of TriAD-SG here seems to be due to the dense nature of the WSDTS data graph and the lack of constants (besides predicates) in the SPARQL queries. MonetDB failed to finish $S1$, $F1$–$F5$, $C1$–$C3$ within a 10-minute limit (marked as "timeout").

4.7 Summary

In this chapter, we investigated a distributed approach to process BGP queries in an efficient and scalable manner. To this end, we presented TriAD, a distributed architecture, which combines intra-node multi-threading with asynchronous inter-node communication for the scalable processing of BGP queries expressed in SPARQL 1.0. TriAD consistently outperforms both centralized and distributed RDF engines, which so far still largely rely on Hadoop-based joins, in which multiple join operators may indeed run in parallel but need to be synchronized at each level of the query plan before the next iteration of Hadoop-based joins is initiated. Especially our comparison to a single Map-side join in Apache's Hadoop and Spark platforms reveals the overhead of the Map and Reduce paradigm for such a very basic query operation.

Chapter 5

Generalized Graph Patterns

Generalized graph patterns (GGP) is a generalization of the BGP query model (discussed in Chapter 4) with navigational semantics extension. A typical pattern in a GGP query is of the form $\langle u, e, v \rangle$, where each of u, e, v can be either a constant or a variable. Moreover, e can denote a regular expression over the edge label set Σ_E, denoting a set of paths (see Section 2.1.1.4) as against a set of edges as in a BGP query. In other words, a GGP pattern extends the set-reachability query model (discussed in Chapter 3) with regular expressions. Due to the lack of schema for many real-world graph datasets, GGP queries are often found to be a better alternative to BGP queries in expressing user needs. Hence, most general purpose graph query languages such as Cypher (Neo4j, 2012), GraphLog (Consens and Mendelzon, 1990), and recently SPARQL, with the introduction of *property paths* in its 1.1 update (Prud'hommeaux et al., 2013), started to support GGP query model.

In this chapter, we focus on the efficient processing of GGP queries on distributed labeled directed multi-graphs. As in the case of the BGP query model, we specifically target conjunctive GGP queries, where we allow only the set operation "AND" among the query patterns. We continue to use RDF (Hayes, 2004), a W3C recommendation, as the representative framework for our data model and chose SPARQL 1.1 (Prud'hommeaux et al., 2013), a recently updated and W3C recommended language, as the query language for expressing GGP queries. However, the techniques discussed in this chapter are general enough to be applied to other representations of data and query models. We propose an extension to our TriAD architecture (discussed in Chapter 4) to tackle GGP query model. The modified system combines the indexing and query processing methods introduced in the previous chapters for BGP and set reachability query models. With only a few state-of-the-art systems that provide a native support for GGP queries available, we evaluated our approach, implemented in TriAD, against them on multiple real-world and synthetic datasets. On an empirical analysis, our approach was able achieve impressive gains over its counterparts on almost all datasets and query workloads.

5.1 Introduction

5.1.1 Motivation

One of the main limitations of the BGP query model is the lack of navigational semantics for the better expressivity and the concise representation of user query needs, especially when the schema of the underlying graph is either unknown or partially known. For instance, consider an user intent to

"Find all persons who are born in the USA".

If in the underlying graph, the schema states that the person vertices are connected to the vertex "USA" via "bornIn" and by a series of zero or more "locIn" edge labels, expressing the above intent as a BGP query is either impossible or impractical. To fill this gap, on the other hand, generalized graph pattern (GGP) queries allow a single (generalized) pattern to express paths concisely. Recalling that a typical pattern in GGP query is of the form $\langle u, e, v\rangle$, where each of u, e, v can be either a constant or a variable, and moreover, e can denote a regular expression over the edge label set Σ_E, the above user intent can be concisely expressed as a pattern in GGP query as shown below.

$$\langle ?persons, \mathsf{bornIn/locIn}^*, \mathsf{USA}\rangle.$$

Moreover, GGP queries, like BGP queries, allow more than one pattern to be part of a single query. For instance, the above user intent can be alternatively expressed as a multi-pattern GGP query as follows.

$$\langle ?persons, \mathsf{bornIn}\ ?city\rangle$$
$$\langle ?city, \mathsf{locIn}^*, \mathsf{USA}\rangle.$$

Processing a single GGP pattern such as "?*city* locIn* USA", where u :=?*city* is a query variable, v := "USA" is a constant, and e :="locIn*" is a regular expression, resolves to finding the set of all vertex bindings for u, such that from each instance of u there exists at least one path to v containing only the edges with label "locIn". In other words, on a "locIn" edge-induced subgraph G^{locIn}, the pattern translates to finding all vertex instances of u in G^{locIn} that are reachable to v, i.e., a set reachabilty query "$V^{\mathsf{locIn}} \rightsquigarrow \{\mathsf{USA}\}$" on G^{locIn}, where V^{locIn} is the vertex set of the graph G^{locIn}. On the other hand, processing a multi-pattern GGP query requires a combination of set reachability and as well as relational join operations to, respectively, handle navigational and pattern matching aspects of a GGP query. Due to the fact the generalized patterns are similar to the SIMPLE PATH queries described in (Mendelzon and Wood, 1995), the complexity of processing a single generalized pattern in GGP queries has the polynomial data complexity. Moreover, conjunctive multi-pattern GGP, like BGP queries, also falls under the polynomial data complexity and non-polynomial expression (query) complexity (Chandra and Merlin, 1977; Vardi, 1982).

Applications. GGP queries are often seen to be a better alternative to BGP queries in many application scenarios. Here, we list some of the applications that rely on GGP querying model.

- *Knowledge graphs.* Knowledge graphs (KG) are increasingly popular among web search applications, NLP architectures, etc. A KG typically comprise of facts about real-world things, such as "Barack_Obama bornIn Honolulu", and as well as ontological facts, such as "Physicists subclassOf Scientists" in a canonicalized form. As seen from the examples, a fact connects two entities by a relation, e.g. bornIn is a relation between two entities Barack_Obama and Honolulu. Some of the relationships, such as locatedIn, parentOf, subClass of, etc., are transitive relations. Applications that rely on knowledge graphs often need to handle transitive relations as part of the query processing. With its recent update, SPARQL, a de facto language for querying KGs represented in RDF, introduced *property paths* (Prud'hommeaux et al., 2013). Property paths allow for annotating pairs of query vertices by regular expressions in which properties and entire paths may be marked by a Kleene "+" or "*", thus, introducing a notion of generalized graph patterns.
- *Social networks.* Another line of applications where GGP queries are one-of-the or the-only choice of querying is in the domain of social networks such as Facebook [1], Twitter [2], LiveJournal [3], etc, where majority of relationships such as "friend", "follower", "following", etc. are transitive. Social search and analytics applications query such transitive relations by expressing them as GGP queries, which are supported by general purpose languages like Cypher (Neo4j, 2012), SPARQL 1.1 (Prud'hommeaux et al., 2013), GraphLog (Consens and Mendelzon, 1990), etc.
- *Biological networks.* Biological networks such as protein-protein interactions (PPI), gene regulatory networks, metabolic networks, etc. are some of the real-world graph datasets, where GGP queries are often seen as a query model to study and analyze the biological structures.

Scope. In this chapter, we limit the scope of the problem to conjunctive GGP queries, where the patterns are either a BGP or a navigational pattern adhering to the grammar specified in Equation. 5.1, thus, covering a broad range of real-world query needs. Specifically, we consider RDF & SPARQL 1.1, due to their popularity, as the representative languages for our data and query model, and focus on the problem of distributed processing of conjunctive GGP queries, expressed in SPARQL 1.1, over large RDF datasets.

Challenges. Efficient and scalable processing of GGP queries in a distributed setting requires addressing some of the key challenges discussed below.

[1] http://facebook.com
[2] http://twitter.com
[3] http://snap.stanford.edu

1. Single-Pattern Processing. A generalized graph pattern is the basic building block of a GGP query representing a navigational query. The pattern can be translated to a form of set reachability query over labeled directed multi-graphs. While, Chapter 3 presents an efficient technique for processing set reachability queries over labeled directed graphs, where edges are unlabeled, processing set reachability queries over labeled directed multi-graphs with regular-expression constraints has not been addressed so far in the literature, albeit the work (Fan et al., 2012) addressing only the single-source single-target reachability. This poses an interesting challenge in efficient processing of a single pattern GGP query.

2. Unified Query Processing. As aforedescribed, typical GGP queries comprise of more than one graph pattern, each translates to a form of set reachability query. As noted in Chapter 3, such set reachability queries can be best processed using graph-based navigational approaches. While GGP queries, *alike to* BGP queries, are often expressed in an SQL style declarative graph query languages such as Cypher, SPARQL, etc., where a row-oriented output is required, relational joins are inevitable in processing GGP queries. An interesting challenge, thus, lies in the combined optimization of relational joins among the patterns (based on shared variables) and graph-based set reachability processing for each pattern.

3. Unified Query Optimizer. An optimal join ordering is one of the crucial factors impacting the query performance of a relational engine. TriAD employees a cost-based query optimization to find an efficient plan for BGP queries. GGP queries, on the other hand, inherently require a graph-based exploration for processing a pattern alongside relational joins for processing a set of triple patterns. A challenging task lies in designing a distributed cost-based query optimizer that generates an efficient query plan with interleaving graph-exploration and join operations by considering both the locality of edges and the cost of individual operators.

5.1.2 State-of-the-art

As opposed to the large variety of BGP query engines, processing of GGP queries in the context of RDF & SPARQL 1.1 so far has been investigated by only very few approaches (Erling and Mikhailov, 2010; Gubichev et al., 2013; Przyjaciel-Zablocki et al., 2012) (of which only (Erling and Mikhailov, 2010) is available), and Horton, Horton+ (Sarwat et al., 2012, 2013) in the context of social networks. Distributed graph engines, such as Berkeley's GraphX (Gonzalez et al., 2014), Apache Giraph (Martella et al., 2015), Microsoft's Trinity (Shao et al., 2013), on the other hand, *can be programmable* to allow for the scalable processing of graph queries over massive, partitioned data graphs. These engines provide generic API's for

implementing various kinds of graph queries, including navigational and basic graph pattern queries that are part of the GGP query model. However, they do not support the kinds of indexing techniques known from the centralized approaches, and are not directly amenable to the declarative style of querying which used in graph query languages like SPARQL, Cypher, etc., albeit in an inefficient way. However, processing a single generalized pattern in a GGP query, under these frameworks, require an iterative form of graph traversal. This may result in as many iterations (and hence communication rounds) as the diameter of the graph in the worst case.

5.1.3 Our Approach & Contributions

5.1.3.1 Our Approach

To address the challenges in processing GGP queries, we propose a novel solution, to fill the gap between the distributed relational engines and the graph engines that, in standalone, efficiently tackle the pattern matching and the navigational queries respectively. As our focus is on RDF & SPARQL, where an SQL style row-oriented output is required, our approach relies on the relational semantics, much like in BGP query model. Our solution, implemented in TriAD, comprises of triple indexes, for efficient processing of BGP queries, and graph reachability indexes, to process property paths. Relying on our previously developed techniques (Chapter 4 and Chapter 3), in our approach, we further adapt the earlier techniques to process GGP queries in an efficient and scalable manner. We also propose a novel, unified and distribution-aware query optimizer that generates efficient query plans which are optimized for GGP queries.

5.1.3.2 Contributions

We summarize the contributions of this chapter as follows.

- We present an *efficient* and *scalable query engine* for processing GGP queries. Specifically, we consider GGP queries to comprise of sets of triple patterns with labeled regular expressions, which allows for formulating the queries as conjunctive queries of relational joins with additional set-reachability predicates (including one or more properties marked by a Kleene "+" or "$*$").
- We provide a *unified indexing scheme*, *cost model* and *query optimization framework* to seamlessly integrate set-reachability predicates into the relational query processor of our TriAD engine (presented in Chapter 4). TriAD employs a strictly fixed, asynchronous message-passing protocol to evaluate a GGP query among all of the compute nodes in parallel. Our protocol requires exactly one round of communication per set reachability predicate and thus avoids a costly, iterative form of communication.
- Our approach is the first to report a *true scale-out* in processing GGP queries over a number of large RDF collections. We present a detailed experimental

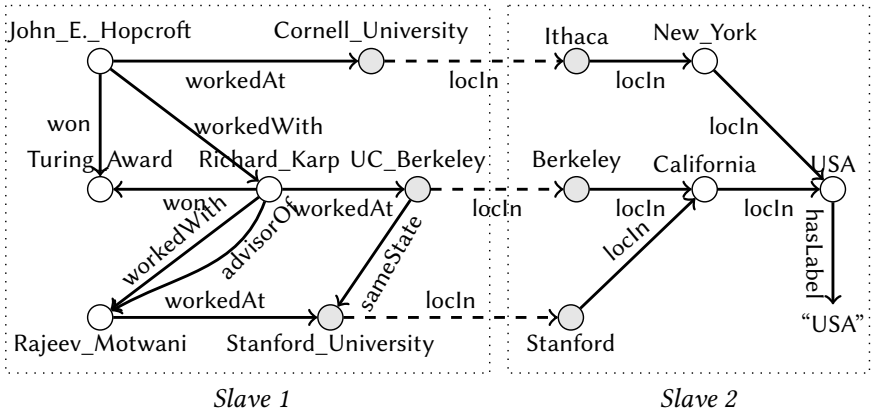


Figure 5.1: An example of RDF

evaluation of our approach over multiple RDF datasets under both strong and weak scaling, and in comparison to the Virtuoso native RDF store.

5.2 Preliminaries

This section serves to establish some of the key notations for our data and query model used in this chapter, and also reviews some of the related works for solving GGP queries, both in centralized and distributed settings.

5.2.1 Data & Query Model

As in for BGP queries, we consider a *labeled directed multi-graph* (see Section 2.2.1.1) as the underlying data model in this chapter. Following Definition 2.3, we denote the input labeled directed muti-graph as $G(V, E, \Sigma_V, \Sigma_E, \Phi)$ and, for brevity, refer to it as just the "*graph*".

We further assume that the graph G is partitioned into a k vertex-disjoint partitions, $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$, where each of G_i is a subgraph of G and $\mathcal{G}$ is called a partitioning of G. We refer to a vertex u as *local boundary* of G_i if there exists an edge $(u, v) \in E$ or $(v, u) \in E$ and $u \in V_i$ and $v \in (V_D - V_i)$. Specifically, u is labeled as *in-boundary* (or *out-boundary*) of G_i if (v, u) (or (u, v)) $\in E$ and $u \in V_i$ and $v \in (V - V_i)$. For each partition G_i, the sets I_i and O_i denote the in- and out-boundaries of all G_i, respectively.

In addition to the graph partitioning $\mathcal{G}$, we refer to G_C $(V_C, E_C, \Sigma_V, \Sigma_E, \Phi)$ as *Cut, vertex-induced subgraph* of G. Where, V_C consists of the union of all in- and out-boundaries,i.e., $V_C = I_i \cup O_i$, of the graph partitions G_i, for $i = 1 \ldots k$.

Example 5.1. *Figure 5.1 shows an example RDF data graph that is partitioned into the two graph partitions G_1, G_2 which are located at slaves 1 and 2, respectively. The gray-shaded vertices represent boundary vertices and the remaining ones represent local vertices. For the first graph partition G_1, the out-boundaries are O_1 = {UC_Berkeley, Stanford_University, Cornell_University} and the in-boundaries are*

$I_1 = \emptyset$. Similarly, for partition G_2, we have $O_2 = \emptyset$ and $I_2 = \{$Berkeley, Ithaca, Stanford$\}$. Here, the cut G_C consists of the three dashed edges.

Generalized graph patterns (GGP), defined in Section 2.2.2.3, constitute the query model of this work. Following Definition 2.7, we denote GGP query as $Q(V_Q, E_Q, \Sigma_V, \Sigma_E, \mathcal{V}, \mathcal{L}, \Phi_Q)$, where V_Q comprises of set of query vertices and the edge set E_Q comprises of a set of triple patterns of the form $\langle u, e, v \rangle$. Here $u, v \in V_Q$ are query vertices, where each of u, v can be either a constant or variable. Function Φ_Q is an injective mapping from V_Q to $\{\Sigma_V \cup \mathcal{V}\}$. While e is an query edge label that can be either a variable in $\mathcal{V}$ or a regular expression from language $\mathcal{L}$ defined over alphabet Σ_E.

We consider SPARQL 1.1 as the representative query language for expressing GGP queries, and specifically focus on the subset of SPARQL 1.1 specification. As discussed in Section 2.2.2.3, we restrict the *property regular expressions language* to the adhere to the following grammar and, henceforth, refer the new query language as SwPP ("*SPARQL 1.0 with Property Paths*").

$$
\begin{aligned}
\text{path} &:= \text{path/path} && (\textit{concatenation of paths}) \\
&:= \sigma && (\textit{single edge element}) \\
&:= \sigma? && (\textit{zero or one edge element}) \\
&:= \sigma* && (\textit{zero or more edge element}) \\
&:= \sigma+ && (\textit{one or more edge element})
\end{aligned} \tag{5.1}
$$

We hereby adopt a simpler definition for property paths than the full syntax proposed by the W3C (Prud'hommeaux et al., 2013). However, by rewriting an entire path expression (denoted as "path" in the above grammar) into a sequence of join conditions, each with a property that denotes a single URI (referred to as "σ") with an optional Kleene "*" or "+", we allow a more general syntax for property paths rather than just a single transitive property. The above grammar in particular allows concatenations of properties into paths of arbitrary length, as long as these can be rewritten into a conjunction of triple patterns and thus conform to Definition 2.7. In our implementation, the distinction between "+", "*" and "?" is very simple. For "+", we merely disallow an equality between a source and a target vertex; while for "?", we restrict the maximum path length to 1.

Example 5.2. *For instance, the query "*Find all professors who won the Turing Award and worked in a US university*" can be specified as a BGP query via SPARQL 1.0 as follows.*

```
SELECT ?person
WHERE { ?person won Turing_Award.
        ?person workedAt ?univ.
        ?univ locIn ?city. ?city locIn ?state.
        ?state locIn ?country. ?country hasLabel "USA" }
.
```

The above query can thus concisely be rewritten into a GGP query via SwPP query language as follows.

```
SELECT ?person
WHERE { ?person won Turing_Award.
        ?person workedAt/locIn*/hasLabel "USA"}
```

5.2.2 Related Work

In this section, we briefly discuss some of the works that we believe are most related to our work in this chapter.

RDF & SPARQL 1.1. Combining relational joins with reachability predicates in SPARQL 1.1 inherently leads to a multi-source, multi-target graph-reachability problem. Very few works so far focused on the combined optimization of relational joins with additional graph-reachability predicates (Cheng et al., 2007; Fan et al., 2014a; Gubichev et al., 2013). In earlier works, the bisimulation-based indexing of path expressions for XML trees (Milo and Suciu, 1999) has been extended to RDF graphs (Picalausa et al., 2012), but the latter did not yet consider property paths. Likewise, (Cheng et al., 2007) proposed an index structure that is limited to DAGs obtained from XML/XLink. (Przyjaciel-Zablocki et al., 2012) finally investigated an initial approach to evaluate property paths via MapReduce. Virtuoso (Erling and Mikhailov, 2010) is a relational backed RDF engine, in its recent update supported SPARQL 1.1 queries. Virtuoso relies on relational joins to process generalized patterns in GGP queries and can process them in a distributed setting.

Conjunctive Regular Path Queries. GGP queries can be alternatively thought as Conjunctive Regular Path Queries (CRPQs). CRPQs has been studied well in literature (Consens and Mendelzon, 1990; Mendelzon and Wood, 1995; Florescu et al., 1998; Calvanese et al., 2000, 2002; Deutsch and Tannen, 2002; Calvanese et al., 2003; Libkin et al., 2013). Neo4j (Neo4j, 2012) is a recent transactional graph database system that supports CRPQs expressed in Cypher or Gremlin query languages. Neo4j is majorly a centralized system with minimal distributed functionality. Mircosoft's Horton+ (Sarwat et al., 2013) was one of the recently proposed distributed system and is most related to our work. Horton+ supports CRPQs as part of its query language and relies on bulk synchronous processing (BSP) paradigm like Google's Pregel (Malewicz et al., 2010), Apache Giraph (Martella et al., 2015) to process a CRPQs. Unlike ours, Horton+ process queries in an iterative model, though scalable, is not efficient to process queries in real-time. On the similar lines G-Path (Bai et al., 2013) uses BSP to process CRPQs over Hadoop system and like Horton+ is not ideal for real-time processing of queries.

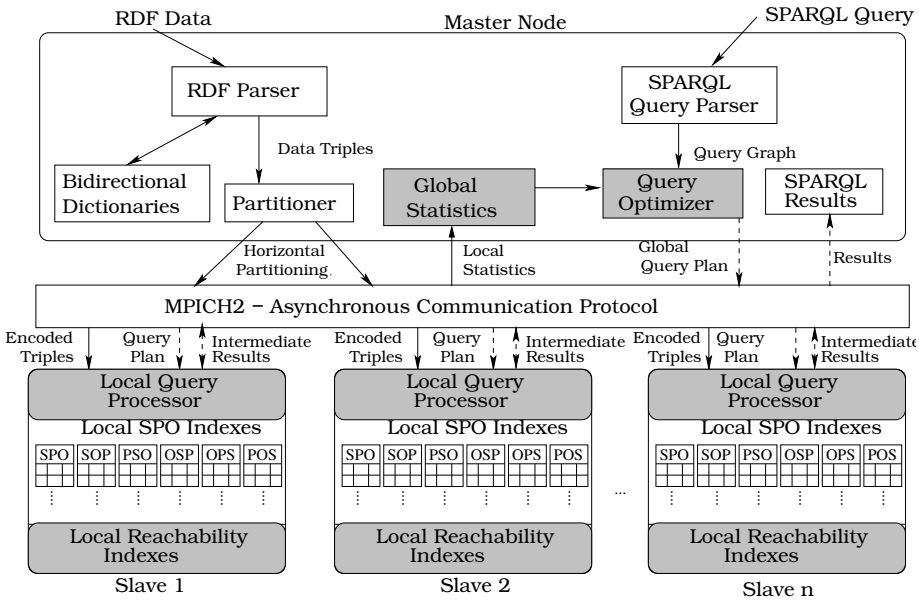


Figure 5.2: Architecture of modified TriAD RDF System to support GGP queries

5.3 System Architecture

To better describe our approach, we here present the extensions made to the architecture of our TriAD engine (discussed in Chapter 4) to process GGP queries. As GGP queries encompass basic-graph and generalized patterns, original TriAD's triple indexes are not sufficient to efficiently process navigational part of the generalized patterns, i.e., property paths in SwPP queries. We thus augment the triple indexes with a set of local graph reachability indexes. Having bimodal indexes are just not sufficient to efficiently process GGP queries, until we adapt all the borrowed modules such as query optimizer, local query processor, statistics, etc. from the original TriAD architecture. Figure. 5.2 depicts the architecture of modified TriAD engine that supports GGP queries. We below highlight the changes made in TriAD (shown in shaded regions in Figure. 5.2) to get an overview; subsequent sections present more details of our approach.

Master Node

Global Statistics. TriAD's cost-based query optimizer relies on statistics to generate efficient plans. In original TriAD, we collect multiple single and pair cardinalities along with the selectivities of the pair-wise predicates. This is sufficient to estimate the cost of basic graph patterns in GGP queries. To estimate the cost of generalized patterns with navigational properties, we collect reachability statistics and stored at the master node. More details about statistics are discussed in Section 5.4.2.

Query Optimizer. We extend the TriAD's cost-based query optimizer by taking into account both the cost of processing basic and generalized graph patterns. This helps to generate an efficient plan (called "*operator-tree*") with interleaving relational joins and set reachability operations. Section 5.5 discusses more details about the query optimizer module.

Slaves

Local Reachability Indexes. In addition to local triple indexes, we build local reachability indexes based on the approach presented in Chapter 3. The local reachability indexes are designed to efficiently tackle generalized patterns of a GGP query, while triple indexes are used for processing basic graph patterns.

Local Query Processor. The query plan (an "*operator-tree*") returned by the query optimizer at the master node is communicated to all slaves. Each slave performs a bottom-up execution of the operator-tree, which comprises of relational and reachability join operators along with the index scans at the leaves. Index scans and relational joins are executed over the triple indexes, while special reachability joins are executed over the customly built reachability indexes. Moreover, all these operations leverage the efficiency of multi-threaded and distributed execution framework that existed in TriAD.

5.4 Index Organization

5.4.1 Local Indexes

In this section, we discuss the details of indexing layout designed for processing GGP queries in TriAD. As discussed in previous section, we use a bimodal indexing layout with specialized indexes for basic-graph and generalized patterns respectively. An RDF fact $\langle subject, predicate, object \rangle$ first encoded into an integer format $\langle s, p, o \rangle$, using a dictionary based encoding (discussed in Chapter 4), is indexed as follows.

Sharding. The encoded triple $\langle s, p, o \rangle$ is first distributed to (at most) two slaves i, j by choosing $i = (s \mod k)$ and $j = (o \mod k)$ as sharding conditions, respectively. Although we use a simple hash for sharding triples, any partitioning schemes such as METIS (Karypis and Kumar, 1998) can be employed in our framework.

Local Triple Indexes. Each slave maintains a six permutation triple indexes grouped into *subject-key* (SPO, SOP, PSO) and *object-key* (OSP, OPS, POS) indexes. These indexes are used to process basic graph patterns in GGP queries. To recap, at slave i, the encoded triple $\langle s, p, o \rangle$ is indexed using either subject-key indexes if $i = (s \mod k)$, or object-key indexes if $i = (o \mod k)$. These indexes are then sorted lexicographically to facilitate merge joins at the lower levels of operator tree. For more details about the triple indexes, please refer Section 4.4.

Reachability Indexes. In order to process generalized patterns with navigational semantics, local reachability indexes are constructed at each slave. These indexes constitute a group of set reachability indexes discussed in Chapter 3, built one each for a property label p. To exemplify, at slave i, the encoded triple $\langle s, p, o\rangle$ is added as an edge (s, o) to the local subgraph G_i^p. This forms a partitioning $\mathcal{G}^p = \{G_1^p, G_2^p, \ldots, G_k^p\}$, where each G_i^p denotes a p edge-induced subgraphs of an input graph G. We then construct compound graph C_i^p at each slave using the approach discussed in Chapter 3. Any off-the-shelf centralized indexes like FERRARI (Seufert et al., 2013), GRAIL (Yildirim et al., 2010), MS-BFS (Then et al., 2014) can be used over the compound graphs C_i^p. These reachability indexes built at each slave facilitates efficient processing of generalized patterns such as $\langle ?x, \mathrm{p}^*, ?y\rangle$, $\langle ?x, \mathrm{p}+, ?y\rangle$ for the property label p.

5.4.2 Index Statistics

To optimize GGP queries consisting of both relational joins among basic graph patterns and of generalized patterns (property paths), we extend the cost-based plan generator which is part of TriAD's architecture. For this, we collect various statistics over the RDF data, both for the basic triple patterns and triple patterns with property paths.

Statistics for Triple Patterns. As in (Gurajada et al., 2014a), our statistics for basic triple patterns include:

1. cardinalities $Card(R_i)$ of relations R_i induced by individual *subject*, *property* and *object* keys, and
2. of relations induced by *subject-object*, *property-subject* and *property-object* pairs

In addition, we compute the join selectivities $Sel(p_i, p_j)$ of all *pairs of properties* p_i, p_j to estimate the cardinality of a join among two triple patterns.

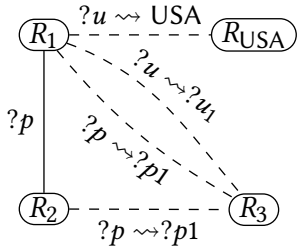
Statistics for Property Paths. In order to plug triple patterns with property paths into our optimizer, we need to also estimate the selectivity of a property path. Analogous to the selectivity in relational systems, selectivity of a propery p, i.e., $Sel(p)$ is computed as follows. Let G^p be the p edge-induced subgraph of G, then

$$Sel(p) = \frac{|R|}{V^p \times V^p}$$

where, R is the set of reachable pairs for the set-reachability query $V^p \rightsquigarrow V^p$. As it can be seen from the above equation, precomputing these selectivities for every possible property path that may occur in a query is clearly intractable. We thus follow a simple sampling-based approach. For each individual property p, we take a randomized sample for sources and targets sets and determine the *reachability selectivity*, $Sel(p)$, as the fraction of randomly sampled source and target vertices (s, t), for which $s \rightsquigarrow t$ holds with respect to the subgraph G^p.

```
SELECT * WHERE {
P1:  ?p workedAt ?u.
P2:  ?p won Turing_Award.
P3:  ?p1 workedAt ?u1.
P4:  ?u locIn* USA.
P5:  ?p workedWith* ?p1.
P6:  ?u sameState* ?u1.}
```

(a) SwPP Query



(b) Query Graph

Figure 5.3: Example SwPP query and its query graph representation

5.5 Query Optimization & Distributed Processing

To evaluate GGP queries, expressed as SwPP queries and as they occur in SPARQL 1.1, we extend TriAD's distributed query processing architecture to support generalized patterns with property paths. We begin with the translation of an SwPP query into a query-graph representation. Edges in this query-graph express join conditions as either exact-match conditions on the subjects or objects of two patterns, or as reachability predicates which each express a connection among a subject and object via a property path. We employ a bottom-up dynamic-programming (DP) based plan generator to enumerate the possible query plans over these join conditions. For query optimization, we rely on our previously collected statistics to compile a logical query plan with the lowest cost estimate. The obtained query plan is broadcast to all slaves, which then all execute the same plan in parallel, but each over a different partition of the sharded triple- and reachability-indexes.

5.5.1 Translation of GGP Queries

Similar to the translation of BGP queries expressed in SPARQL 1.0 queries, SwPP queries are first translated into a graphical representation for optimization. These query graphs are generated by introducing a vertex for each triple pattern (thus representing a relation) in the query, while the edges that connect two such vertices represent equi-joins. These equi-joins are based on the variables (i.e., either the subjects or objects) that are shared by two such triple patterns. Edges for equi-joins are labeled with the shared variables.

In addition to SPARQL 1.0 queries, SwPP queries contain triple patterns with property paths. Following (Gubichev et al., 2013), we represent a property path by a distinguished edge among two such query nodes, whose labels denote the reachability predicates among the subjects or objects in the respective nodes' triple patterns. In case the subject or object of a connected triple pattern is either a constant or an unbound variable (i.e., the variable is not present in the other triple patterns), we create a new query node for the same and add an edge between the respective query nodes.

Example 5.3. *Figure 5.3 shows an example SwPP query and its corresponding graph representation. Here, patterns P_1, P_2, P_3 are basic graph patterns (BGP) whose property each consists of a single URI. These are represented as nodes R_1, R_2, R_3, respectively, in the query graph. Since, the property path of P_4 points to only the constant USA, a separate node R_{USA} representing a relation consisting of just a singleton tuple is added to the query graph. The equi-join on the shared variable ?p is represented by the continuous line between R_1 and R_2. A reachability edge, denoted by a dashed line for each property path, is added between the respective subjects' and objects' query nodes. This is the case between R_1 and R_{USA} for the property path of P_4, between R_1 and R_3 for the property paths of P_5 and P_6, and between R_2 and R_3 for P_5.*

5.5.2 Plan Optimization

Once the query is translated into its graph representation, classical join-order-enumeration techniques (Gurajada et al., 2014a; Neumann and Weikum, 2010a) can be employed to find a cost-efficient execution plan. We extend TriAD's optimizer to handle SwPP queries by adding a new operator—*Distributed Reachability Join* (DRJ)—and respective cost estimator for property paths. Next, we briefly discuss these operators, which is followed by a discussion of the cost estimation and join-order enumeration.

5.5.2.1 Physical Query Operators.

TriAD employs three physical operators—coined Distributed Index Scan (DIS), Distributed Merge Join (DMJ) and Distributed Hash Join (DHJ)—for processing index scans and equi-joins among triple patterns in SPARQL 1.0. Each of these operators works over the sharded partitions of the triple indexes described in Section **??** in parallel. In short, the DIS operators, which only occur at the leaves of the query plan, each build a relation by invoking a parallel scan over the respective SPO permutation index that was selected by the optimizer. The DMJ and DHJ operators each take two sharded relations plus the join keys (i.e., the shared variables) as input and perform a hash- or merge-join, respectively, to generate a new intermediate relation.

Distributed Reachability Join (DRJ). Analogously, we define a new DRJ operator to process triple patterns with property paths. This enhanced join operator takes two sharded relations R_i, R_j as input and returns as output the subset of tuples in the cross-product $R_i \times R_j$, for which all of the attached *join conditions* $\mathcal{C}$ hold:

- for each shared variable ?x in $\mathcal{C}$, a pair of tuples in R_i and R_j must have equal values for ?x; and
- for each reachability predicate ?x $\rightsquigarrow$?y in $\mathcal{C}$, a vertex s that becomes bound to ?x by a tuple in R_i must be reachable to a vertex t that becomes bound to ?y by a tuple in R_j.

The evaluation of the DRJ operator is backed by the index structures for property paths described in Section 5.4.1.

5.5.2.2 Query Optimization.

The DP table of the optimizer is initialized with the cost estimates for the DIS operations of each query vertex R_i. The scan costs for R_i depend on whether the constants in the triple pattern match a respective SPO permutation index "*idx*". For instance, if the subject and predicate are constants and the object is a variable, choosing an SPO or PSO permutation costs much less compared to any of the remaining permutations.

In the query graph, we introduce two special kinds of query vertices, namely one for property paths with a constant subject or object, and one for property paths with (at least one) unbound variable. Scanning a singleton tuple as input has a unit cost of 1, while scanning a relation constructed for an unbound variable corresponds to the number of triples in the subgraph G^p of G that is induced by property p. In the latter case, we thus set the cardinality to $Card(R_i)$ of a unary relation R_i that is constructed from all vertices in G^p to $|V^p|$, where V^p is the vertex set of p edge-induced subgraph G^p. As an example, consider the property path $?x$ locIn$*$ $?y$, and let variable $?y$ be unbound (i.e., not occurring as a shared variable in any other triple pattern). Then the number of unique bindings for $?y$ is the number of vertices in the edge-induced subgraph consisting only of locIn edges. To summarize, we have,

$$Card(R_i) := \begin{cases} 1 & \text{if } R_i \text{ is a singleton tuple;} \\ Card(R_i^{idx}) & \text{if } R_i \text{ matches the SPO index } idx; \\ |V^P| & \text{if } R_i \text{ is a unary relation for the property } p. \end{cases} \tag{5.2}$$

Equation 5.3 summarizes the cost estimates we obtain for a DIS operator with respect to the precomputed cardinalities $Card(R_i)$ and available SPO permutations.

$$Cost(R_i) \propto \begin{cases} 1 & \text{if } R_i \text{ is a singleton tuple;} \\ Card(R_i)/k & \text{if } R_i \text{ is sharded across } k \text{ slaves} \end{cases} \tag{5.3}$$

Once the DP table is initialized with the costs estimates for the DIS operators, we continue to build the query plan in a bottom-up manner. At each DP step, we merge two branches Q^{left}, Q^{right} into a combined plan Q by a join operator *op* together with a set of join conditions $\mathcal{C}$. If there is at least one reachability edge between two relations R_i, R_j that connect Q^{left} and Q^{right}, a DRJ operator is employed. Assuming independence among the join conditions $\mathcal{C}$, we plug in our precomputed index statistics as follows.

$$Cost(Q^{left} \bowtie_{\mathcal{C}}^{op} Q^{right}) \propto \sum_{C_i \in \mathcal{C}} Card(Q_i^{left}) \cdot Card(Q_i^{right}) \cdot Sel(C_i) \tag{5.4}$$

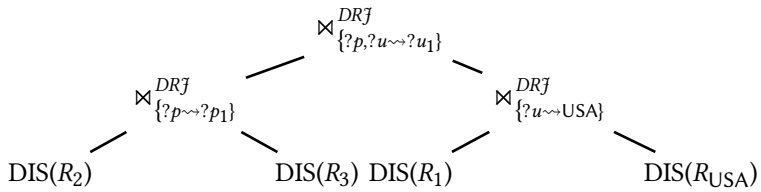


Figure 5.4: Example plan for the query of Figure 5.3

While processing the conditions $C_i \in \mathcal{C}$, we also iteratively estimate the cardinality $Card(Q_i)$ of a subquery Q_i of Equation 5.4 as defined next.

$$Card(Q_i) := \begin{cases} Card(Q_i) & \text{if } i = 1 \\ Card(Q_i) \cdot \prod_{j=1}^{i-1} Sel(C_j) & \text{if } i > 1 \end{cases} \tag{5.5}$$

Thus, if C_j is a graph-reachability predicate, $Sel(C_j)$ denotes the reachability selectivity $Sel(p)$ of the property p that is associated with C_j. If C_j refers to an equi-join, $Sel(C_i)$ denotes the precomputed join selectivity $Sel(p_i, p_j)$ for the pair of properties associated with the two triple patterns of the equi-join. The combined cost for a (sub-)query Q then is defined recursively.

$$Cost(Q) = \begin{cases} \max\left(Cost(Q^{left}), Cost(Q^{right})\right) \\ + \; Cost(Q^{left} \bowtie^{op}_{\mathcal{C}} Q^{right}) \\ + \; Cost(Q^{left} \rightleftharpoons^{op} Q^{right}) \end{cases} \tag{5.6}$$

Here, $Cost(Q^{left} \bowtie^{op}_{\mathcal{C}} Q^{right})$ denotes the cost of processing the join operator $op \in \{\text{DMJ}, \text{DHJ}, \text{DRJ}\}$ with Q^{left} and Q^{right} as operands and join conditions $\mathcal{C}$ (Equation 5.4). Likewise, $Cost\ (Q^{left} \rightleftharpoons^{op} Q^{right})$ accounts for the shipping costs that incur when the resharding of intermediate relations is required. The shipping cost is proportional to the size and width of Q^{left} and Q^{right}, respectively. Using $\max(\cdot, \cdot)$ as cost aggregation finally accounts for the parallel execution of the two branches (Gurajada et al., 2014a). Figure 5.4 shows an example query plan for the query of Figure 5.3.

5.5.3 Distributed Query Execution

We embed the new DRJ operator into TriAD's multi-threaded and asynchronous processing framework to support the distributed execution SwPP queries. The principal processing flow and communication protocol (Gurajada et al., 2014a) remain unchanged and merely require an additional initialization of the source and target vertices for the distributed set-reachability queries, which are now triggered by the DRJ operators at their respective positions in the query plan.

1. Scanning Base Relations. The leaves of the operator tree always represent distributed index scans (DIS). Each slave scans its local SPO permutation index and selects tuples according to the constants associated with the DIS operator. Due to the layout of our SPO indexes, this merely requires initializing an

iterator at the first tuple in a permutation list that matches the constants. For a DRJ operator with a reachability predicate, whose source or target is a single constant, a singleton relation is created directly from that constant. If the DRJ operator has a reachability predicate with an unbound variable via a property p, a (sharded) unary relation with the local vertices of V^p is created.

2. Query-Time Sharding. During the execution of the query plan, resharding of intermediate relations may be required to ensure the proper execution of joins (DMJ, DHJ) and set-reachability (DRJ) operations. With six SPO permutations, each DMJ operator requires sharding of at most one of its base relations at query time, while the DHJ operator requires sharding of at least one of its intermediate relations, depending on the locality of the tuples with respect to the join key. Sharding for the DRJ operator depends on the locality of the join keys based on shared variables (if present) and the locality of the vertices that become bound to the source and target variables of the reachability predicates. Thus, resharding may be required for both input relations of a DRJ operator.

3. Parallel Execution of Operators. In addition to the concurrent execution of the operators across the slaves, each slave also locally pursues the execution of the query plan in a multi-threaded fashion. Starting from the leaves of the query plan, all operators are locally executed in one separate thread for each *execution path* (EP) (i.e., for each distinct leaf-to-root path) in the query plan. Since slaves may take different amounts of time to execute an operator over their local partition of the index, an *asynchronous* exchange of messages for resharding the partial relations at query time makes this step more efficient than a synchronous protocol. As soon as all the shards for the two input relations of a join operator are in place, the threads of the two EPs at each slave are merged into one, and the next join operations can be invoked locally. For a DRJ operator with a graph-reachability predicate, whose source or target variables become bound to constants due to a shared variable, the respective source and target sets for the distributed set-reachability query are initialized from those constants. These are then resharded to the slaves that hold the graph partitions containing the source and target vertices.

Example 5.4. *Consider the example query of Figure 5.3 together with the plan of Figure 5.4. Upon receiving this global query plan from the master, each slaves initializes 4 threads for the 4 EPs in the plan. For our RDF data graph of Figure 5.1 and a partitioning over $k = 2$ slaves, we obtain the following sharded base relations.*

R_1 :

$?p$	$?u$
John_E._Hopcroft	Cornell_University
Richard_Karp	UC_Berkeley
Rajeev_Motwani	Stanford_University

R_2 :

$?p$
John_E._Hopcroft
Richard_Karp

R_3 :

$?p_1$	$?u_1$
John_E._HopCroft	Cornell_University
Richard_Karp	UC_Berkeley
Rajeev_Motwani	Stanford_University

R_{USA} :

Slave 1

R_1 :

R_2 :

R_3 :

R_{USA} : USA

Slave 2

After the iterators to R_1, R_2, R_3 and R_{USA} have been initialized, the 4 threads are merged into 2. The first remaining thread executes the DRJ operator for $Q_1 := \bowtie^{DRJ}_{?p \rightsquigarrow ?p_1} (R_2, R_3)$, which is processed concurrently over the shards of R_2, R_3 located at both slaves. For Q_1, the source and target columns are $?p$ and $?p_1$ respectively, while the property that is transitively queried is workedWith. First, resharding of R_2, R_3 is performed with respect to the values under the source and target columns of the two relations. The respective shards are sent and received by all slaves asynchronously. Based on the constants the slaves obtain for $?p$ and $?p_1$ from the new shards of R_2 and R_3, we then invoke the distributed set-reachability query $Q(S, T)$ over the induced subgraph $G^{\text{workedWith}}$. Finally, the resulting reachable pairs of source and target vertices are used to generate the result tuples for Q_1. In a similar way, and in parallel to the thread executing Q_1, the second remaining thread proceeds with the execution of the DRJ operator for $Q_2 := \bowtie^{DRJ}_{?u \rightsquigarrow USA} (R_1, R_{USA})$ again concurrently over the shards located at both slaves.

Slave1

R^{Q_1} :

$?p$	$?p_1$	$?u_1$
John_E._Hopcroft	John_E._Hopcroft	Cornell_University
John_E._Hopcroft	Richard_Karp	UC_Berkeley
John_E._Hopcroft	Rajeev_Motwani	Stanford_University
⋮	⋮	⋮

R^{Q_2} :

Slave2

R^{Q_1} :

R^{Q_2} :

$?p$	$?u$
John_E._Hopcroft	Cornell_University
Richard_Karp	UC_Berkeley
Rajeev_Motwani	Stanford_University

After the threads for Q_1, Q_2 have both finished generating their intermediate relations R^{Q_1}, R^{Q_2}, the two threads are merged into one final thread at each slave.

$Q_3 := \bowtie_{\{?p, ?u \rightsquigarrow ?u_1\}} (R^{Q_1}, R^{Q_2})$ is processed concurrently over the shards of R^{Q_1}, R^{Q_2}. Unlike Q_1 and Q_2, Q_3 comprises of multiple join conditions which are processed sequentially. Starting with the equi-join on the shared variable $?p$, we obtain another intermediate relation R^{Q_3} from which we subsequently also filter tuples with respect to the graph-reachability predicate $?u \rightsquigarrow ?u_1$.

R^{Q_3} :

$?p$	$?p_1$	$?u$	$?u_1$
John_E._Hopcroft	John_E._Hopcroft	Cornell_University	Cornell_University
John_E._Hopcroft	Richard_Karp	UC_Berkeley	UC_Berkeley
John_E._Hopcroft	Richard_Karp	UC_Berkeley	Stanford_University
⋮	⋮	⋮	⋮

5.6 Evaluation

Here, we provide a detailed evaluation of our approach in processing GGP queries. We implemented our approach in TriAD RDF engine (discussed in Chapter 4). We used GCC-4.7.3 with `-O3` optimization and MPICH2-1.4.1 and Boost-1.55 as external libraries. We ran all of the following experiments on a compute cluster with up to 11 nodes, out of which 1 was dedicated as the master node. Each node runs Debian 7.5, has 48GB of RAM and an Intel E5530@2.40GHz quad core CPU with HT enabled.

5.6.1 Datasets & Benchmark

Datasets. We used three large-scale, both real-world and synthetic, RDF datasets for our evaluation: (i) LUBM-500M[4] (scaled to 500 million triples) is generated using UBA 1.7 in N3 format, (ii) Freebase-500M (with 500 million triples) refers to a subset of a recent Freebase snapshot[5] and (iii) a recent snapshot of DBpedia[6] (with 417,445,957 triples).

Queries. We manually designed three queries for each dataset (L1–L3 for LUBM, F1–F3 for Freebase, D1–D3 for DBpedia) to capture a mixture of reachability queries and relational joins. All SwPP queries are listed in our Appendix A.2.

5.6.2 Efficiency

We first discuss the distributed processing of SwPP queries for the fixed snapshot of the three datasets described above. For TriAD, we used 5 slaves for this setting (plus 1 master node). As competitor, we used the Virtuoso 7.1.0 native RDF store, which is the only available RDF store we are aware of that supports full property-path processing. We remark that the open-source edition of Virtuoso 7.1.0 does not support distribution. We thus compare against a centralized installation of Virtuoso on one of our compute nodes.

[4] http://swat.cse.lehigh.edu/projects/lubm/
[5] https://developers.google.com/freebase/data
[6] http://downloads.dbpedia.org/2015-04/core/

(a) LUBM-500M *(query times in seconds)*					
	#Slaves	**L1**	**L2**	**L3**	**Geo.-Mean**
TriAD	1	6.437	0.331	42.681	4.497
TriAD	5	**1.250**	**0.162**	**8.516**	**1.199**
Virtuoso (cold)	1	10.050	12.624	57.776	19.425
Virtuoso (warm)	1	4.963	5.452	56.603	11.527
(b) Freebase-500M *(query times in seconds)*					
	#Slaves	**F1**	**F2**	**F3**	**Geo.-Mean**
TriAD	1	1.084	1.568	0.677	1.048
TriAD	5	**0.356**	0.642	**0.423**	0.459
Virtuoso (cold)	1	6.590	4.112	13.809	7.206
Virtuoso (warm)	1	1.196	**0.002**	5.601	**0.238**
(c) DBpedia *(query times in seconds)*					
	#Slaves	**D1**	**D2**	**D3**	**Geo.-Mean**
TriAD	1	24.822	0.713	29.407	8.044
TriAD	5	**7.973**	**0.412**	**11.223**	**3.328**
Virtuoso (cold)	1	46.185	19.352	317.899	65.741
Virtuoso (warm)	1	27.820	2.395	302.753	27.222

Table 5.1: Performance evaluation of SwPP queries

A. LUBM-500M. The results for processing SwPP queries (L1, L2, L3) are shown in Table 5.1(a). L1 resembles a single, non-selective reachability join. Processing L1 thus involves an index scan for two input relations and a respective evaluation of the reachability join. We can observe that the centralized version of TriAD performs better than Virtouso in a *cold* cache and comparable to Virtuoso in a *warm* cache setting. We however achieve a significant scale-out for L1 when we evaluate the query on a cluster of 5 slaves. Next, L2 is a selective query with two regular joins and a single reachability join. For this query, we can observe that TriAD achieves a better performance compared to Virtuoso in both the cold and warm cache settings. The non-selective query L3 contains two reachability joins in conjunction with two regular joins. Also here, TriAD continues to perform better than Virtuoso under both a cold and warm cache and further scales out very well in a distributed setting.

B. Freebase-500M. For Freebase, we considered three queries (F1, F2, F3) which we designed along the lines of the L1, L2, L3 LUBM queries. The performance of TriAD for Freebase-500M shows a similar behavior as the one we observed for LUBM-500M. The results are shown in Table 5.1(b). For F3, Virtuoso tends to report different results over repeated runs, which indicates problems with their current support for property paths.

C. DBpedia. We once more considered three queries (D1, D2, D3) consisting of a mixture of relational joins and graph-reachability predicates for DBpedia. The runtime performance of TriAD in comparison with Virtuoso is shown in Table 5.1(c). Also here, TriAD continues to perform very well compared to Virtuoso

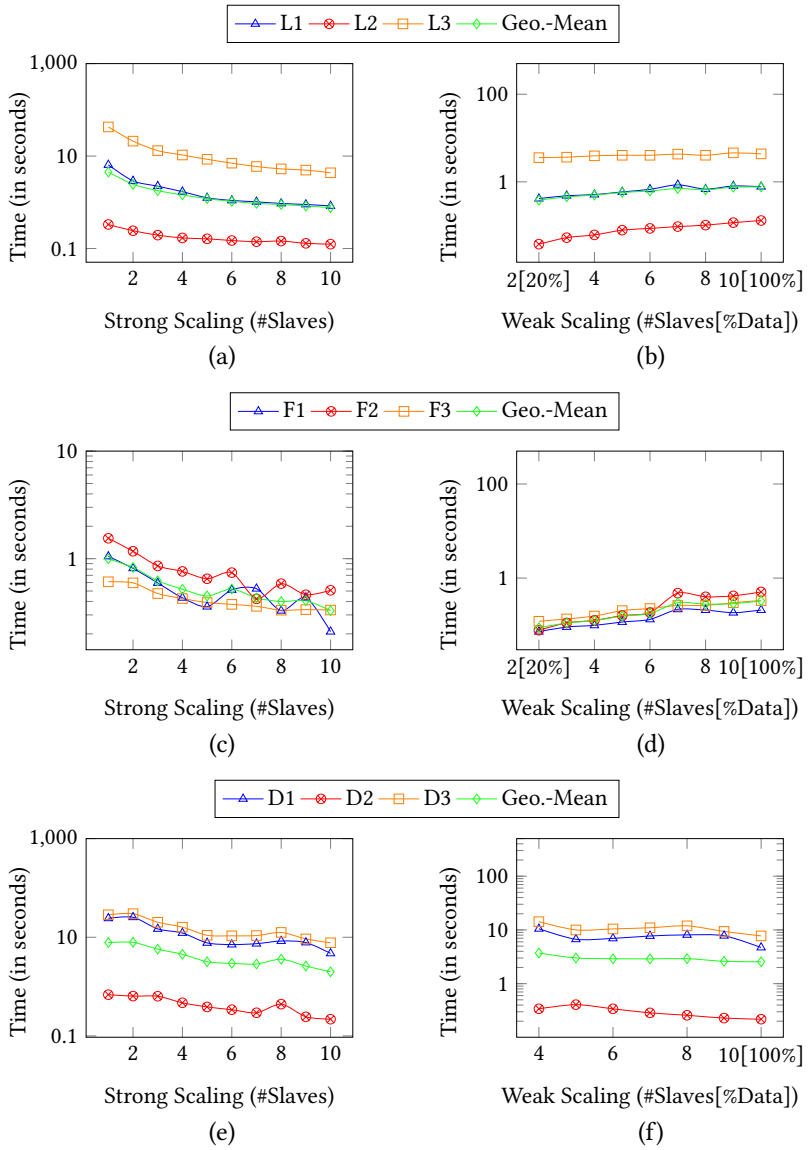


Figure 5.5: Scalability (strong,weak) of SwPP queries for LUBM (a,b), Freebase (c,d) , and DBPedia (e,f) datasets

under both cold and warm cache settings.

5.6.3 Scalability Tests

We finally evaluated the scalability of TriAD for GGP queries by varying the number of slaves from 1 to 10. For this evaluation, we again considered LUBM-500M, Freebase-500M and DBpedia. The results under *strong scaling* are shown in Figure 5.5(a) for LUBM-500M, in Figure 5.5(c) for Freebase-500M, and in Figure 5.5(e) for DBpedia, respectively. It can be observed that the our approach is consistently scalable across multiple datasets. As our last series of runs, we also evaluated the performance of TriAD under *weak scaling*, by increasing the size (from 20%–100%) of the collections as well as the number of slaves (from 2–10) in equal proportions. The results are shown in Figure 5.5(d)–(f), and depict the textbook results for weak scalability, albeit with a slight upward trend for Freebase dataset as the result size increased with higher percentage of data.

5.7 Summary

In this chapter, we presented a distributed solution and an extension to TriAD for processing GGP queries in an efficient and scalable manner. Relying on the combination of existing BGP querying framework with the adapted set reachability algorithms, TriAD, which to our knowledge is the currently fastest, distributed engine that explicitly tackles the processing of generalized patterns in GGP queries. Specifically, in TriAD we augment the index-based set-reachability solution to implement a new relational query operator to tackle the kind of generalized graph-pattern queries. Our evaluation over both real-world and synthetic RDF collections confirm that TriAD achieves very significant gains compared to the only currently available, native RDF store that supports SPARQL 1.1 with property paths.

Chapter **6**

Conclusions and Future Directions

6.1 Conclusions

This thesis addressed an important and challenging task of efficient and scalable querying of labeled graphs. Specifically we focused on three query models: *set reachability, basic graph patterns, generalized graph patterns* that are vital in many graph applications. To this end, we presented a distributed architecture called "TriAD" that adopts both graph-based and relational models to holistically process the three query models.

To process set reachability queries in TriAD, we developed a distributed solution based on a graph model. By precomputing and materializng the reachability among the boundary vertices and indexing them along with local graphs, our solution requires only a single round of communication to process any set reachability query irrespective of the size, partitioning, and topology of the graph. On the other hand, to process BGP queries that belong to pattern matching class, we resorted to a relational model. A multi-pattern BGP query is thus executed as a series of relational joins. To facilitate efficient join executions, we adopted an asynchronous communication protocol in TriAD, and proposed several techniques such as multi-threaded and asynchronous execution framework, join-ahead pruning via graph summarization, etc. Finally, to process GGP queries, which combine the BGP and set reachability query model, we proposed an approach based on the duxality of graph-based and relational model. To this end, we proposed a bimodal indexing layout that integrates the set reachability and triple indexes. Further, we developed a unified cost-based query optimizer that generates an efficient query plan interleaving set reachability and relation join operations, thereby, processing the GGP queries in an efficient and scalable manner.

We empirically evaluated TriAD in comparison to multiple state-of-the-art systems over several real-world and synthetic datasets. Our evaluation demostrated the superior efficiency of TriAD over its competitors. In conclusion, TriAD was able to achieve success by adopting different strategies for processing set reachability and BGP queries, and integrating them when needed for processing GGP queries.

6.2 Future Directions

Although, we addressed some of the key challenges in efficient and scalable querying of labeled graphs. There are many future possiblities which can be extended with our work. We below list some of them with respect to the considered query models in our work.

- In solving set reachability queries, we proposed a framework where a query can be processed using a single round of communication. This approach benefits in acheiving efficiency over iterative-based techniques and providing real-time processing of queries. We identified two interesting future works that could leverage this framework: i) supporting set reachability queries with length restriction, and ii) finding top-k shortest reachable pairs. Both the problems have many practical applications where real-time processing is a necessity.
- In our second query model, we considered only conjunctive BGP queries. An immediate next direction would be to tackle BGP queries with other algebraic operators such as disjunction, negation, difference, etc.
- We considered a subset of SPARQL 1.1, i.e., SwPP, language specification as representation language for our GGP query model. This restricts the grammar of our generalized patterns to be simple. An interesting and challenging future work would be to extend this grammar to support arbitrary complex regular expressions. However, this requires developing solutions first to process set reachability with regular expressions constraints.

Bibliography

Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2009). SW-Store: A vertically partitioned DBMS for semantic web data management. *VLDB Journal*, 18(2):385–406.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88.

Agrawal, R., Borgida, A., and Jagadish, H. V. (1989). Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, volume 18, pages 253–262.

Aittokallio, T. and Schwikowski, B. (2006). Graph-based methods for analysing networks in cell biology. *Briefings in Bioinformatics*, 7(3):243–255.

Aleman-Meza, B., Hakimpour, F., Budak Arpinar, I., and Sheth, A. P. (2007). SwetoDblp ontology of Computer Science publications. *Web Semantics*, 5(3):151–155.

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The Object-Oriented Database System Manifesto. *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240.

Atre, M., Chaoji, V., Zaki, M., and Hendler, J. (2010). Matrix Bit loaded: a scalable lightweight join query processor for RDF data. *Proceedings of the 19th international conference on World wide web*, pages 41–50.

Bai, Y., Wang, C., Ning, Y., Wu, H., and Wang, H. (2013). G-path: Flexible path pattern query on large graphs. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 333–336, New York, NY, USA. ACM.

Beckett, D., Berners-Lee, T., Prud'hommeaux, E., and Carothers, G. (2014). RDF 1.1 Turtle.

Belleau, F., Nolin, M. A., Tourigny, N., Rigault, P., and Morissette, J. (2008). Bio2RDF: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics*, 41(5):706–716.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009). DBpedia - A crystallization point for the Web of Data. *Journal of Web Semantics*, 7(3):154–165.

Blackman, K. R. (1998). Technical note: IMS celebrates thirty years as an IBM product. *IBM Systems Journal*, 37(4):596–603.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10008(10):6.

Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. (2008). Freebase. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 1247.

Bornea, M. a., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., and Bhattacharjee, B. (2013). Building an efficient RDF store over a relational database. *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, page 121.

Bray, T., Paoli, J., Maler, E., and Microsystems, S. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation*, 0:1–37.

Broekstra, J., Kampman, A., and Harmelen, F. V. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. *International Semantic Web Conference ISWC*, 1:54–68.

Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. (2013). Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA. USENIX.

Buluc, A., Meyerhenke, H., Safro, I., Sanders, P., and Schulz, C. (2013). Recent Advances in Graph Partitioning. *arXiv*, pages 1–36.

Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. (1996). A query language and optimization techniques for unstructured data. *ACM SIGMOD Record*, 25:505–516.

Buneman, P., Fernandez, M., and Suciu, D. (2000). UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76.

Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. Y. (2000). Containment of Conjunctive Regular Path Queries with Inverse. *Proc.\ of the 7th Int.\ Conf.\ on the Principles of Knowledge Representation and Reasoning (KR˜2000)*, pages 176–185.

Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. Y. (2002). Rewriting of Regular Expressions and Regular Path Queries. *Journal of Computer and System Sciences*, 64:443–465.

Calvanese, D., Giacomo, G. D., and Lenzerini, M. (2003). Reasoning on regular path queries. *Acm Sigmod*, 32(4):83–92.

Carothers, G. (2014). RDF 1.1 N-Quads - A line-based syntax for an RDF datasets.

Chandra, A. K. and Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 77–90, New York, NY, USA. ACM.

Chen, Y. and Chen, Y. (2008). An efficient algorithm for answering graph reachability queries. In *Proceedings - International Conference on Data Engineering*, pages 893–902.

Cheng, J., Yu, J. X., and Ding, B. (2007). Cost-based query optimization for multi reachability joins. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications*, DASFAA'07, pages 18–30, Berlin, Heidelberg. Springer-Verlag.

Cheng, J., Yu, J. X., Lin, X., Wang, H., and Yu, P. S. (2006). Fast computation of reachability labeling for large graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3896 LNCS, pages 961–979.

Codd, E. F. (1983). A relational model of data for large shared data banks. *Commun. ACM*, 26(6):64–69.

Cohen, E., Halperin, E., Kaplan, H., and Zwick, U. (2003). Reachability and Distance Queries via 2-Hop Labels. *SIAM Journal on Computing*, 32:1338–1355.

Consens, M. P. and Mendelzon, A. O. (1989). Expressing structural hypertext queries in graphlog. In *Proceedings of the Second Annual ACM Conference on Hypertext*, HYPERTEXT '89, pages 269–292, New York, NY, USA. ACM.

Consens, M. P. and Mendelzon, A. O. (1990). Graphlog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 404–416, New York, NY, USA. ACM.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, third edition.

Cruz, I. F., Mendelzon, A. O., and Wood, P. T. (1988). G+: recursive queries without recursion. In *Expert Database Conf.*, pages 645–666.

Cudré-Mauroux, P., Enchev, I., Fundatureanu, S., Groth, P., Haque, A., Harth, A., Keppmann, F. L., Miranker, D., Sequeda, J. F., and Wylot, M. (2013). NoSQL databases for RDF: An empirical evaluation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8219 LNCS, pages 310–325.

Date, C. J. and Darwen, H. (1997). *A Guide to the SQL Standard (4th Ed.): A User's Guide to the Standard Database Language SQL*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Demetrescu, C. and Italiano, G. F. (2006). Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms*, 4(3):353–383.

DeRose, S. J., Maler, E., Orchad, D., and Walsh, N. (2010). XML Linking Language (XLink). *W3C Recommendation*, 23(May).

Deutsch, A. and Tannen, V. (2002). *Optimization Properties for Classes of Conjunctive Regular Path Queries*, pages 21–39. Springer Berlin Heidelberg, Berlin, Heidelberg.

Diestel, R. (2012). *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J. (2010). Hadoop++. *Proceedings of the VLDB Endowment*, 3(1-2):515–529.

Eckman, B. A. and Brown, P. G. (2006). Graph data management for molecular and cell biology. *IBM Journal of Research and Development*, 50:545—-.

EMBL, SIB Swiss Institute of Bioinformatics, and Protein Information Resource (PIR) (2013). UniProt. In *Nucleic acids research*, pages 41: D43–D47.

Erling, O. and Mikhailov, I. (2010). *Virtuoso: RDF Support in a Native RDBMS*, pages 501–519. Springer Berlin Heidelberg, Berlin, Heidelberg.

Fan, W., Li, J., Luo, J., Tan, Z., Wang, X., and Wu, Y. (2011). Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 925–936, New York, NY, USA. ACM.

Fan, W., Wang, X., and Wu, Y. (2012). Performance Guarantees for Distributed Reachability Queries. *Proceedings of the VLDB Endowment*, 5(11):1304–1315.

Fan, W., Wang, X., and Wu, Y. (2014a). Answering graph pattern queries using views. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 184–195.

Fan, W., Wang, X., Wu, Y., and Deng, D. (2014b). Distributed graph simulation: Impossibility and possibility. *Proceedings of the VLDB Endowment*, 7(12):1083–1094.

Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. (2012). SAP HANA Database - Data Management for Modern Business Applications. *ACM Sigmod Record*, 40(4):45–51.

Fernández, M., Florescu, D., Kang, J., Levy, A., and Suciu, D. (1998). Catching the boat with strudel: Experiences with a web-site management system. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 414–425, New York, NY, USA. ACM.

Florescu, D., Levy, A., and Suciu, D. (1998). Query containment for conjunctive queries with regular expressions. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '98*, pages 139–148.

Gao, S. and Anyanwu, K. (2013). Prefixsolve: Efficiently solving multi-source multi-destination path queries on rdf graphs by sharing suffix computations. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 423–434, New York, NY, USA. ACM.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2008). Database Systems: The Complete Book. *Education*, page 1248.

Gonzalez, J., Low, Y., and Gu, H. (2012). Powergraph: Distributed graph-parallel computation on natural graphs. *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 17–30.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). GraphX : Graph Processing in a Distributed Dataflow Framework. *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 599–613.

Grün, C. (2011). Basex. the xml database.

Gubichev, A., Bedathur, S., Seufert, S., and Weikum, G. (2010). Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 499–508, New York, NY, USA. ACM.

Gubichev, A., Bedathur, S. J., and Seufert, S. (2013). Sparqling kleene: fast property paths in RDF-3X. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, page 14.

Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. (2014a). TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. *ACM SIGMOD International Conference on Management of Data (SIGMOD 2014)*, pages 289–300.

Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. (2014b). Using graph summarization for join-ahead pruning in a distributed RDF engine. In *Proceedings of the Sixth Workshop on Semantic Web Information Management, SWIM 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 41:1–41:4.

Gurajada, S. and Theobald, M. (2016a). Distributed processing of generalized graph-pattern queries in SPARQL 1.1. *CoRR*, abs/1609.05293.

Gurajada, S. and Theobald, M. (2016b). Distributed set reachability. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1247–1261, New York, NY, USA. ACM.

Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., and Sahli, M. (2016). Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal*, 25(3):355–380.

Harris, S. and Gibbins, N. (2003). 3store: Efficient Bulk RDF Storage. *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, pages 1–20.

Harris, S., Lamb, N., and Shadbolt, N. (2009). 4store: The design and implementation of a clustered RDF store. In *CEUR Workshop Proceedings*, volume 517, pages 94–109.

Hassanzadeh, O. and Consens, M. (2009). Linked movie data base. In *CEUR Workshop Proceedings*, volume 538.

Hayes, P. (2004). RDF Semantics. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

Huang, J., Abadi, D. J., and Ren, K. (2011). Scalable SPARQL Querying of Large RDF Graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134.

Jagadish, H. V. (1990). A compression technique to materialize transitive closure. *TODS*, 15(4):558–598.

Jena (2007). Jena Semantic Web Framework. http://jena.sourceforge.net/.

Jin, R., Ruan, N., Xiang, Y., and Wang, H. (2011). Path-Tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs. *ACM Transactions on Database Systems*, 36(1):1–44.

Jin, R., Xiang, Y., Ruan, N., and Fuhry, D. (2009). 3-HOP: a high-compression indexing scheme for reachability query. *Proceedings of the 35th SIGMOD Conference*, pages 813–826.

Jin, R., Xiang, Y., Ruan, N., and Wang, H. (2008). Efficiently answering reachability queries on very large directed graphs. *SIGMOD*, pages 595–607.

Karypis, G. and Kumar, V. (1998). A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392.

Kelley, B. P., Yuan, B., Lewitter, F., Sharan, R., Stockwell, B. R., and Ideker, T. (2004). PathBLAST: A tool for alignment of protein interaction networks. *Nucleic Acids Research*, 32(WEB SERVER ISS.).

Khan, A., Li, N., Yan, X., Guan, Z., Chakraborty, S., and Tao, S. (2011). Neighborhood based fast graph search in large networks. *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*, page 901.

Klyne, G. and Carroll, J. J. (2004). Resource Description Framework (RDF): Concepts and Abstract Syntax. *W3C Recommendation*, 10:1—-20.

Kyrola, A., Blelloch, G., and Guestrin, C. (2012). GraphChi: Large-Scale Graph Computation on Just a PC Disk-based Graph Computation. *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 31–46.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

Liang, Z., Xu, M., Teng, M., and Niu, L. (2006). NetAlign: a web-based tool for comparison of protein interaction networks. *Bioinformatics (Oxford, England)*, 22(17):2175–7.

Libkin, L., Martens, W., and Vrgoč, D. (2013). Querying graph databases with XPath. *Proceedings of the 16th International Conference on Database Theory - ICDT '13*, page 129.

Lin, J. and Dyer, C. (2010). Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. (2012). Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2010). GraphLab: A New Framework for Parallel Machine Learning. *The 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, pages 8–11.

Maier, D., Stein, J., Otis, A., and Purdy, A. (1986). Development of an object-oriented dbms. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 472–482, New York, NY, USA. ACM.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 135.

Martella, C., Shaposhnik, R., and Logothetis, D. (2015). *Practical graph analytics with apache giraph*. Apress.

Martín, M. S., Gutierrez, C., and Wood, P. T. (2011). SNQL: A Social Network query and transformation language. In *CEUR Workshop Proceedings*, volume 749.

Mendelzon, A. O. and Wood, P. T. (1995). Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing*, 24(6):1235–1258.

Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. (2002). Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827.

Milo, T. and Suciu, D. (1999). Index Structures for Path Expressions. *Proceedings of the 7th International Conference on Database Theory*, pages:277–295.

Mongiovì, M., Di Natale, R., Giugno, R., Pulvirenti, A., Ferro, A., and Sharan, R. (2010). SIGMA: a set-cover-based inexact graph matching algorithm. *Journal of bioinformatics and computational biology*, 8(2):199–218.

MPI, Lusk, E., Huss, S., Saphir, B., and Snir, M. (2009). MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):623.

Natarajan, M. (2000). Understanding the Structure of a Drug Trafficking Organization: a Conversational Analysis. *Crime Prevention Studies*, 11:273–298.

Neo4j (2012). Neo4j: World's Leading Graph Database. http://neo4j.org/.

Neumann, T. and Weikum, G. (2008). RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1:647–659.

Neumann, T. and Weikum, G. (2009). Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 627–640, New York, NY, USA. ACM.

Neumann, T. and Weikum, G. (2010a). The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113.

Neumann, T. and Weikum, G. (2010b). x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *Proceedings of the VLDB Endowment*, 3(1):256–263.

Oracle (2006). Anatomy of an XML Database : Oracle Berkeley DB XML. *An Oracle White Paper*, (September):1 – 16.

Orlin, J. (1977). Contentment in graph theory: Covering graphs with cliques. *Indagationes Mathematicae (Proceedings)*, 80(5):406–424.

Park, D. M. R. (1981). Concurrency and automata on infinite sequences. In Deussen, P., editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer.

Peng, P., Zou, L., Özsu, M. T., Chen, L., and Zhao, D. (2016). Processing SPARQL queries over distributed RDF graphs. *The VLDB Journal*, 25(2):243–268.

Picalausa, F., Luo, Y., Fletcher, G. H. L., Hidders, J., and Vansummeren, S. (2012). A structural approach to indexing triples. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7295 LNCS, pages 406–421.

Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haridasan, M. (2012). Managing Large Graphs on Multi-cores with Graph Awareness. *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, page 4.

Prud'hommeaux, E., Harris, S., and Seaborne, A. (2013). SPARQL 1.1 Query Language. Technical report, W3C.

Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. *W3C Recommendation*, 2009(January):1–106.

Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., and Lausen, G. (2012). RDFPath: Path query processing on large RDF graphs with MapReduce. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7117 LNCS, pages 50–64.

Qian, R. (2013). Understand Your World with Bing. http://blogs.bing.com/search/2013/03/21/understand-your-world-with-bing/.

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*, volume 8. McGraw-Hill, Inc.

Roditty, L. and Zwick, U. (2008). Improved dynamic reachability algorithms for directed graphs. *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, 37(5):1455–1471.

Rodriguez, M. a. (2015). The Gremlin Graph Traversal Machine and Language. *Proc. 15th Symposium on Database Programming Languages*, pages 1–10.

Rohloff, K. and Schantz, R. E. (2011). Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store. *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing*, pages 35–44.

Ronen, R. and Shmueli, O. (2009). SoQL: A language for querying and creating data in social networks. In *Proceedings - International Conference on Data Engineering*, pages 1595–1602.

Sakr, S. and Al-Naymat, G. (2010). Relational processing of RDF queries. *ACM SIGMOD Record*, 38(4):23.

Sangiorgi, D. (2009). On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems*, 31(4):1–41.

Sarma, A. D., Gollapudi, S., and Panigrahy, R. (2010). A Sketch-Based Distance Oracle for Web-Scale Graphs. *Wsdm*, pages 401–410.

Sarwat, M., Elnikety, S., He, Y., and Kliot, G. (2012). Horton: Online query execution engine for large distributed graphs. In *Proceedings - International Conference on Data Engineering*, pages 1289–1292.

Sarwat, M., Elnikety, S., He, Y., and Mokbel, M. F. (2013). Horton+. *Proceedings of the VLDB Endowment*, 6(14):1918–1929.

Schaik, S. J. V. and Moor, O. D. (2011). A Memory Efficient Reachability Data Structure Through Bit Vector Compression. *SIGMOD*, pages 913–924.

Seufert, S., Anand, A., Bedathur, S., and Weikum, G. (2013). FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *Proceedings - International Conference on Data Engineering*, pages 1009–1020.

Shang, Z. and Yu, J. X. (2013). Catch the wind: Graph workload balancing on cloud. In *Proceedings - International Conference on Data Engineering*, pages 553–564.

Shao, B., Wang, H., and Li, Y. (2013). Trinity- A Distributed Graph Engine on a Memory Cloud. *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, page 505.

Sidirourgos, L., Goncalves, R., Kersten, M. L., Nes, N. J., and Manegold, S. (2008). Column-Store Support for RDF Data Management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(212):1553–1563.

Singhal, A. (2012). Official Google Blog: Introducing the Knowledge Graph: things, not strings. https://googleblog.blogspot.co.za/2012/05/introducing-knowledge-graph-things-not.html.

Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA. ACM.

Sugar, C. and Gareth, J. (2003). Finding the number of clusters in a data set : An information theoretic approach. *Journal of the American Statistical Association*, 98:750–763.

Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., and Xie, G. (2015). SQLGraph: An Efficient Relational-Based Property Graph Store. *SIGMOD*, pages 1887–1901.

The MPI Forum (1993). MPI : A Message Passing Interface. In *Proceedings of the Conference on High Performance Networking and Computing*, pages 878–883.

The UniProt Consortium (2014). UniProt: a hub for protein information. *Nucleic Acids Research*, 43(Database issue):D204–12.

Then, M., Kaufmann, M., Chirigati, F., Hoang-Vu, T., Pham, K., Kemper, A., Neumann, T., and Vo, H. T. (2014). The more the merrier: Efficient multi-source graph traversal. *PVLDB*, 8(4):449–460.

Tian, Y., Balmin, A., and Corsten, S. (2013). From "think like a vertex" to "think like a graph". *Proceedings of the VLDB Endowment*, 7:193–204.

Tian, Y., McEachin, R. C., Santos, C., States, D. J., and Patel, J. M. (2007). SAGA: A subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239.

Tian, Y. and Patel, J. M. (2008). TALE: A tool for approximate large graph matching. In *Proceedings - International Conference on Data Engineering*, pages 963–972.

Tong, H., Gallagher, B., Faloutsos, C., and Eliassi-Rad, T. (2007). Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 737.

Trißl, S. and Leser, U. (2007). Fast and Practical Indexing and Querying of Very Large Graphs. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data SIGMOD 07*, pages 845–856.

Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., and Boncz, P. (2012). Heuristics-based query optimisation for SPARQL. *Proceedings of the 15th International Conference on Extending Database Technology - EDBT '12*, page 324.

Urbani, J., Dutta, S., Gurajada, S., and Weikum, G. (2016). KOGNAC: efficient encoding of large knowledge graphs. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 3896–3902.

Vaglini, G. (1991). Communication and concurrency. *Information and Software Technology*, 33(6):462.

Vardi, M. Y. (1982). The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 137–146, New York, NY, USA. ACM.

Veloso, R. R., Cerf, L., Jr., W. M., and Zaki, M. J. (2014). Reachability queries in very large graphs: A fast refined online search approach. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 511–522.

Wang, H., He, H., Yang, J., Yu, P. S., and Yu, J. X. (2006). Dual labeling: Answerng graph reachability queries in constant time. In *Proceedings - International Conference on Data Engineering*, volume 2006, page 75.

Weiss, C. U. O. Z., Weiss, C., Karras, P. N. U. o. S., Bernstein, A. U. o. Z., Karras, P., and Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment archive*, 1(1):1008–1019.

Wood, P. T. (2012). Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., Stoica, I., and AMPLab, E. (2013). GraphX: A Resilient Distributed Graph System on Spark. *First International Workshop on Graph Data Management Experiences and Systems*, page 2.

Yan, D., Cheng, J., Lu, Y., and Ng, W. (2014). Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992.

Yildirim, H., Chaoji, V., and Zaki, M. J. (2010). GRAIL: scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284.

Yu, J. X. and Cheng, J. (2010). *Graph Reachability Queries: A Survey*, pages 181–215. Springer US, Boston, MA.

Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., and Liu, L. (2013). TripleBit: A Fast and Compact System for Large Scale RDF Data. *Proc. VLDB Endow.*, 6(7):517–528.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark : Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10.

Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. (2013). A distributed graph engine for web scale RDF data. *Proceedings of the 39th international conference on Very Large Data Bases*, pages 265–276.

Zhang, X., Chen, L., Tong, Y., and Wang, M. (2013). EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Proceedings - International Conference on Data Engineering*, pages 565–576.

Zou, L., Mo, J., Chen, L., Özsu, M., and Zhao, D. (2011). gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB*, 4(8):482–493.

Appendices

Chapter A

Additional Details

A.1 Giraph Implementations of DSR Queries

A.1.1 Giraph

The following program code illustrates our implementation of DSR queries in Giraph. In superstep 0, all source vertices are first added to a `newSources` array. Thus, the function `isSource(.)` returns true if the vertex is a source. In the subsequent supersteps, `newSources` represents the additional sources from which the current vertex `v` is reachable. If `newSources` is not empty, then we iteratively propagate these sources to all neighbors of vertex `v`.

Distributed Set Reachability in Giraph

```
public void compute(Vertex v, Iterable m){
  ArrayList<Integer> newSources = new ArrayList<Integer>();
  if(getSuperStep() == 0){
     if(isSource(v))
       newSources.add(v.getId().get());
     v.getValue().clearSources()
  }else
    for(IntWritable msg : m)
      newSources.add(m.get());

  newSources.removeAll(v.getValue().getSources());
  if(newSources.size() > 0){
      v.addSources(newSources);
       for(Edge<IntWritable, NullWritable> e : v.getEdges()){
         IntWritable nb = e.getTargetVertexId();
         for(int src: newSources)
           sendMessage(nb,new IntWritable(src));   } } }
}
```

A.1.2 Giraph++

Unlike in Graph, the Giraph++ API exposes the underlying partitioning information along with each call of the `compute(.)` function. The code for DSR processing is similar to Giraph, except that the vertices that are local to the current source vertices are directly updated using a centralized local reachability computation via `localProcess(.)`. After the local processing, for each vertex we again communicate its reachable list of vertices to the remote neighbors.

Distributed Set Reachability in Giraph++

```
public void compute(Partition p){
    ArrayList<Integer> q_sources = new ArrayList<Integer>();
    ArrayList<Integer> newSources = new ArrayList<Integer>();
    if(getSuperStep() == 0){
       if(isSource(v))
         sources.add(v.getId().get();
    }else{
      MessageStore<IntWritable, IntWritable> mstore
               = getCurrentMessageStore();
      for(Vertex v : p.getVertices()){
        if(mstore.hasMessagesForVertex(v.getId())){
           newSources.clear();
           for(IntWritable message :
              mstore.getVertexMessages(v.getId()))
              newSources.add(message.get());
           newSources.removeAll(v.getValue().getSources());
           if(newSources.size() > 0){
              q_sources.add(v.getId());
              v.getValue().addNewSources(newSources);
           } } }
    }
    localProcess(p,q_sources);
    for(Vertex v : p.getVertices()){
      if(v.getValue().getNewSources().size() > 0){
         for(Edge<IntWritable, NullWritable> edge
                        : v.getEdges()){
            int nb = edge.getTargetVertexId().get();
            if(!p.contains(nb)
               for(int src : v.getValue().getNewSources())
                 sendMessage(nb,new IntWritable(src));
         }
         v.getValue().addSources(v.getValue().getNewSources());
         v.getValue().getNewSources().clear();}
   }
}
```

A.1.3 Giraph++wEq

The following code depicts our DSR implementation in Giraph++wEq, including our proposed equivalence-sets optimization. We first compute equivalence sets in our DSR system and prepare an adjacency graph as input to Giraph. For each vertex v in the input graph, in addition to its adjacent neighbors, we also add their equivalence sets (our in-virtual vertices) as counterparts. This graph is loaded into Giraph using a custom input reader. The below code shows the DSR computation. The implementation shares a major part of the code with the Giraph++implementation, where the only difference lies in the communication of the reachable sets of vertices in each superstep. After the local processing, we iterate over each vertex and send its reachable list of sources to only the in-virtual vertices instead of all neighbors.

Distributed Set Reachability in Giraph++wEq

```
public void compute(Partition p){
    ArrayList<Integer> q_sources = new ArrayList<Integer>();
    ArrayList<Integer> newSources = new ArrayList<Integer>();
    if(getSuperStep() == 0){
       if(isSource(v))
         sources.add(v.getId().get();
    }else{
      MessageStore<IntWritable, IntWritable> mstore
               = getCurrentMessageStore();
      for(Vertex v : p.getVertices()){
        if(mstore.hasMessagesForVertex(v.getId())){
           newSources.clear();
```

```
            for(IntWritable message :
               mstore.getVertexMessages(v.getId()))
               newSources.add(message.get());
            newSources.removeAll(v.getValue().getSources());
            if(newSources.size() > 0){
               q_sources.add(v.getId());
               v.getValue().addNewSources(newSources);
            } } }
    }
    localProcess(p,q_sources);
    for(Vertex v : p.getVertices()){
      if(v.getValue().getNewSources().size() > 0){
         for(int eq_nb : v.getEqList())
            for(int src : v.getValue().getNewSources())
               sendMessage(new IntWritable(eq_nb),new IntWritable(src));
         v.getValue().addSources(v.getValue().getNewSources());
         v.getValue().getNewSources().clear();}
    }
}
```

A.2 SPARQL Queries with Property Paths

A. LUBM Queries

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix ub:  <http://www.lehigh.edu/∼zhp2/2004/0401/univ-bench.owl#>
```

L1: SELECT $*$
WHERE {?x rdf:type ub:ResearchGroup.
?x ub:subOrganizationOf* ?y.
?y rdf:type ub:University. }

L2: SELECT $*$
WHERE {?x rdf:type ub:FullProfessor.
?x ub:headOf ?d.
?d ub:subOrganizationOf* ?y.
?y rdf:type ub:University. }

L3: SELECT $*$
WHERE {?$r1$ rdf:type ub:ResearchGroup.
?$r1$ ub:subOrganizationOf* ?y.
?y rdf:type ub:University. }
?$r2$ rdf:type ub:ResearchGroup.
?$r2$ ub:subOrganizationOf* ?y.

B. Freebase Queries

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix fb:  <http://rdf.freebase.com/ns>
```

F1: SELECT ∗
WHERE {?*p* fb:people.person.place_of_birth ?*city* .
?*city* fb:location.location.containedby* ?*state*.
?*country* fb:location.location.contains ?*state*.

F2: SELECT ∗
WHERE {?*p* fb:people.person.place_of_birth ?*city* .
?*city* fb:location.location.containedby* ?*state*.
?*country* fb:location.location.contains ?*state*.
?*p* fb:award.award _winner.awards_won ?*prize*.
?*p* rdf:type fb:government.us _president.

F3: SELECT ∗
WHERE {?*p* fb:award.award _winner.awards_won ?*prize*.
?*prize* rdf:type* ?*z*
?*z* fb:award.award_honor.ceremony ?*c*
?*p* fb:people.person.sibling_s* ?*p*1
?*p* rdf:type fb:government.us _president.
?*p*1 fb:award.award_winner.awards_won ?*prize*

C. DBpedia Queries

Namespace prefixes available from: http://de.dbpedia.org/sparql?nsdecl

D1: SELECT ∗
WHERE {?*s*1 rdf:type ?*s*.
?*s* rdfs:subClassOf* ?*o*.
?*o* owl:equivalentClass yago-res:wordnet_medium_106254669

D2: SELECT ∗
WHERE {?*s* foaf:isPrimaryTopicOf wiki:North_Auburn,_California .
?*s* dbpedia-owl:isPartOf* ?*c*
?*x* dbpedia-owl:hometown ?*c*.
?*x* foaf:isPrimaryTopicOf ?*r*.

D3: SELECT ∗
WHERE {?*s* dbpprop:leaderTitle ?*title*.
?*title* rdf:type ?*class*.
?*class* rdfs:subClassOf* ?*class*2.
?*class*2 owl:equivalentClass yago-res:wordnet_abstraction_100002137 .
?*s* dbpedia-owl:isPartOf* ?*c*.
?*x* dbpedia-owl:hometown ?*c*.
?*x* foaf:isPrimaryTopicOf ?*r*.

List of Figures

List of Tables

List of Algorithms