ABSTRACT

HONG, SEOKYONG. Graph Analytics on Modern Graph Processing Systems. (Under the direction of Raju R. Vatsavai.)

Popularity of graph databases has significantly increased in recent years. With increasing volume and applications of graphs, computational graph analytics has been widely employed for understanding and exploring real-world problems formulated as graphs. In graph analytics, fundamental tasks include graph pattern matching and graph mining. Graph pattern matching provides a way to retrieve interesting subgraphs that match to graph patterns and conditions given by users. Graph mining aims at deriving hidden properties and knowledge embedded in graph databases by using various operations. While these two tasks can be independently performed on graphs, they are frequently used together for solving complex graph problems.

Due to increased popularity of graph databases and maturity in graph analytics, recent years have also witnessed proliferation of several commercial and open source systems for data scientists to accomplish their desired analytical tasks. However, due to lack of standards, many of these systems provide features that are limited in functionality, scalability, and usability, making it difficult for users to choose appropriate system and functionality for building a given application.

In graph pattern matching, recent systems, most commonly, allow users to describe graph patterns in a query/programming interface, which are then optimized and processed by an underlying graph engine. The impact of diversity observed in their interfaces, computational models, and level of optimization has not been well evaluated across different data models in the literature. This makes it complicated for data scientists to understand their advantages and limitations and, in turn, limits the ability of users in choosing a suitable system that meets their expectations.

Another limitation of current graph database systems is that they are hand-optimized for built-in graph mining operations. Though such optimized operations are important for graph analysis, current systems are limited by number of such operations and these optimizations vary from system to system. For example, RDF-based graph systems do not natively support graph mining operations. As a result, graphs are frequently converted and loaded into different systems to apply various graph mining algorithms, which degrade the efficiency of graph analysis.

In order to understand the ecosystem of graph databases and their functionalities, in this thesis, we first evaluated six recent graph processing systems based on two modern graph data models (RDF and property graph) with graph pattern matching workloads. By benchmarking major graph database systems, we discuss their advantages and limitations in both quantitative and qualitative ways. Second, we implemented five important graph mining operations and
optimized them in RDF-based systems to augment the functionality of these platforms. Third, we focused on graph clustering analysis and implemented two density-based clustering operations, one that considers non-structural attributes of graph entities represented as RDF, and the other that considers structural properties of graph entities represented as matrix. Finally, we conclude our thesis with a few possible research directions.
Graph Analytics on Modern Graph Processing Systems

by
Seokyong Hong

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2018

APPROVED BY:

_________________________  _______________________
Munindar P. Singh         Rada Y. Chirkova

_________________________  _______________________
Min Chi                  Raju R. Vatsavai
Chair of Advisory Committee
DEDICATION

I dedicate this dissertation work to my parents who raised me with love and devotion and have been a constant source of support and encouragement during the challenges of life.
BIOGRAPHY

Seokyong Hong was born and grew up in Seoul, South Korea. He attended Hansung University in Seoul, South Korea and graduated with a B.A. in Computer Engineering. He served in the Republic of Korea Air Force (ROKAF) from 1999 to 2002. Then, he started his graduate research in the same university in 2003 and received his Master of Engineering degree in 2005. He joined North Carolina State University in 2008 and receives his Ph.D. of computer science in 2018.
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Ranga R. Vatsavai who gave me an opportunity to pursue my Ph.D. study under his supervision and assistance. I also would like to thank my master’s program advisor, Young-Hwan Park, for encouraging me in my choice of career and supporting me in his life and in heaven.
TABLE OF CONTENTS

List of Tables .................................................................................. vii
List of Figures .................................................................................. viii
Listings .............................................................................................. ix

Chapter 1 Introduction ..................................................................... 1
  1.1 Graph Analytics on Modern Graph Processing Systems .......... 1
  1.2 Contribution .......................................................................... 2
  1.3 Outline ............................................................................... 4

Chapter 2 Modern Graph Analytics ................................................. 5
  2.1 Graph and Graph Data Models ............................................. 5
  2.2 Graph Analytics .................................................................. 8
  2.3 Modern Graph Processing Systems ..................................... 9

Chapter 3 Benchmark Graph Analysis Systems with Graph Pattern Matching Workloads .................................................. 11
  3.1 Benchmarked Graph Analysis Systems ................................ 11
  3.2 Benchmark Suite Design ..................................................... 12
    3.2.1 LUBM Benchmark Suite ............................................ 12
    3.2.2 Extension for Data Generation .................................... 13
    3.2.3 Query Translation ....................................................... 15
  3.3 Benchmark Results ............................................................. 18
    3.3.1 Evaluation Environments ........................................... 19
    3.3.2 Quantitative Evaluation ............................................... 19
    3.3.3 Qualitative Evaluation ............................................... 25
  3.4 Conclusion ......................................................................... 28

Chapter 4 Graph Mining Operations for RDF Graphs ..................... 30
  4.1 Introduction ....................................................................... 30
  4.2 Background ........................................................................ 31
    4.2.1 Graph Mining Operations ......................................... 31
    4.2.2 SPARQL Query Processing ....................................... 31
  4.3 Graph Mining Operations in SPARQL .................................. 32
    4.3.1 Graph Representation in RDF ..................................... 32
    4.3.2 Node Eccentricity (NE) .............................................. 32
    4.3.3 Triangle Counting (TC) ............................................... 33
    4.3.4 Connected Components (CC) ..................................... 37
  4.4 Performance Evaluation ...................................................... 38
    4.4.1 Experiment Setup ..................................................... 38
    4.4.2 Experimental Results ................................................ 39
  4.5 Conclusion ......................................................................... 42
LIST OF TABLES

Table 3.1 Characteristics of Datasets ................................................. 15
Table 3.2 Structural Properties of Benchmark Queries. ............................. 18
Table 3.3 Parameters for HDFS and Spark ........................................... 19
Table 3.4 Before-and-after Graph Size Changes ...................................... 21

Table 4.1 Dataset Properties ............................................................... 38
Table 4.2 Numbers of Iterated Records in TC Operations ...................... 41

Table 5.1 DBSCAN Terms ................................................................. 46
Table 5.2 Experiment Datasets ............................................................ 55
Table 5.3 Input Graphs and Number of Iterations in Connected Components .. 57

Table 6.1 Evaluation Datasets ............................................................. 75
Table 6.2 Generated Dataset Size ......................................................... 75
LIST OF FIGURES

Figure 1.1 Contributions ......................................................... 3
Figure 2.1 RDF Triples and Graph Representation ............................ 6
Figure 2.2 An Example SPARQL Query ........................................ 7
Figure 2.3 An Example Property Graph ....................................... 7
Figure 2.4 Graph Analytics Process .......................................... 8
Figure 3.1 Inverse Selectivity of Queries in Log Scale ....................... 18
Figure 3.2 Data Loading Time .................................................. 20
Figure 3.3 Query Execution Times in Log Scale ............................. 22
Figure 3.4 Per-system Query Execution Times ............................... 26
Figure 4.1 A Comparison of the Query Plans for TC Operations .......... 35
Figure 4.2 A Comparison of the Query Plans for CC Operations .......... 37
Figure 4.3 Execution Times on In-memory Triplestores .................... 39
Figure 4.4 Execution Times on Disk-based Triplestores ................... 40
Figure 5.1 A GeoSPARQL Example ............................................ 44
Figure 5.2 DBSCAN ($\epsilon = d$ and $MinPts = 3$) .......................... 47
Figure 5.3 Offloaded Pairwise Distance Computation ....................... 53
Figure 5.4 Multithreaded Spatial Searching .................................. 54
Figure 5.5 Performance Comparisons of Initial and Optimized Operations . 56
Figure 5.6 Execution Time of Connected Components ...................... 57
Figure 5.7 Multi-threaded Distance Computation ........................... 58
Figure 5.8 Clustering Result .................................................... 59
Figure 5.9 Clusters with A Non-spatial Attribute: Greek Administrative Geography . 63
Figure 5.10 Clusters with A Non-spatial Attribute: Geonames ............ 64
Figure 6.1 Preprocessing Step ................................................... 74
Figure 6.2 Data Loading Time with Different Graph Serialization .......... 76
Figure 6.3 Pre-processing Performance ....................................... 77
Figure 6.4 Community Detection Time ....................................... 78
Figure 6.5 SpGEMM-based vs. Two-step Approaches ....................... 79
Figure 6.6 CombBLAS vs. KDT ................................................. 80
<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 3.1</td>
<td>Example Data in OWL</td>
<td>13</td>
</tr>
<tr>
<td>Listing 3.2</td>
<td>Example Data in GraphML</td>
<td>14</td>
</tr>
<tr>
<td>Listing 3.3</td>
<td>Example Data in Property Graph/JSON</td>
<td>14</td>
</tr>
<tr>
<td>Listing 3.4</td>
<td>LUBM Query2 in SPARQL</td>
<td>15</td>
</tr>
<tr>
<td>Listing 3.5</td>
<td>LUBM Query2 in Cypher</td>
<td>16</td>
</tr>
<tr>
<td>Listing 3.6</td>
<td>LUBM Query2 in Gremlin</td>
<td>16</td>
</tr>
<tr>
<td>Listing 3.7</td>
<td>LUBM Query2 in Python with NetworkX</td>
<td>16</td>
</tr>
<tr>
<td>Listing 3.8</td>
<td>LUBM Query2 in Scala with GraphX</td>
<td>17</td>
</tr>
<tr>
<td>Listing 3.9</td>
<td>LUBM Query7</td>
<td>27</td>
</tr>
<tr>
<td>Listing 3.10</td>
<td>Optimized LUBM Query7</td>
<td>28</td>
</tr>
<tr>
<td>Listing 5.1</td>
<td>An Example GeoSPARQL Query</td>
<td>45</td>
</tr>
<tr>
<td>Listing 5.2</td>
<td>Pre-processing Query</td>
<td>50</td>
</tr>
<tr>
<td>Listing 5.3</td>
<td>Distance Computation Query</td>
<td>51</td>
</tr>
<tr>
<td>Listing 5.4</td>
<td>Core Points Retrieval Query</td>
<td>51</td>
</tr>
<tr>
<td>Listing 5.5</td>
<td>Graph Creation Query</td>
<td>51</td>
</tr>
<tr>
<td>Listing 5.6</td>
<td>Cluster Retrieval Query</td>
<td>52</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Graph Analytics on Modern Graph Processing Systems

Graphs are a powerful tool for modeling interactions of entities and extensively used in various domains to formulate real-world problems. In biology, for example, protein-protein interactions within a cell and metabolism in organisms are modeled as graphs [75]. Entities and their ties and interactions in social networks [54], and molecular substances and affinities between compounds in chemistry [14] also can be naturally captured in graphs. Its increasing popularity along with the advent of new technologies such as World Wide Web, social media, and online banking results in huge volume of graph databases. For example, DBpedia [22] reported that its 2014 knowledge graphs contain three billion triples in Resource Description Framework (RDF) [31] and PayPal announced that around six billion payment transactions were processed in 2016 [44].

As in traditional data analytics, typical tasks with graph databases are analyzing them in order to discover interesting and valuable knowledge, which is the goal of graph analytics. [51]. In graph analytics, graph pattern matching and graph mining are two fundamental tasks. Graph pattern matching is used for retrieving subgraphs from databases that match to user-defined graph patterns and conditions. Graph mining aims at finding hidden knowledge from graph databases and consists of various algorithms in multiple classes each of which has different functionality. Those two tasks have their own rights to graph analysis and are complementarily used for solving complex graph problems [110, 40, 112, 32, 107, 2]. Therefore, supporting those tasks is the key requirement of comprehensive graph analysis systems.

Recently, various graph processing systems have been introduced, and greatly help data scientists to perform their desired analysis tasks [34, 23, 45, 70, 104, 88, 96, 83, 101, 93, 15, 60]. They are backed on different base architectures (standalone server computers, CPU/GPU clusters, and shared-memory supercomputers) and storages (in-memory, relational database systems, and NoSQLs). They provide different graph data models (matrix, RDF, and property...
graph [10]) and query/programming interfaces (general programming languages, SPARQL [79], Gremlin [39], and Cypher [40]). Many of those systems come with built-in graph mining operations optimized by hands with which data scientists can perform their desired analysis tasks.

While recent systems give a number of options available for graph analysis tasks, data scientists cannot rely on a single system for extensive graph explorations since each system has limited features and capabilities. In case of graph pattern matching, those systems allow data scientists to describe their interesting patterns in their interfaces but required efforts to write such patterns differ according to the types of languages and users’ skillsets. Graph pattern matching performance also varies due to different levels of optimizations and efficiency of processing and storage engines. In graph mining, while providing a lot of graph mining operations, in-memory graph processing of NetworkX [34] cannot process even moderate-scale graphs [40]. Cluster-based high-performance systems such as GraphX [104], Giraph [83], TitanDB [88], GraphMat [93], and CombBLAS [15] have a few built-in graph mining operations. RDF-based systems such as Jena [45], GraphDB [70], and uRiKA-GD [60] are optimized for graph pattern matching workloads but provide no built-in graph mining operations. Therefore, it is important to compare and evaluate recent processing systems with various graph analysis workloads in order to allow data scientists to select exact systems for their tasks. On the other hand, it is highly recommended for each system to support more extended graph mining operations.

1.2 Contribution

In this dissertation, we first evaluate six recent graph processing systems in order to explore their advantages and limitations for graph pattern matching tasks. We consider RDF and property graph data models since graph pattern matching queries often involve conditions on graph entities’ attributes. Next, we extend some of those systems by developing a set of important graph mining operations. Specifically, we summarize the major contributions of our first work as follows.

- We developed a benchmark suite to evaluate RDF and property graph systems by extending a de facto standard for benchmarking RDF database systems. The benchmark suite provides extended graph serialization formats and benchmark queries written in several query/programming languages.
- We quantitatively and qualitatively evaluated six recent graph processing systems with the benchmark suite which are based on RDF and property graph data models.

The contributions of our second work are summarized as follows.
• We augmented RDF database systems as comprehensive graph analysis systems for RDF graphs by developing five fundamental graph mining operations in SPARQL interface. The operations include degree distribution [8], node eccentricity [94], triangle counting [97], connected components [63], and PageRank [72].

• We optimized those operations by analyzing query plans and shared our optimization process to give a guideline for developing more extended mining operations.

The contributions of our third work are summarized as follows.

• We developed a density-based clustering operation for RDF graphs. The operation clusters entities with their spatial and non-spatial attributes.

• We explained a limitation observed in current RDF database systems for spatial analysis, and proposed an optimization which avoids expensive pairwise distance computations by leveraging R-tree index and concurrent distance computations.

• We integrated the operation into a popular open source geospatial analysis tool called QGIS [80].

The contributions of our last work are summarized as follow.

• We developed a structural community detection operation on CombBLAS [15].

• We optimized the operation by (1) replacing an expensive matrix-matrix multiplication operation required to compute structural similarity [105] with row-wise intersections in pre-processing phase, and (2) reusing computed similarity for testing communities with different parameter setting, and (3) parallelizing those computations by using hybrid parallel programming.

Figure 1.1 summarizes our contributions and related chapters in this thesis.

Figure 1.1. Contributions
1.3 Outline

The rest of the thesis is organized as follows. In Chapter 2, we introduce graph analytics and present general graph analytics procedures. In Chapter 3, we present the methodology of our benchmark work and present evaluation results. Chapter 4 and Chapter 5 present the graph mining operations and the density-based spatial clustering operation for RDF graph databases, respectively. Chapter 6 introduces the structural community detection operation. Finally, we conclude this thesis and provide future work in Chapter 7.
Chapter 2

Modern Graph Analytics

2.1 Graph and Graph Data Models

Many real-world problems originating from computational sciences can be efficiently and intuitively represented as graphs. Example scientific disciplines that deal with graphs include, but not limited to, cheminformatics, bioinformatics, geoinformatics, astronomy, meteorology, material science, and geology. For example, in bioinformatics alone, graphs are applied to metabolic pathways, signaling pathways, gene regulatory networks, partonomies, chemical structure graphs, gene clustering, and topological adjacency relations [69]. A formal definition of a graph is shown in Definition 1.

**Definition 1** \( G = (V, E) \) is a graph where \( V \) is a set of elements and \( E \) is a relation where \( E \subseteq V \times V \). The elements of \( V \) are called nodes (or vertices) and the elements of \( E \) are called edges.

If edges in a graph have directions, the graph is called a directed graph and defined as shown in Definition 2

**Definition 2** A graph \( G = (V, E) \) is a directed graph if there is an arc from a node \( u \) to a node \( v \) where \((u, v) \in E\).

A labeled graph is a graph whose nodes and edges have labels and defined as shown in Definition 3.

**Definition 3** A labeled graph \( G = (V, E, L_V, L_E, \mu, \nu) \) is a graph that has four additional tuples: \( L_V, L_E, \mu, \) and \( \nu \) where \( L_V \) is a set of node labels, \( L_E \) is a set of edge labels, \( \mu: V \rightarrow L_V \) is a node labeling function, and \( \nu: E \rightarrow L_E \) is an edge labeling function. Specifically, a graph \( G = (V, E, L_V, \mu) \) is called a node-labeled graph and a graph \( G = (V, E, L_E, \nu) \) is called an edge-labeled graph.
If entities in a labeled graph have attributes associated to them, the graph is called a *property graph* and defined as shown in Definition 4.

**Definition 4** A *property graph* \( G = (V, E, L_V, L_E, A_V, A_E, \mu, \nu) \) is a labeled graph that has two additional tuples: \( A_V \) and \( A_E \) where \( A_V \) is a set of node attributes and \( A_E \) is a set of edge attributes.

For computer-aided graph processing, it is required that graphs are represented in computer-readable forms. Currently, matrix, *Resource Description Framework* (RDF) [31], and *property graph* [10] are frequently used for this purpose. First, matrix is a traditional way to represent graphs, and adjacency matrix and adjacency list are two representative data structures. In this model, it is complicated to assign attributes to graph entities except edge weights without special data structures. Next, RDF was established by the *World Wide Web Consortium* (W3C), and is a key standard for representing and linking resource on the Web. The basic unit in this model is a triple of \(<\text{subject}, \text{predicate}, \text{object}>\). Each triple (1) represents two resources on the Web and their relationship or (2) gives a resource an attribute. For example, assuming a university scenario, a statement “an associate professor in North Carolina State University whose name is John Doe teaches CSC478” can be represented with a set of triples as shown in Figure 2.1(a).

Note that resources and relation types are URIs, which can uniquely identify them on the Web. By the nature of the RDF data model, triples form directed graphs where predicates are correspondent to directed edges. For example, Figure 2.1(b) shows a graph representation of the triples in Figure 2.1(a).

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.ncsu.edu/professor#prof328">http://www.ncsu.edu/professor#prof328</a></td>
<td><a href="http://www.ncsu.edu/action#teaches">http://www.ncsu.edu/action#teaches</a></td>
<td><a href="http://www.ncsu.edu/course#csc478">http://www.ncsu.edu/course#csc478</a></td>
</tr>
<tr>
<td><a href="http://www.ncsu.edu/professor#prof328">http://www.ncsu.edu/professor#prof328</a></td>
<td><a href="http://www.ncsu.edu/property#type">http://www.ncsu.edu/property#type</a></td>
<td>&quot;Associate Professor&quot;</td>
</tr>
<tr>
<td><a href="http://www.ncsu.edu/professor#prof328">http://www.ncsu.edu/professor#prof328</a></td>
<td><a href="http://www.ncsu.edu/property#name">http://www.ncsu.edu/property#name</a></td>
<td>&quot;John Doe&quot;</td>
</tr>
</tbody>
</table>

(a) Triples

(b) Graph Representation

Figure 2.1. RDF Triples and Graph Representation
RDF graphs are managed by dedicated database management systems, called *triplestores* [45, 70, 25, 96, 60]. While those systems depend on different back-end processing platforms including RDBMS, NoSQL, clusters, and shared-memory supercomputers, they provide an logical view of triples and a standard query interface. The query interface allows users to interact with triplestores for managing and querying RDF data. The standard query language is called *SPARQL* [79] and it provides a declarative way for users to describe their queries. For example, a query to find the names of associate professors who teach CSC478 can be written in SPARQL as shown in Figure 2.2(a), which can be represented as a graph shown in Figure 2.2(b).

(a) SPARQL Query

(b) Graph Representation

Figure 2.2. An Example SPARQL Query

Property graph data model is frequently used in recent NoSQL-based graph processing sys-

Figure 2.3. An Example Property Graph
tems such as Dex [59], Neo4j [100], TitanDB [88], and GraphX [104]. As defined in Definition 4, nodes and edges have attributes (or properties) and they are assigned to those entities as key-values pairs. Figure 2.3 shows an example of property graph. Unlike RDF, there is no standard for graph serialization and query interface in this model.

### 2.2 Graph Analytics

![Figure 2.4. Graph Analytics Process](image)

Once real-world entities along with their ties/interactions have been modeled as graphs, routine tasks over such graph databases are analyzing them to discover valuable associations and properties, which positions graph analytics at the center of recent data analytics. For example, biologists examine protein-protein interaction graphs to investigate biological processes within a cell with various methods [75]. In social network analysis, it is an important task to identify and characterize special or exceptional relationships between social entities [7]. Figure 2.4 illustrates the general graph analytics process [51]. An interesting graph is extracted from data sources, transformed into a suitable form the target graph analysis system can process, and loaded into the system (ETL process). Then, the loaded graph is analyzed with three core tasks: graph pattern matching, graph mining, and graph visualization [110, 51, 32, 2, 38].

Graph pattern matching aims at finding interesting subgraphs occurred in graphs. Users can describe graph patterns through query/programming interfaces systems provide, and those patterns also form graphs as shown in Figure 2.2(a). Structural graph pattern matching is defined in terms of graph isomorphism [46].

**Definition 5** Graphs $G$ and $H$ are isomorphic if there is a bijection, $f$, that maps the node sets of $G$ and $H$ such that two nodes $u$ and $v$ are adjacent in $G$ if and only if $f(u)$ and $f(v)$ are also adjacent in $H$. 

8
Definition 5 shows that graph pattern matching aims to find all matched subgraphs in a graph $G$ which are isomorphic to a given pattern $P$. The pattern $P$ specifies structural constraints. In this thesis, we consider real-world graphs, which require additional constraints on node and edge labels and attributes [32].

Graph mining is another important task in graph analytics that aims to uncover hidden knowledge and obscure properties from graph databases [51, 1]. It consists of various operations which can be grouped into multiple categories according to their goals. Those categories include classification [55] and clustering [27, 105] that can be observed in traditional data mining. On the other hand, graph structures result in unique properties such as diameter, degree, and eccentricity to be mined [94, 8, 46], and bring about additional operation categories such as link analysis [72, 56].

Representing graphs in visualized forms is a key task for graph discovery process [38]. This task gives an intuitive way for data scientists to explore graph structures in an interactive manner. For example, they can iteratively figure out interesting parts of a graph and move to those parts for further exploration with visualization tools or export them for continuous analysis. Recent large scale graphs especially demand fast algorithms and effective representation formats [42, 81]. Note that we focus only on graph pattern matching and graph mining in this thesis.

In real-world graph analytics, graph pattern matching, graph mining, and graph visualization are cooperatively and interchangeably performed on graph databases. For example, graph pattern matching can be used for graph filtering [107] and dimensionality reduction [2] in order to reduce the computational complexity of following graph mining operations. In another example, new properties such as ranks and centrality scores computed by graph mining operations can augment graphs and used for filtering graph entities for graph pattern matching. Those tasks are repeated until data scientists perform enough exploration and, hence, gather desired results.

2.3 Modern Graph Processing Systems

The advent of new technologies such as World Wide Web and social media services along with increasing number of applications produce large scale graph databases. Accordingly, efficiently analyzing such databases is the core interest in graph analytics [93, 54, 85]. For this, a number of graph processing systems have been introduced [34, 23, 45, 70, 104, 88, 96, 83, 101, 93, 15, 60]. Those systems show high diversity in their graph data models, base architectures, storage backends, interfaces, and graph serialization formats. For example, NetworkX [34], Neo4j [23], Jena [45], and GraphDB [70] run on standalone computers. GraphX [104], TitanDB [88], uRiKA-GX [96], Giraph [83], GraphLab [101], GraphMat [93], and CombBLAS [15] leverage high-
performance clusters, and uRiKA-GD [60] is a shared memory graph processing supercomputer. Those systems rely on different graph data models such as matrix [101, 93, 15], RDF [45, 70, 96, 60], and property graph [34, 23, 88, 83]. Those systems are also backed on various storage sub-systems including in-memory [34], relational database systems, and NoSQL, and provide different query/programming interfaces such as general programming languages, SPARQL [79], Cypher [40], Gremlin [39], and so on.

Those systems typically allow users to describe graph patterns in their query/programming interfaces along with a set of primitive operations. However, diversity in those interfaces requires different amount of effort devoted to writing graph patterns. Moreover, those systems have different levels of optimizations, features, and capabilities, which makes it complicated for users to select optimal systems for their requirements. There are a few comparisons of recent graph processing systems. However, those comparisons are limited in that they evaluated systems of a specific graph data model such as RDF [32] and property graph [63, 102]. Next, most of the systems accompany hand-optimized built-in graph mining operations which can be used for graph analytics. However, they provide just a few such operations. For example, the latest version of GraphX (1.2.0) and CombBLAS (1.6.0) just have six operations. Moreover, RDF-based systems such as Jena, GraphDB, and uRiKA-GD do not have graph mining operations. To conclude, current graph processing systems have limited features, capabilities, and graph mining operations, which makes it unavoidable to interchangeably use those systems according to given tasks. Therefore, the ETL process often occurs multiple times for transforming and loading graphs onto different systems to apply various operations.
Chapter 3

Benchmark Graph Analysis Systems with Graph Pattern Matching Workloads

As discussed in Chapter 2, the advent of recent graph analysis systems broadens options that data scientists can select for their graph analytics tasks. Those systems show diversities in terms of graph data models, system architectures, query interfaces, and graph serialization formats. The presence of such diversity makes users to perform ad-hoc evaluations with desired workloads. Specifically, such evaluations require multiple fastidious steps: transforming datasets into several different serialization formats, loading datasets into systems via different interfaces, and writing and customizing queries in different query languages. Hence, a full investigation into each system is problematic in that it is time consuming, error-prone, and can easily cause a deviation from the ultimate goal.

This study aims at providing an evaluation of recent six graph analysis systems with graph pattern matching workloads. More specifically, we focus on graph analysis systems which are based on the two popular graph models: RDF and property graph. In order to provide full understandings of those systems, we compare those systems in terms of not only graph pattern matching performance but also different aspects of importance such as expressiveness of query languages, optimization, graph serialization, and before-and-after graph data size.

3.1 Benchmarked Graph Analysis Systems

Triplestores such as Apache Jena [45], Eclipse RDF4J [91], OpenLink Virtuoso [76], and Onto-text GraphDB [70] are considered as graph database management systems for RDF graphs since they support transactions. uRiKA-GD [60] from YarcData (a sister company of Cray) is also
a triplestore, which is built around a shared-memory supercomputer architecture, previously known as the Cray XMT. The software stack of the system processes each SPARQL query in parallel by leveraging a huge number of hardware threads. For example, uRiKA-512 has 512 Threadstorm IV graph accelerators (processors) with 128 hardware threads per processor [43]. It has separate 68 x86-based processors dedicated to parallel I/O. Processors are interconnected in a Cray-designed 3D torus network.

Property graph-based systems are broadly categorized into two classes. First, graph database systems are designed specifically for optimal graph data store and retrieval over NoSQL database systems. Examples in this class include Dex [59], Neo4j [23], and Titan [88]. Second, systems such as NetworkX [34], and GraphX [104] are general purpose graph processing systems and typically focus on graph analysis rather than transactional graph data management. Each of those systems provides a distinct computation model and built-in graph mining operations optimized to the model. For property graph-based systems, there is no standardized query interface. For example, Neo4j and Titan provide their own query languages called Cypher [40] and Gremlin [39], respectively. NetworkX, itself, is a Python library for in-memory graph analysis and the general programming interface of Python is the only way to write graph analysis queries. GraphX is another graph analysis library running on top of Spark [109]. GraphX queries are written in several programming languages such as Python and Scala by using built-in data structures and operations.

3.2 Benchmark Suite Design

For comprehensive evaluations, it is inevitable to rely on a benchmark suite that supports various graph serialization formats and query languages while managing the structural characteristics of benchmark datasets as the same. However, existing graph benchmark suites do not satisfy those requirements. Therefore, we developed a benchmark suite by extending a popular benchmark suite for triplestores called Lehigh University Benchmark (LUBM) [33]. The extension supports several graph serialization formats and benchmark queries in different languages for property graph-based systems. In this section, we explain the benchmark suite and introduce our extension.

3.2.1 LUBM Benchmark Suite

LUBM was originally developed for evaluating the performance of triplestores along with their reasoning capabilities [33] and is a de facto standard for evaluating RDF and NoSQL graph database systems [71]. Its synthetic data generator produces datasets in two web ontology languages: OWL [57] and DAML+OIL [41]. Datasets are randomly generated and can be repeated
Listing 3.1. Example Data in OWL

by using same seed parameter values. They simulate a university use-case where entities in universities are represented as nodes and their associations as edges. The unit of data generation is a university and the number of universities is used as a scale factor. In each university, entities such as students, professors, and departments are randomly connected within certain ranges of degree. The details on the data profile of LUBM can be found at [36]. The benchmark query set consists of 14 queries written in SPARQL, each of which shows different selectivity, complexity, class hierarchy, and inference requirement.

3.2.2 Extension for Data Generation

LUBM datasets are generated in two serialization formats in RDF and they are rarely supported by current property graph-based systems. In addition, triplestores often require different formats for data loading. Hence, we extend the LUBM data generator so that it is able to generate benchmark datasets in several graph serialization formats. For triplestores, the extension generates datasets in N-triple, which is an RDF data exchange format supported by the majority of current triplestores. For property graphs, however, it is not straightforward to generate datasets, preserving the relevance of semantics with minimizing redundancy while supporting current graph analysis systems. This is because (1) there is no standard for graph serialization formats, and (2) different systems have their own requirements on serialized graph structure even though they have a common serialization format. Specifically, GraphML [30] is supported by NetworkX and Neo4j but not by GraphX. In addition, Neo4j strictly expects node descriptions to be placed before edge descriptions in a GraphML document. Addressing such difficulties, our extended data generator produces property graphs in two serialization formats which are GraphML formatted to be compatible with NetworkX and Neo4j and Property Graph/JSON [50] for GraphX. For example, Listing 3.1 shows an example data in OWL generated by the original LUBM data generator. The information is represented in GraphML as shown in Listing 3.2. In the beginning, all attributes for nodes and edges are declared with "key" elements. Then, node descriptions are enforced to place before edge descriptions so that resulting GraphML files are compatible to all the systems we benchmark. In RDF, a URI is used as an internal identifier of a node. However, such URIs are internally replaced with numeric IDs.
in property graph-based systems. Therefore, we add an additional “uri” attribute in GraphML so that nodes can be identified with their URIs in queries. We add a “labels” attribute to each “node” element for Neo4j to recognize the type of the node. Each edge has its ID, and source and destination nodes’ URIs which are encoded as internal numeric IDs. We add an “label” attribute to represent the types of edges since it is required to retrieve edges by their types.

The extended data generator also produces datasets in Property Graph/JSON for GraphX. Nodes and edges are serialized in two separate files: nodes.json and edges.json as shown in Listing 3.3. Since GraphX does not provide a Property Graph/JSON data loader, we implemented a custom loader. It loads the two files and generates a graph RDD (Resilient Distributed Dataset), which is the data structure for GraphX. Each graph entity is declared between {} and each attribute is represented as attribute_name: attribute_value. Attributes are delimited by commas. Since URIs are replaced with unique internal IDs when data are loaded, they are stored as separate attributes.

Table 3.1 shows the characteristics of graphs generated by the extended data generator with universities = 100, 1,000, and 10,000. Data size, the number of nodes and edges, and the
### Table 3.1. Characteristics of Datasets

<table>
<thead>
<tr>
<th>Universities</th>
<th>Size (GB)</th>
<th>Nodes</th>
<th>Edges</th>
<th>Triples</th>
<th>Triangles</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.1 (OWL)</td>
<td>2,179,766</td>
<td>6,645,928</td>
<td>13,405,381</td>
<td>462,078</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2.0 (GraphML)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.5 (JSON)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.9 (N-Triple)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>11.3 (OWL)</td>
<td>21,715,766</td>
<td>66,219,021</td>
<td>133,573,854</td>
<td>4,613,477</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20.1 (GraphML)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13.9 (JSON)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>18.9 (N-Triple)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>115.6 (OWL)</td>
<td>217,006,852</td>
<td>661,597,197</td>
<td>1,334,681,190</td>
<td>46,014,940</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>202.5 (GraphML)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>140.7 (JSON)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>190.7GB (N-Triple)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SELECT ?X ?Y ?Z
WHERE {
  ?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:University .
  ?Z rdf:type ub:Department .
  ?X ub:undergraduateDegreeFrom ?Y
}

Listing 3.4. LUBM Query2 in SPARQL

The number of triangles are proportional to the number of universities while the generated graphs have a single connected component. In each dataset, triples corresponding to node properties occupy 50% of the total number of triples. Different formats result in different sizes of generated datasets. For example, GraphML produces the largest datasets to represent the same graphs.

#### 3.2.3 Query Translation

While triplestores have a standard query interface, property graph-based systems provide different query interfaces. Hence, it is necessary to rewrite the benchmark queries in several languages to compare those systems. For query selection and rewriting, we took the approach proposed in [32]. We selected 9 queries out of 14 original LUBM queries and those queries were written in two query languages (Cypher and Gremlin) and two general programming languages (Python with NetworkX and Scala with GraphX).

Cypher is the default query language of Neo4j. It is designed to describe graph patterns and provides an intuitive way similar to SPARQL, which allowed us to rewrite queries with a minimal effort. Listing 3.5 shows the LUBM query2 in Cypher which is translated from the original SPARQL query in Listing 3.4.

Gremlin is a procedural graph query language used by Titan and a part of Apache Tinkerpop project [78] which is an open source graph computing framework. A Gremlin query is described...
MATCH
  (x:GraduateStudent)-[:memberOf]->(z:Department),
  (z)-[:subOrganizationOf]->(y:University),
  (x)-[:undergraduateDegreeFrom]->(y)
RETURN
  x.uri, y.uri, z.uri;

Listing 3.5. LUBM Query2 in Cypher

graph.V.filter {it. type == 'GraduateStudent' }
  .as('graduate_student')
  .out('memberOf').filter {it. type == 'Department' }
  .out('subOrganizationOf')
  .filter {it. type == 'University' }
  .inE('undergraduateDegreeFrom')
  .outV.retain ('graduate_student')
  .sideEffect { f << "${it.uri}\n"} .iterate()

Listing 3.6. LUBM Query2 in Gremlin

as a collection of predefined steps each of which maps to a particular pipe operation as shown in Listing 3.6. Chaining pipes creates a pipeline where graph entities are processed through the pipeline. Each pipe operation broadly falls into four categories: transform, filter, sideEffect, and branch. Steps can easily represent a traverse along edges in a graph. It also provides a few primitives that allow such traversals to resume from previous steps such as back and reference intermediate records produced at previous steps such as retain. Gremlin especially focuses on graph traversals and this makes it less straightforward to write graph pattern matching queries. We discuss the limitations of the language for writing pattern matching queries in Section 3.3.3.

Listing 3.7 shows the query2 implementation in Python with NetworkX. Describing graph patterns in NetworkX requires iterating nodes in graphs and, in turn, edges for each node. This often resulted in nested loops for writing the benchmark queries via the given interface.

for x in graph:
  if x['type'] == 'GradStudent':
    for edge in graph.edges(x):
      if edge['type'] == 'undergraduateDegreeFrom':
        y = edge.neighbor
      if edge['type'] == 'memberOf':
        z = edge.neighbor
      if y and z are matched:
        if y['type'] == 'University':
          if z['type'] == 'Department':
            if graph.has_edge(y,z,'subOrganizationOf'):
              print x,y,z

Listing 3.7. LUBM Query2 in Python with NetworkX
val memberOf = graph.triplets.filter(
    triplet => (triplet.srcAttr.type == "GraduateStudent" &&
    triplet.dstAttr.type == "Department" &&
    triplet.attr == "memberOf"
  )
).map {
  triplet => (triplet.srcId, (triplet.srcAttr.uri, triplet.dstId, triplet.dstAttr.uri))
}
val undergraduateDegreeFrom = graph.triplets.filter(
    triplet => (triplet.srcAttr.type == "GraduateStudent" &&
    triplet.dstAttr.type == "University" &&
    triplet.attr == "undergraduateDegreeFrom"
  )
)
...
val subOrganizationOf = ...
# joining triplets to retrieve patterns
val temp = memberOf.join(undergraduateDegreeFrom)
...
val result = temp.join(subOrganizationOf)
...

Listing 3.8. LUBM Query2 in Scala with GraphX

GraphX provides a set of graph processing primitives that can be used for graph analysis. For describing LUBM queries, we extensively used triplet and join. Triplet, which is inspired by RDF triples, provides a node-edge-node view of a graph and an object-oriented view of triplet for accessing attributes of graph entities. filter and map operations are used to select triplets that we are interested in and transform them into desired structure. Then the connection of the selected triplets is performed by join operations. Listing 3.8 shows the simplified code of LUBM query2 for GraphX.

The benchmark queries have different properties in terms of structure and selectivity. For example, those queries have different pattern structures as shown in Table 3.2. "Neighborhood" is a basic triple pattern which consists of a node and its neighbor. "Chain" is a pattern consisting of multiple triples where the object of a pattern is the subject of another pattern. "Star" consists of a subject and multiple objects connected to it and "Triangle" consists of three triples where three nodes are adjacent to each other forming a triangle. Figure 3.1 shows the inverse selectivity of the queries in log scale, which is computed as

$$inverse \ selectivity = \frac{number \ of \ output \ records}{number \ of \ input \ triples} \times 100 \quad (3.1)$$

Each of Q4, Q5, Q7, Q8, and Q12 returns a constant number of output records regardless of the number of universities. Q6, Q9, and Q14 retrieve increasing numbers of records at an almost
same rate as input size increases. Q2 produces more output records when universities = 1,000 than universities = 100 but the same number of output records when universities = 10,000.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Q2</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q12</th>
<th>Q14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighborhood</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Chain</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Star</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Triangle</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Multi-edge</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Inference</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 3.1. Inverse Selectivity of Queries in Log Scale

### 3.3 Benchmark Results

In this section, we start with describing the evaluation environments. We evaluated six graph analysis systems with our extended benchmark suite. First, we compare the system quantitatively in terms of data loading and query processing. Then, we present qualitative comparisons of those systems.
Table 3.3. Parameters for HDFS and Spark

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfs.replication</td>
<td>3</td>
</tr>
<tr>
<td>dfs.blocksize (in byte)</td>
<td>268435456</td>
</tr>
<tr>
<td>spark.driver.memory</td>
<td>5g</td>
</tr>
<tr>
<td>spark.executor.memory</td>
<td>40g</td>
</tr>
<tr>
<td>spark.executor.extra.JavaOptions</td>
<td>-XX:+UseG1GC</td>
</tr>
<tr>
<td>spark.shuffle.memoryFraction</td>
<td>0.3</td>
</tr>
<tr>
<td>spark.storage.memoryFraction</td>
<td>0.5</td>
</tr>
</tbody>
</table>

3.3.1 Evaluation Environments

In this benchmark, we selected NetworkX 1.9.1, Jena Fuseki 1.1.1 with TDB (a disk-based native store), Neo4j 2.2.3 community edition as standalone graph analysis systems. Those systems were tested on a desktop that has an i5 1.3GHz quad-core processor, 16GB DDR3 RAM, and a 250GB solid state disk. For Java-based systems (Jena and Neo4j), we configured the JVM maximum heap size to 80% of the physical memory.

For high-performance systems, we tested GraphX packaged in Spark 1.3.1, Titan 0.5.1 with HBase 1.0 as the backend storage, and uRiKA-64. GraphX and Titan were tested on a cluster at Oak Ridge National Laboratory consisting of 9 virtual machines from CADES (Compute and Data Environment for Science) [49]. Each virtual machine has 32 virtual processors, 64GB RAM, and a 500GB local disk. For the underlying storage, we used Hadoop Distributed File System (HDFS) which is packaged in Hadoop 2.4.0. For both systems, we used 8 worker nodes (region servers for HBase). For Titan, we used three nodes of zookeeper quorum with all default parameters. We configured HDFS and Spark as shown in Table 3.3. We used a uRiKA-64 system at Oak Ridge National Laboratory. It is a supercomputer-based graph analysis system and has a 2TB shared-memory, 8,192 hardware threads, and 125TB Lustre file system.

3.3.2 Quantitative Evaluation

Data loading time  Figure 3.2 shows data loading times of five graph analysis systems. Note that we did not measure data loading time of GraphX since data loading and processing are overlapped. NetworkX showed the best data loading performance among standalone systems with U100. However, it was not comparable to show the benefit of in-memory graph data loading. Moreover, NetworkX could not load more than U100 since the data size explosively increased when loaded. Jena showed 16-29% better data loading performance than Neo4j. However, it could not load U500 due to insufficient heap space. As for Titan, the data loading time sharply increased when the number of universities ≥ 1K. For universities ≥ 3K, we stopped the
data loading after 12 hours. As many property graph-based systems, Titan internally allocates IDs for node and edge entities. It allocates a block of IDs ahead of creating entities. However, those pre-allocated IDs are depleted too soon compared to the data ingestion rate of back-end storages. In addition, the use of HBase backend contributed to linear increase of loading time. When inserting a new entry, it first fills a region in one region server. When the region size exceeds a configured limit, the region splits. Then, if the number of regions in the region server exceeds a limit, it finds a new region in another region server. Therefore, the loading process did not fully utilize multiple region servers when we bulk-load graphs. In uRiKA, multiple I/O processors enabled concurrent data loading and database creation. This parallelized I/O and data processing capability resulted in the best data loading performance out of the five systems.

Figure 3.2. Data Loading Time

Graph Size Changes after Loading in order to check the effect of data loading in terms of storage consumption, we measured the change on graph size after loading U100, U200, and U300. Table 3.4 shows the before-and-after graph sizes along with change ratios. When loaded on NetworkX, the size of graph data increased tremendously since it produced a lot of Python objects in hierarchical dictionaries for labels and attributes. For Jena and Neo4j, the input data additionally occupied 25 - 88% more storage space for storing metadata and additional data structures. In case of Titan, however, the graph size decreased around 89% since it maintains edges in a compressed way and graphs are compressed when they are loaded into HBase. On
Table 3.4. Before-and-after Graph Size Changes

<table>
<thead>
<tr>
<th>System</th>
<th>Size</th>
<th>U100</th>
<th>U200</th>
<th>U300</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input Size</td>
<td>1.8GB</td>
<td>3.7GB</td>
<td>5.5GB</td>
</tr>
<tr>
<td>NetworkX</td>
<td>Loaded Size</td>
<td>13.1GB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Increase Ratio</td>
<td>627.7%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Jena</td>
<td>Input Size</td>
<td>1.0GB</td>
<td>2.0GB</td>
<td>3.1GB</td>
</tr>
<tr>
<td></td>
<td>Loaded Size</td>
<td>1.9GB</td>
<td>3.9GB</td>
<td>5.8GB</td>
</tr>
<tr>
<td></td>
<td>Increase Ratio</td>
<td>88.5%</td>
<td>88.5%</td>
<td>88.5%</td>
</tr>
<tr>
<td>Neo4j</td>
<td>Input Size</td>
<td>1.8GB</td>
<td>3.7GB</td>
<td>5.5GB</td>
</tr>
<tr>
<td></td>
<td>Loaded Size</td>
<td>2.3GB</td>
<td>4.6GB</td>
<td>6.9GB</td>
</tr>
<tr>
<td></td>
<td>Increase Ratio</td>
<td>25.2%</td>
<td>25.2%</td>
<td>25.2%</td>
</tr>
<tr>
<td>Titan</td>
<td>Input Size</td>
<td>1.8GB</td>
<td>3.7GB</td>
<td>5.5GB</td>
</tr>
<tr>
<td></td>
<td>Loaded Size</td>
<td>200MB</td>
<td>405MB</td>
<td>615MB</td>
</tr>
<tr>
<td></td>
<td>Increase Ratio</td>
<td>-89.4%</td>
<td>-89.4%</td>
<td>-89.4%</td>
</tr>
<tr>
<td>uRiKA</td>
<td>Input Size</td>
<td>1.9GB</td>
<td>3.8GB</td>
<td>5.6GB</td>
</tr>
<tr>
<td></td>
<td>Loaded Size</td>
<td>1.9GB</td>
<td>3.8GB</td>
<td>5.6GB</td>
</tr>
<tr>
<td></td>
<td>Increase Ratio</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

uRiKA, it stores graphs as in their input formats, which resulted in no change on the graph size.

**Query Processing Time** we examined the six graph analysis systems to check their pattern matching performance. Figure 3.3 compares the execution times of individual pattern matching queries. As we explained previously, NetworkX could load only the U100 dataset. With the dataset, it showed the best performance for Q5, Q6, and Q9 among standalone systems. It is notable that NetworkX performs in-memory graph analysis which avoids expensive disk I/Os but it did not show any benefit of in-memory processing. The inefficiency on its programming interface such as loop-based graph iterations degrades the advantage of in-memory analysis. Among standalone systems, Jena processed many of the queries relatively well and showed comparable performance against Q2, Q4, Q12, and Q14 while Neo4j showed the best performance against Q7 and Q8. For high performance systems, uRiKA was superior to the others for most of the queries with 1K or more universities. For Q6 and Q12 with some datasets, GraphX showed slightly better performance but the difference was negligible. uRiKA also showed better performance for Q2, Q5, Q6, Q7, Q9, and Q14 with universities < 1K than standalone analysis systems; however, for Q4 and Q12, Jena outperformed uRiKA with those datasets. We omitted the results of Q2, Q9, and Q12 for Titan due to a limitation of Gremlin, which will be discussed in Section 3.3.3. Titan showed excessively slower performance than the other systems for the remaining queries.
Figure 3.3. Query Execution Times in Log Scale
Next, we checked the effect of query selectivity on query execution time. As shown in Figure 3.1, each benchmark query has a distinct selectivity and a selectivity pattern. We observed that query selectivity affects the query processing times on some systems as shown in Figure 3.4(a)-3.4(e). For example, Neo4j and uRiKA showed nearly flat or gentle slopes on execution times for queries that have increasing selectivities (e.g., Q4, Q5, Q7, Q8, and Q12). For queries that have constant selectivities (e.g., Q6, Q9, and Q14), the execution times increased more steeply on Neo4j and exponentially on uRiKA. Jena also showed similar patterns for several queries but we observed certain exceptions. We will discuss the observation in Section 3.3.3. Different from Neo4j and uRiKA, we could not observe strong correlation between query selectivity and processing time from Titan and GraphX. We infer that it is because per-partition query selectivities can be different from the global query selectivities depending on data partitioning policy, as graph data are partitioned in multiple nodes in distributed graph analysis platforms.

### 3.3.3 Qualitative Evaluation

**Data Serialization and Loading** support for various graph serialization formats along with corresponding data loaders is a fundamental requirement for graph analysis systems that deal with graph data from heterogeneous sources. In RDF, there are several serialization formats. Jena could support those formats. However, we observed that uRiKA only supports two of them: N-triple and N-quad. Some of the property graph-based systems support GraphML natively or with third-party plugins. However, GraphX does not support GraphML and there exists no available plugins. We notice that it is not trivial to implement a GraphML loader for distributed graph analysis systems. The major difficulty is that simply partitioning GraphML data over multiple nodes results in loss of required information. First, GraphML locates metadata about node and edge properties and actual graph data in separate places. Next, it uses XML notation where each element is described over multiple lines and elements are structured in a nested manner. Therefore, it is necessary to pre-processing GraphML data but it adds additional costs [12]. Thus, we chose Property Graph/JSON for GraphX since a JSON object represents a node or an edge and each object can be naturally represented as a line in generated graph datasets.

**Query Interface** for graph pattern matching, Cypher and SPARQL have the best expressiveness in that they expose intuitive ways for users to list interesting structural constraints in declarative ways. On the other hand, Gremlin has several limitations on expressing graph pattern queries compared to Cypher and SPARQL. First, Gremlin users should consider not only structural constraints but also orders in which nodes and edges are traversed. This is be-
Figure 3.4. Per-system Query Execution Times
cause Gremlin is designed based on the principle of graph traversal rather than graph pattern matching. Next, it is not trivial and sometimes not possible to write complex pattern queries to retrieve results in a desired fashion. For example, Q9 finds triangles by traversing graphs as students→professors→courses→students and returns the URI attribute of retrieved students. Although we can retrieve the result set with the same size (which implies that the same patterns have been identified), it was not trivial to save or print out results as triangular patterns as the original query intends – a collection of URIs in the form of student, professor, and course in a single line from three vertices in the matched subgraphs, instead of URI only from students. It is primarily because each traversal operation in Gremlin finds vertices and, thus, only outputs properties of vertices at the current step, instead of expressing properties of vertices from matched subgraphs. In this work, our goal was to evaluate the ability to obtain the same analysis results, which should include the time to retrieve the result set and present to users. As a result, this study omitted to implement query 2, 9, and 12 in Gremlin.

**Optimization Issues**  Cypher and SPARQL are declarative query languages. An important advantage of such languages is that they can decouple query optimization tasks from query writing tasks. However, we found a significant problem on Jena’s query optimization. Figure 3.4(f) shows the execution times of Q4, Q7, and Q12 on Jena. Those queries have the highest selectivities and show similar selectivity patterns. However, the execution times of those queries showed quite different patterns: while the execution times of Q4 and Q12 are low and increase slowly or remain steady, the execution time of Q7 is relatively high and increases proportional to the number of universities. Having analyzed the query plan of Q7, we found that Jena produces different query plans depending on the order of triple patterns. For example, Listing 3.9 shows Q7 and Listing 3.10 shows an equivalent query. The only difference of the two queries is that the last triple pattern is just moved next to the first one. However, the query in Listing 3.10 resulted in a more optimized query plan which contains less number of operations, which resulted in 99% improvement on execution time as shown in Figure 3.4(f). In addition, the optimized
query showed an almost similar pattern as Q4 and Q12. This observation shows that it is not guaranteed that Jena always produce optimal query plans regardless of triple pattern orders. That is, how a user represents a pattern as a query can significantly affect the performance. In case of Gremlin, there are multiple ways to traverse a given graph pattern, each of which results in different query selectivity. Titan does not change user-defined traversal orders being aware of selectivity. Therefore, users are responsible for optimizing queries by finding optimal traversal orders. In addition, complex graph patterns often break traversals. That is, they cannot be traversed in a “connecting-the-dots” manner and require to jump to previously visited nodes. This also requires intermediate states at certain points to be recorded during traversals. Hence, Gremlin users need to explicitly represent such decisions in their queries by using a set of relevant primitives. In NetworkX, complex graph pattern matching queries produce several nested loops. Since the size and the level of each loop along with the number of loops can increase computational complexity, minimizing the overall cost is users’ responsibility. In GraphX, triplets are mapped to triples in SPARQL and join operation links triples by their nodes. With the use of such primitives, graph patterns which have multiple edges and long chains produce multiple join operations in their query representation. In such cases, a typical optimization is ordering joins to minimize query processing costs. However, the order of performing joins is still on users’ hand.

3.4 Conclusion

In this work, we benchmarked recent graph analysis systems including triplestores and property graph-based systems with graph pattern matching workloads. For this benchmark, we extended the de facto standard LUBM benchmark in order to support systems in both graph data models and different graph serialization formats and query languages. The benchmark results show that the evaluated systems have their own advantages and limitations in different aspects of graph pattern matching tasks. Triplestores showed good performance against many of the benchmark
queries in general. In addition, their standardized SPARQL query interface was easy-to-use and intuitive enough to describe graph patterns with a minimum of effort. Such a standard query interface allows query writing decoupled from query optimization and processing. However, we found that there is a significant problem on Jena’s query optimization, which diluted the advantage. Property graph-based systems show large variations in different aspects. Neo4j showed good performance among standalone graph analysis systems. Its Cypher query language provides an intuitive way to describe graph patterns similar to SPARQL. Among distributed graph analysis systems, GraphX showed the best graph pattern matching performance. However, as graph size increased, the performance was exponentially degraded in our settings. We expect that a more elaborated setting of system parameters can optimize the performance of GraphX but it requires more users’ efforts. The data compression strategies of Titan with HBase back-end allowed graphs to be managed in a compact way but it showed a few orders of magnitude slower query execution times than the other systems. Also, it was not intuitive to write graph patterns in its Gremlin language.
Chapter 4

Graph Mining Operations for RDF Graphs

4.1 Introduction

As discussed in Chapter 2, Resource Description Framework (RDF) is one of important computational graph data models. While the primary goal of RDF query processing is graph pattern matching, it overlooks the importance of graph mining tasks which are another integral part in graph analytics. Addressing the problem, recent research efforts proposed a few graph mining operations [46, 95]. In [46], the authors proposed a triangle finding operation for directed graphs. The operation finds triangles by processing a single SPARQL query. In the query, three representative patterns for possible triangles are defined and they are used for pruning redundant isomorphic triangles in advance, which speeds up the triangle finding operation. The authors in [95] proposed a programming abstraction for building iterative graph mining operations in SPARQL and introduced three iterative operations based on the abstraction: peer-pressure clustering, graph diffusion, and label propagation. Those operations can be considered as good initiatives that extends triplestores to holistic and native graph analysis systems. However, those efforts deliver only a single non-iterative graph operation for directed graphs or a general idea for building iterative operations with three operations. Hence, the number of available operations is still limited. Moreover, they are not optimized through query plan analysis which can give important guideline for developing other graph mining operations. In our recent work [51], we proposed five important graph analysis operations to provide extended native graph mining operations. Those operations are highly optimized through query plan analysis. We introduce three operations of them along with their optimization processes.
4.2 Background

We start with brief explanation of three graph mining operations we present in this chapter: node eccentricity, triangle counting, and connected components. Then, we describe the query processing mechanism of a popular open-source triplestores, Apache Jena [45], which is the base triplestore for our optimization.

4.2.1 Graph Mining Operations

Given a graph $G = (V, E)$ where $V$ and $E$ are the sets of nodes and edges, respectively, graph mining aims at finding obscure properties and knowledge from $G$. It consists of a broad range of analysis operations, each of which reveals different aspects of $G$. The eccentricity of a node $x \in V$ is defined as $\max_{y \in V} d(x, y)$ if $G$ is a connected graph and, otherwise, $\infty$ where $d(x, y)$ is the distance of a shortest path from $x$ to any node $y$. Node eccentricity is an important topological attribute of a graph in that it can be extended to compute other attributes such as radius and diameter of graphs [94]. Triangle counting is another important graph operation which counts triangles in $G$ [46]. Formally, a triangle is a subgraph that consists of three nodes $\{x, y, z\} \subset V$ where there exist $\{\{x, y\}, \{y, z\}, \{z, x\}\} \subset E$. For this operation, it is especially critical to design an efficient processing algorithm since it can produce a huge amount of intermediate results during computation. Connected components aims at finding subgraphs, $G_1 = \{V_1, E_1\}, ..., G_n = \{V_n, E_n\}$ where, for $i \neq j, V_i \cap V_j = \phi$, and $V = V_1 \cup ... \cup V_n$ and $E = E_1 \cup ... \cup E_n$. This operation is important in that many real-world applications necessitate the decomposition of large graphs into connected components [63].

4.2.2 SPARQL Query Processing

Apache Jena is a popular open-source triplestore [46]. Its query processing engine is used for several extended triplestores such as Parliament [48] and uRiKA, a Cray’s supercomputer-based graph analysis appliance [43]. Jena executes query plans in an iterator-based manner [35]. In this approach, each operator in a query plan is mapped into a corresponding iterator. At query execution time, the query engine calls the root iterator in a query plan and the calling propagates through the iterator tree, returning one record per call. It is notable that some iterators such as ones for GROUP BY and JOIN make the query plan staged. That is, those iterators require all input data to be ready before they are performed. This staged query execution can make a huge impact on the overall query processing performance if a query plan contains several stages which generate excessively large amount of intermediate data. Jena provides a staging-free counterpart operation of JOIN, named SEQUENCE. This operation is more efficient since it returns an intermediate record immediately without waiting for the entire intermediate data.
have been processed and its physical implementation is much more efficient than that of JOIN.

### 4.3 Graph Mining Operations in SPARQL

In this section, we present three graph mining operations along with the optimization process we performed.

#### 4.3.1 Graph Representation in RDF

Since RDF graphs are directed, it is necessary to represent undirected graphs in the RDF data model. In order to do this, two general approaches can be considered. First, each undirected edge can be stored twice. The second approach is that undirected edges are stored as if they are directed. When graph operations are performed, their “undirect” properties are reproduced from directed edges. We use the latter approach to reduce the storage space.

#### 4.3.2 Node Eccentricity (NE)

Algorithm 1 shows the NE operation. At the initialization step, the NE operation creates a temporary named graph, $G_T$, which maintains intermediate state and inserts a starting node $n$. At the computation step, the operation retrieves the one-level neighbors of each node in $G_T$ and stores them into $G_T$ with the next iteration number, $I_{next}$. This process is repeated until the number of nodes in $G_T$ converges. As a convergence condition, the operation counts the
number of triples in $G_T$ and compares it with the previous counter. If the counter is equal to the previous counter, Converges() returns true and the processing step ends. The eccentricity of $n$ is equal to the number of iterations if $|N| = |G_T|$. Otherwise, the eccentricity is infinite since it means that there are disconnected components in the graph. At the finalization step, the intermediate state is discarded by dropping $G_T$. A naïve SPARQL query may iterate over every node in $G_T$ and try to insert into $G_T$ nodes that have been already inserted at each iteration. This approach possibly yields not only unnecessary computations but also large intermediate data during query processing. In order to avoid the problem, we leverage tightly binding triple patterns and filter operations SPARQL provides. First, nodes inserted to $G_T$ at iteration $I$ are labeled with $I + 1$ so that only those nodes can be iterated at the next iteration. Second, we minimize intermediate triples passed to INSERT command by using a NOT EXIST filter.

4.3.3 Triangle Counting (TC)

As shown in Algorithm 2, the initial TC operation requires a single issue of a query to count the triangles in $G$. It unions two patterns `<?s ?p ?o>` and `<?o ?p ?s>` per triangle edge for recovering the “undirected” property and joins three edges on their common nodes. Two filter conditions are used to reduce isomorphic triangle patterns. Those two filter conditions sufficiently impose the necessary ordering because $\text{STR}(?x) < \text{STR}(?z)$ is automatically met by the transitive property of inequality. Despite its simplicity, the initial operation, however, is not optimized in that it can produce a lot of intermediate data through a complex query plan. Figure 4.1(a) shows the query plan generated from the initial TC operation generated by the Jena query engine. The query plan has a SEQUENCE operation that consists of three large children and it iterates the entire triples several times for each triple returned from one child operation tree. Note that if we add the unnecessary filter condition $\text{STR}(?x) < \text{STR}(?z)$ into the query,
then the three children perform semantically same computations, increasing the computational complexity. In addition, there are still many duplicate triples flowing into the GROUP BY operation which are later removed by a DISTINCT keyword in the operation. In order to minimize such computational complexity and unnecessary intermediate data, we optimize the initial algorithm as shown in Algorithm 3. At initialization step, the optimized TC operation

**Algorithm 3 Optimized Triangle Counting**

```sql
function INITIALIZE
    CREATE GRAPH G_T;
    INSERT { G_T { ?node <link> ?neighbor }}
    WHERE {
        SELECT ?node ?neighbor
        WHERE {
            ?node <edge> ?neighbor .
            FILTER(STR(?node) < STR(?neighbor))
        }
    }
function PROCESS
    SELECT (COUNT(DISTINCT *) AS ?count)
    WHERE {
        G_T {
        }
    }
function FINALIZE
    DROP GRAPH G_T;
```

creates a temporary named graph, $G_T$. Then, it retrieves all the edges in $G$ and stores edges that meet the lexicographic ordering enforced by the filter condition into the temporary graph. Instead of a UNION operation, we used a property path expression ?node <edge> ?neighbor to recover the “indirect” property of edges, which was introduced in SPARQL 1.1. The ordering on edges can successfully prune isomorphic triangle patterns in advance since it converts all the possible triangle patterns into a single pattern: $x \rightarrow y, y \rightarrow z, x \rightarrow z$. Moreover, the initial query can reduce redundant computations which exist in the initial TC operation. In the processing step, the TC operation assembles three triple patterns that form triangles. The resulting query plans for the optimized TC operation are shown in Figure 4.1(b).
Figure 4.1. A Comparison of the Query Plans for TC Operations
Algorithm 4 Connected Components

function Initialize
  CREATE GRAPH $G_T$;
  INSERT { $G_T$ { ?node <label> ?node }}
  WHERE {
    ?node <edge> |\^{<edge>} ?neighbor
  }

function Process
  while !Converges() do
    DELETE { $G_T$ { ?s <counts> ?o }} WHERE { $G_T$ { ?s <counts> ?o }};
    DELETE { $G_T$ { ?node <label> ?previous }}
    INSERT { $G_T$ { ?node <label> ?update; <counts> 1 }}
    WHERE {
      $G_T$ {
        ?node <label> ?previous
      }
      }.
      { SELECT ?node (MIN(?label) AS ?update)
        WHERE {
          $G_T$ {
            ?neighbor <label> ?label
          }
          ?node <edge> |\^{<edge>} ?neighbor
        }
        GROUP BY ?node
      }
      FILTER(STR(?previous) > STR(?update))
    }

function Finalize
  DROP GRAPH $G_T$;
4.3.4 Connected Components (CC)

The algorithm for our CC operation performs \textit{min-label propagation} [106]. At initial time, each node \( n \) set \( \text{min}_\text{label}(n) = n \) and propagates its \( \text{min}_\text{label}(n) \) to the neighbors. When \( n \) receives \( \text{min}_\text{label}^* \) from its neighbors, it updates its \( \text{min}_\text{label}(n) = \min(\text{min}_\text{label}^*) \) and propagates the new \( \text{min}_\text{label}(n) \). This process is repeated until all nodes agree on termination. In our SPARQL-based approach, \( \text{min}_\text{labels} \) for all \( N \) nodes are initialized in a temporary named graph \( G_T \) as triples of \( <n_i,<\text{label}>n_i> \) \((1 \leq i \leq N)\). At computation step, an UPDATE command (DELETE followed by INSERT) retrieves the egocentric network \( G_{ego}(n_i) \) centered on \( n_i \) where nodes in \( G_{ego}(n_i) \) represent \( \text{min}_\text{label}^* \). Then, it calls an aggregation function \( \min(\text{min}_\text{label}^*) \) on \( G_{ego}(n_i) \) to calculate a new \( \text{min}_\text{label}(n_i)' \) and updates the corresponding labeling triple \( <n_i,<\text{label}>\text{min}_\text{label}(n_i)> \) in \( G_T \) as \( <n_i,<\text{label}>\text{min}_\text{label}(n_i)'> \). At the same time, the UPDATE command inserts into \( G_T \) a triple of \( <n_i,<\text{count}>1> \) for \( n_i \) whose \( \text{min}_\text{label}(n_i) \) has been updated. The UPDATE command is repeatedly processed until there is no \( <n_i,<\text{count}>1> \) in \( G_T \). This initial CC operation is shown in Algorithm 4.

![Query Plan](image)

(a) Query Plan from the Initial CC Operation

(b) Query Plan from the Optimized CC Operation

Figure 4.2. A Comparison of the Query Plans for CC Operations

Having analyzed the initial CC operation, we observed a notable problem on the underlying query processor in the current release of Jena, which can make a huge impact on the overall query processing performance. The observation is that when a binary JOIN operation has one input from a nested SELECT query with a GROUP BY or ORDER BY operation while the other input comes from a graph, the order of the two input sources represented in a SPARQL query

37
Table 4.1. Dataset Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Edges</th>
<th>Triangles</th>
<th>Max Degree</th>
<th>Input Node to NE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca-AstroPh (AS)</td>
<td>18,772</td>
<td>198,110</td>
<td>1,351,441</td>
<td>1,008</td>
<td>node:2444</td>
</tr>
<tr>
<td>com-DBLP (DB)</td>
<td>317,080</td>
<td>1,049,866</td>
<td>2,224,385</td>
<td>343</td>
<td>urn:0</td>
</tr>
<tr>
<td>soc-Epinion1 (EP)</td>
<td>75,879</td>
<td>508,837</td>
<td>1,624,481</td>
<td>3,079</td>
<td>node:0</td>
</tr>
</tbody>
</table>

decides the selection of the actual operation for the join. If the nested SELECT block appears before the GRAPH block, then the compiler replaces the JOIN operation with a SEQUENCE operation. Since the SEQUENCE operation does not generate a stage, it can minimize a large amount of in-memory intermediate data. Figure 4.2(a) shows the query plan generated from the UPDATE SPARQL query for the initial CC operation. The query plan contains two staging operations (JOIN and GROUP BY) which are expensive as discussed in Section 4.2.2. We switch the nested SELECT and the GRAPH blocks of the query: BlockA and BlockB so as to produce a more optimized query plan. The query plan of the optimized SPARQL query is shown in Figure 4.2(b). Compared to the initial query plan, the optimized plan contains just one staging operation (GROUP BY). This optimization also could improve the performance of the operation on several well-known triplestores. We discuss its impact in the following section.

4.4 Performance Evaluation

We tested the performance of the three graph mining operations and checked the effect of the optimizations. In this section, we explain the experiment setup and discuss the evaluation results.

4.4.1 Experiment Setup

We chose three datasets from Stanford Network Analysis Platform (SNAP) dataset collection [53]. The properties of the datasets are shown in Table 4.1. We converted those graph data into N-triples. For node eccentricity (NE) operation, we used the input nodes shown on the sixth column. The experiment was performed on a standalone workstation which has an i5-4252U 1.3GHz quad-core processor, 16 GB 1600 MHz DDR3 RAM, and a 256GB solid state disk. For triplestores, we selected three open-source implementations, Jena Fuseki 2.3.1 [45], Sesame 4.0.1 [82], and Parliament 2.7.9 [48]. We chose Jena in-memory store and Sesame MemoryStore as in-memory triplestores, and Jena TDB, Sesame NativeStore, and Parliament as disk-based triplestores since Parliament does not support in-memory processing. We did not consider triplestores that do not support insert/update/delete of individual triples such as RDF-3X [64]. For the selected triplestores, we set the maximum heap space for JVM with -Xmx14000M. We
only took processing time into account and set the cut-off time for query execution to two hours.

Figure 4.3. Execution Times on In-memory Triplestores

4.4.2 Experimental Results

Figure 4.3 and Figure 4.4 show the execution times of our graph mining operations on in-memory and disk-based triplestores, respectively. Among three operations, the NE operation took relatively less execution times on both in-memory and disk-based settings since it has $O(|E|)$ time complexity. Over all datasets, in-memory processing of the NE operation outperformed disk-based processing. Jena showed superior performance for the NE operation than
Figure 4.4. Execution Times on Disk-based Triplestores
Table 4.2. Numbers of Iterated Records in TC Operations

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Operation</th>
<th>Initialization</th>
<th>Computation</th>
<th>Group By</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca-AstroPh</td>
<td>Initial TC</td>
<td>0</td>
<td>52,859,332</td>
<td>10,811,528</td>
<td>63,670,860</td>
</tr>
<tr>
<td></td>
<td>Optimized</td>
<td>792,320</td>
<td>4,499,218</td>
<td>1,351,441</td>
<td>6,642,979</td>
</tr>
<tr>
<td>com-DBLP</td>
<td>Initial TC</td>
<td>0</td>
<td>25,245,229</td>
<td>2,224,385</td>
<td>27,469,614</td>
</tr>
<tr>
<td></td>
<td>Optimized</td>
<td>2,099,732</td>
<td>7,773,591</td>
<td>2,224,385</td>
<td>12,097,708</td>
</tr>
<tr>
<td>soc-Epinion1</td>
<td>Initial TC</td>
<td>0</td>
<td>113,557,432</td>
<td>4,326,715</td>
<td>117,884,147</td>
</tr>
<tr>
<td></td>
<td>Optimized</td>
<td>1,017,674</td>
<td>23,667,393</td>
<td>1,624,481</td>
<td>26,309,548</td>
</tr>
</tbody>
</table>

Sesame, and Parliament showed the best performance among disk-based triplestores. However, the Sesame NativeStore showed extremely low performance when the operation ran on it. For triangle counting, the optimization performed on the initial TC operation improved the performance 32% to 92% on in-memory triplestores and 6% to 88% on disk-based triplestores. While the initial TC operation was highly affected by $d_{max}$ of the datasets, the optimized counterpart showed execution times bounded to $O(|N| \cdot d_{max}^2)$ where $d_{max}$ is the maximum degree of each dataset. For the optimized TC operation, Sesame MemoryStore showed better performance than Jena SDB while the disk-based triplestores showed varying superiority according to datasets. The CC operation has $O(k \cdot |E|)$ time complexity and the execution times showed the same pattern according the properties of the datasets. $k$ means the number of iterations and its values were 9, 14, and 8 for ca-AstroPh, com-DBLP, and soc-Epinion1, respectively. For in-memory processing, the execution times of the initial CC operation were extremely high when it was executed on Jena SDB due to lots of intermediate data through the operation tree. On the other hand, Sesame showed much worse performance than other disk-based triplestores for both of the initial and the optimized operations. The optimization performed on the initial CC operation could improve the performance at least 30% on in-memory triplestores and at least 13% on disk-based ones.

**Comparison of the Initial and Optimized TC Operations** as discussed in the previous section, the advantage of the algorithmic optimization performed on the TC operation includes reduced computations and intermediate data. In order to check its effect, we counted the number of iterated records for operations in the query plans the Jena query processing engine produced. As shown in Table 4.2, the intermediate triples passed to the GROUP BY operation could be filtered out in advance for the two datasets: ca-AstroPh and soc-Epinion1. However, for com-DBLP, we found that no intermediate data were pruned before the operation, which minimized the benefit from the optimization. In addition, the optimized TC operation avoided duplicate computations and produced a query plan which did not contain a large SEQUENCE subtree as shown in Figure 4.1(b), which resulted in around 2 - 10 times less triple iterations through
the query plan.

**Comparison of the Initial and Optimized CC Operations** we analyzed the query plans the Jena query engine produced and found its inefficiency which a lot affects the performance. As shown in Figure 4.3(c) and Figure 4.4(c), the optimization greatly improved the performance of the CC operation on Jena SDB more than 98%. In disk-based processing, the optimized CC operation showed 52 - 63% performance improvement. Since in-memory processing is highly affected by available memory space and does not leverage index structures, it is more sensitive to the volume of intermediate records in query plans. Our optimization could reduce the intermediate data size by replacing an expensive JOIN operation with a less expensive counterpart. The result also confirms that the same problem exists on another triplestore, Sesame and Parliament. The optimization improved the performance up to 75% on Sesame MemoryStore, 37% on Sesame NativeStore, and 72% on Parliament. It is notable that Jena and Sesame are popular open-source triplestores and have many variations that use their query processing engines such as Parliament, uRiKA, uSeekM, and so on. Hence, we highlight the importance of this optimization for developing more extended graph mining operations on current triplestores.

### 4.5 Conclusion

In this work, we make an effort to extend triplestores to general graph analysis systems. For this, we introduced three graph mining written in SPARQL, and explain their optimization processes by analyzing query plans. The optimization process achieved 6% to 92% performance improvement for triangle counting and at least 13% better performance for connected components operation. The evaluation result showed that RDF and triplestores are suitable for complex native graph mining. We believe that our optimization work can be also leveraged for building a more extended set of graph analysis operations. We observed that current SPARQL query processing engines often do not produce optimal query plans. Therefore, users are still responsible for manually optimizing their operations by analyzing query plans.
Chapter 5

Spatial Clustering for RDF Graph Databases

5.1 Introduction

Graph clustering analysis is one of important graph mining tasks which is frequently used for finding connected components and communities in graphs [86, 111]. In most cases, graph clustering analysis considers structural properties of graphs. However, lots of recent graphs represented in RDF and property graphs have attributes associated to their entities and clustering them according to such attributes are highly recommended. In this chapter, we introduce a spatial clustering operation which can group entities in RDF graphs with their spatial and non-spatial attributes. The operation uses a recent standard established by Open Geospatial Consortium (OGC) called GeoSPARQL. It gives a way to assign spatial attributes to entities in RDF graphs and basic primitive functions required for spatial analysis. With increasing applications such as location-based services and widespread use of mobile devices, GeoSPARQL enlarges its importance in RDF [92, 29]. However, building a spatial clustering operation with GeoSPARQL is not straightforward especially because of expensive pairwise distance computations required in spatial clustering and limited spatial primitive functions the current version of GeoSPARQL provides. We also introduce a solution for avoiding such computations.

5.2 Background

In this section, we introduce the GeoSPARQL standard and spatial clustering.
5.2.1 GeoSPARQL

In spite of its popularity, RDF does not support spatial attributes on graph entities. As an extension to RDF, the GeoSPARQL standard was recently established by the Open Geospatial Consortium (OGC) [77]. It augments RDF with a vocabulary capable of representing spatial properties and spatial functions compatible to Simple Feature Access [37]. First, the vocabulary defines several RDF types. `ogc:Feature` identifies RDF resources that are associated to geometry objects. Each geometry object has a type of `ogc:Geometry`. RDF resources are associated with geometry objects by `geo:hasGeometry` edges. The GeoSPARQL standard currently supports two serialization types for geometries: Well-Known Text (WKT) and Geography Markup Language (GML). Geometries serialized in WKT and GML are related to geometry objects with `geo:asWKT` and `geo:asGML` edges, respectively. Figure 5.1(a) shows an example that three RDF resources (Object1-3) have geometry objects (Geometry1-3) whose geometries are serialized in WKT. The RDF triples form a graph as shown in Figure 5.1(b). Next, GeoSPARQL provides spatial functions in three classes. The first class includes functions that retrieve properties of geometries such as dimensions, types, and coordinate systems. Functions in the second class compute topological relationships between geometries such as `equals`, `disjoint`, `intersects`, `within`, `crosses`, and so on. The last class consists of functions for spatial operations such as calculating distances between geometries and computing buffers and convex hulls of geometries. GeoSPARQL functions are embedded into SPARQL queries and used like built-in operations as shown in Listing 5.1. The example query retrieves pairs of RDF resources whose geometries

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object1</td>
<td>geo:hasGeometry</td>
<td>Geometry1</td>
</tr>
<tr>
<td>Object2</td>
<td>geo:hasGeometry</td>
<td>Geometry2</td>
</tr>
<tr>
<td>Object3</td>
<td>geo:hasGeometry</td>
<td>Geometry3</td>
</tr>
<tr>
<td>Geometry1</td>
<td>geo:asWKT</td>
<td>POLYGON((24 35, 35, 25, 35, 34, 24, 24, 35))</td>
</tr>
<tr>
<td>Geometry2</td>
<td>geo:asWKT</td>
<td>POLYGON((23 34.6, 23.7 34.7, 23.2 34.1, 23 34.6))</td>
</tr>
<tr>
<td>Geometry3</td>
<td>geo:asWKT</td>
<td>POINT(23.4 34.4)</td>
</tr>
</tbody>
</table>

(a) Triples

(b) Graph Representation

Figure 5.1. A GeoSPARQL Example
are disjoint to each other along with their geometric distances.

5.2.2 Spatial Clustering and DBSCAN

Spatial clustering is frequently used in various domains such as hotspot detection, spatial epidemiology, and landscape ecology [61]. It aims at grouping similar spatial objects into a same cluster [62]. For this, several algorithms have been proposed each of which has its own properties suitable for particular applications. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [26] is a popular spatial clustering algorithm which takes the density of spatial objects into account for clustering and has multiple variations.

The DBSCAN algorithm has two user configurable parameters: $\epsilon$ and $MinPts$ which are used for defining spatial density. Using those parameters, it defines several terms as shown in Table 5.1. Algorithm 5 presents a simplified pseudo-code of the DBSCAN algorithm. For each input data point in $D$, it first checks if the point has been assigned to any cluster. If the point is not included in any cluster, then DBSCAN finds all density-reachable points from the point, assigning them including the pivot point to a new cluster. This process is repeated for all data points in $D$. At the end of computation, data points that have not been assigned to any clusters are marked as outliers. The time complexity of the algorithm is $O(|D|^2)$ and leveraging index structures can reduce it to $O(|D|\log|D|)$ [73]. The authors of the DBSCAN algorithm also proposed a generalized DBSCAN extension that can deal with not only spatial attributes but also non-spatial attributes [84]. There are parallel extensions of the algorithm [3, 73]. The parallel algorithm proposed in [3] builds an adjacency list from a graph and finds core points. In the following step, it performs a breadth-first search to cluster nodes in the graph. The algorithm is parallelized on GPUs using the CUDA library. The authors in [73] propose another parallel DBSCAN algorithm which scales over many-core clusters. The algorithm builds clusters by growing local trees on multiple cores independently and merging those trees at a minimized communication cost.
Table 5.1. DBSCAN Terms

<table>
<thead>
<tr>
<th>Terms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>directly density-reachable</td>
<td>A point ( x ) is directly density-reachable from a point ( y ) if ( \text{distance}(x, y) \leq \epsilon ) and (</td>
</tr>
<tr>
<td>density-reachable</td>
<td>A point ( x ) is density-reachable from a point ( y ) if there is a chain of points ( p_i (1 \leq i \leq n) ) such that ( p_1 = y, p_n = x ), and ( p_{j+1} ) is directly density-reachable from ( p_j ) ( (1 \leq j &lt; n) ).</td>
</tr>
<tr>
<td>core point</td>
<td>A point ( x ) is a core point if (</td>
</tr>
<tr>
<td>border point</td>
<td>A point ( x ) is a border point if it is density-reachable from any point ( y ) but ( x ) is not a core point.</td>
</tr>
<tr>
<td>outlier</td>
<td>A point ( x ) is an outlier if there is no point from which it is density-reachable.</td>
</tr>
</tbody>
</table>

Algorithm 5 DBSCAN

```plaintext
function DBSCAN(D, \( \epsilon \), MinPts)
    \( C \leftarrow \{\} \)
    \( c \leftarrow 1 \)
    for \( i = 1 \) do n
        if \( d_i \in D \) is not in any cluster then
            \( N \leftarrow \text{DensityReachablePoints}(d_i, \epsilon, \text{MinPts}) \)
            if \( |N| \geq \text{MinPts} \) then
                \( C_c \leftarrow N \)
                \( c \leftarrow c + 1 \)
```

46
5.3 Graph-oriented DBSCAN (G-DBSCAN)

In this section, we first discuss several challenges on implementing the DBSCAN algorithm in GeoSPARQL. Then, we introduce our G-DBSCAN algorithm.

5.3.1 Challenges

GeoSPARQL provides a seamless extension for representing and analyzing spatial attributes over RDF graphs. However, developing spatial clustering operations in GeoSPARQL is not straightforward. First, the original DBSCAN algorithm presented in [84] is sequential and fits well into general programming interfaces. The parallelized algorithms in [3, 73] also rely on general programming interfaces along with optimized libraries (e.g., CUDA and MPI) including a variety of functions. However, SPARQL only supports set-oriented primitive operations and, by itself, does not provide a way to describe sequential and iterative algorithms. Next, triplestores are more optimized to deal with graph-oriented problems, especially graph pattern matching. Hence, any implementations in non-graph-centric approaches do not guarantee optimal analysis performance.

5.3.2 Revisision of the DBSCAN Analysis Problem

Figure 5.2. DBSCAN (\(\epsilon = d\) and \(MinPts = 3\))
In order to develop a spatial clustering operation in RDF, we revisit the original DBSCAN problem to convert it into a more graph-oriented counterpart. The definition of ‘density-reachable’ ensures that any two border points in a same cluster are connected with a chain of one or more core points while any two border points in different clusters do not have such chains. For example, in Figure 5.2(a), A and I are connected to each other via a chain of core points C, E, and G. Likewise, J and L are connected via a core point K. This also means that all core points in a cluster are connected and if two core points are connected, the connection is symmetric. Outliers have no core points within their \( \epsilon \)-distance neighbors. By using this property, we re-define the DBSCAN problem as:

**Definition 6** Given a dataset \( D \), G-DBSCAN first converts \( D \) into a directed graph \( G(V, E) \) where \( V \) is a set of data points in \( D \) and \( E \) is a set of “directly density-reachable” edges. Then, it finds all connected components, \( C_i \) (\( 0 \leq i \leq n \)) where any two nodes \( a \in C_i \) and \( b \in C_i \) are connected through a chain of greater than or equal to one edge while any two nodes \( f \in C_i \) and \( g \in C_j \) (\( 0 \leq j \leq n \) and \( i \neq j \)) are disconnected. If a node \( c \in V \) is not included in any \( C_i \), it is an outlier.

Figure 5.2(b) shows a new DBSCAN problem based on 6 converted from the original problem Figure 5.2(a). It describes that core and border points are nodes and connected with each other with edges of “directly density-reachable”. In the following section, we explain our algorithm in detail.

### 5.3.3 G-DBSCAN Algorithm

Algorithm 6 shows the pseudo-code of G-DBSCAN. The G-DBSCAN algorithm consists of three phases: initialization, processing, and finalization. In each phase, we devise multiple GeoSPARQL queries that are optimized especially to reduce intermediate data. Those queries are embedded into a Python program so that they are sequentially executed. Note that the Python program is just a skeleton by which required queries are issued in their correct order and the execution of the algorithm is controlled. Hence, it does not maintain any intermediate state during the execution of the algorithm, allowing any programming languages to be used. Python was selected here as we are developing a plugin for the popular open-source software, QGIS [80].

**Initialization** first, the algorithm retrieves interesting triples by applying a set of predicates \( \Theta \). Those predicates are used to select interesting triples from a large dataset and can consist of temporal, spatial, and other conditional filters that can be described in SPARQL and GeoSPARQL primitives. For example, users are only interested in point geometries representing gas stations placed in the city of Cary created in 2015. All those conditions can be represented
Algorithm 6 G-DBSCAN Algorithm

\textbf{function} G-DBSCAN(D, \epsilon, \text{MinPts}, \Theta)
\begin{align*}
S & \leftarrow \{\} \\
\Delta & \leftarrow \{\} \\
\text{INITIALIZATION}(D, S, \Delta, \epsilon, \Theta) & \\
\text{PROCESSING}(D, S, \Delta, \text{MinPts}) & \\
\text{FINALIZATION}(S)
\end{align*}

\textbf{function} \text{INITIALIZATION}(D, S, \Delta, \epsilon, \Theta)
\begin{align*}
S & \leftarrow \text{Preprocess}(D, \Theta) \\
\Delta & \leftarrow \text{ComputeDistance}(S, \epsilon)
\end{align*}

\textbf{function} \text{PROCESSING}(D, S, \Delta, \text{MinPts})
\begin{align*}
T & \leftarrow \{\} \\
L & \leftarrow \{\} \\
S & \leftarrow S \cup \text{ComputeCorePoints}(S, \Delta, \text{MinPts}) \\
T & \leftarrow \text{GenerateGraph}(S, \Delta) \\
L & \leftarrow \text{ConnectedComponents}(T) \\
\text{VisualizeClusters}(L, S, D)
\end{align*}

\textbf{function} \text{FINALIZATION}(S)
\begin{align*}
\text{RemoveState}(S)
\end{align*}

\textbf{function} \text{ConnectedComponents}(T)
\begin{align*}
L & \leftarrow \{\} \\
\text{Initialize}(T, L) \\
\text{while } \text{NotConverge()} \text{ do} \\
& \hspace{1cm} \text{PropagateLabels()} \\
\text{Finalize}(L) \\
\text{return } L
\end{align*}
CREATE GRAPH {S};
// For Spatial Attribute (WKT geometry)
INSERT { GRAPH {S} { ?node temp:center ?wkt }}
WHERE {
  ?node geo:hasGeometry ?geometry .
  ?geometry geo:asWKT ?wkt .
  // Θ can be added here
};
// For Non-spatial Attribute (population)
INSERT { GRAPH <state> { ?node temp:center ?p }}
WHERE {
  ?node geo:hasGeometry ?geometry .
  ?node { prefix}:population ?population .
  // Θ can be added here
  BIND ( xsd:integer(?population) AS ?p)
};

Listing 5.2. Pre-processing Query

in Θ. This phase further reduces input data by pre-processing triples. For example, the triples
<Object3, geo:hasGeometry, Geometry3> and <Geometry3, geo:asWKT, POINT(23.4 34.4)> in Figure 5.1(a) can be merged into <Object3, geometry, POINT(23.4 34.4)>, which reduces the number of triples fed into the G-SPARQL operation. Listing 5.2 shows the GeoSPARQL queries for pre-processing. Note that G-DBSCAN can use non-spatial attributes such as population and area for clustering. Next, G-DBSCAN computes pairwise distances between spatial objects by performing the query in Listing 5.3 and stores computed distances into a temporary graph Δ. The query performs a Cartesian product for computing all pair distances in S which can produce a large volume of intermediate data. In order to minimize such intermediate data, we prune spatial object pairs whose left objects’ URIs are lexicographically greater than right ones. Since we use a symmetric distance measure, we reproduce all the distances when they are inserted into Δ. Second, object pairs whose distances are greater than ϵ are pruned since they are not ultimately connected by “directly density-reachable” edges.

**Processing** in this phase, COMPUTECOREPOINTS(S, Δ, MinPts) finds core points from Δ and marks them in S as core points by adding <object-uri, cluster_label: object-uri> triples. The corresponding query is shown in Listing 5.4. It finds ϵ-distance neighbors for each spatial object and performs a group-by operation to count the number of those neighbors. If the count is greater than or equal to MinPts, the object is considered as a core. Then, CREATEGRAPH(S, Δ) generates a graph T from core points in S and distances in Δ which consists of spatial objects as nodes and their “directly density-reachable” relationship as edges as shown in Figure 5.2(b). For this, the query in Listing 5.5 is processed. T is passed to the connected components (CC) operation for producing clusters. We use the CC implementation proposed in Chapter 4. Spatial objects involved in any clusters are stored in a temporary graph L. In the end of this phase,
// For Spatial Attribute
INSERT { GRAPH {Δ} { ?s1 ?s2 ?dist . ?s2 ?s1 ?dist }}
WHERE {
  GRAPH {S} {
    ?s1 temp:center ?g1 .
    ?s2 temp:center ?g2 .
    FILTER (STR(?s1) <= STR(?s2)) .
    BIND (geof:distance(?g1, ?g2, units:metre) AS ?dist) .
    FILTER (?dist <= ϵ)
  }
}
// For Non-spatial Attribute
INSERT { GRAPH {Δ} { ?s1 ?s2 ?dist . ?s2 ?s1 ?dist }}
WHERE {
  GRAPH {S} {
    ?s1 temp:center ?g1 .
    ?s2 temp:center ?g2 .
    FILTER (STR(?s1) <= STR(?s2)) .
    BIND (ABS(?g2 - ?g1) AS ?dist) .
    FILTER (?dist <= ϵ)
  }
}

Listing 5.3. Distance Computation Query

INSERT { GRAPH {S} { ?g1 temp:cluster_label ?g1 }}
WHERE {
  SELECT ?g1 (COUNT(?g2) AS ?pts)
  WHERE {
    GRAPH {Δ} {
      ?g1 ?g2 ?dist .
      FILTER (?g1 != ?g2)
    }
  }
  GROUP BY ?center1
  HAVING (?pts >= MinPts)
}

Listing 5.4. Core Points Retrieval Query

INSERT { GRAPH {T} { ?s1 temp:link ?s2 }}
WHERE {
  GRAPH {S} {
    ?s1 temp:cluster_label ?label .
  }.
  GRAPH {Δ} {
    ?s1 ?s2 ?dist
  }.
  FILTER (STR(?s1) != STR(?s2))
}

Listing 5.5. Graph Creation Query
SELECT ?wkt ?label
WHERE {
    SELECT ?wkt ?label
    WHERE {
        GRAPH {S} {
            ?node temp:center ?center
        }.
        ?node geo:hasGeometry ?geometry .
        ?geometry geo:asWKT ?wkt .
        GRAPH {L} {
            ?node <temp:labels> ?label
        }
    }
    UNION
    {
        SELECT ?wkt ('OUTLIER' AS ?label)
        WHERE {
            GRAPH {S} {
                ?node temp:center ?center
            }.
            ?node geo:hasGeometry ?geometry .
            ?geometry geo:asWKT ?wkt .
            FILTER (! EXISTS { GRAPH {L} {
                ?node <temp:labels> []
            })))
        }
    }
}

Listing 5.6. Cluster Retrieval Query

VisualizeClusters(L, S, D) retrieves and presents spatial objects in a graphical way. Outliers are detected by finding spatial objects that do not exist in L but S. The corresponding query is shown in Listing 5.6.

Finalization in this phase, all intermediate states previously generated are removed by simply destroying intermediate graphs such as S and L.

5.3.4 Limitations

The proposed G-DBSCAN operation is suitable for the RDF data model and able to be expressed with the set-based primitive operations provided by SPARQL and GeoSPARQL. The operation can be solely described in SPARQL/GeoSPARQL, which can run on any GeoSPARQL-compatible triplestores. Regardless of such advantages, we observed some limitations that restrict its performance. First, the pairwise distance computation significantly affects clustering performance since it performs an expensive Cartesian product operation Whose time complexity is $O(n^2)$ where $n$ is the number of spatial objects in S. Another problem of the operation
is that it produces a lot of intermediate records even though most of them are pruned by the \( \epsilon \)-distance condition. Existing GeoSPARQL triplestores internally manage spatial objects leveraging spatial indexes such as R-tree and KD-tree. Those indexes can be used for avoiding the expensive Cartesian product operation. However, the current version of GeoSPARQL does not have any index-aware primitive operations that can be used for avoiding Cartesian product. Next, intermediate graphs used for maintaining states are eventually discarded in the end and, therefore, keeping them in memory can give an opportunity to reduce unnecessary I/O and transaction overheads. However, current GeoSPARQL-compatible triplestores do not support in-memory repositories that can be selectively used for computation.

5.3.5 Index-based Distance Computation

We extend the G-DBSCAN algorithm in order to overcome the limitations previously discussed by leveraging offloaded and multi-threaded distance computation.

**Offload Distance Computation** we overcome the above mentioned limitations by offloading distance computation in a separate process. Figure 5.3 describes this process. First, it retrieves spatial objects from the preprocessed graph \( S \) and indexes them in an in-memory R-tree on the fly. Then, it iteratively retrieves \( \epsilon \)-distance neighbors for each object by using the index. In each iteration, it computes the distances between the object and its neighbors, pruning any objects whose distances are greater than \( \epsilon \). Remaining spatial objects are serialized in an N-triple file as triples in a form of \(<\text{source}_\text{object}, \text{neighbor}_\text{object}, \text{distance}>\). In the end, the file is bulk-loaded into the triplestore as a temporary graph \( \Delta \).

![Figure 5.3. Offloaded Pairwise Distance Computation](image)

This approach has several advantages. First, it can avoid expensive Cartesian product by
using spatial index, which reduces computational overhead and intermediate data size. Second, it can leverage in-memory storage for managing intermediate states, which avoids materialization and transaction overheads. Last, \texttt{geof:distance()} is the only GeoSPARQL function used in G-DBSCAN. By offloading distance computation, it is not necessary to rely on the primitive, which allows use of other available libraries. Moreover, G-DBSCAN can be applied to any existing triplestores.

**Distance Computation Process**

- R-tree Index (Geometries)
- Thread1 ThreadN...
- Spatial Object List
- Per-thread Files
- Merged N-triple File

![Distance Computation Process](image)

Figure 5.4. Multithreaded Spatial Searching

**Concurrent $\epsilon$-neighbor Searching** we further optimize the distance computation by using multithreaded spatial search as shown in Figure 5.4. For this, spatial objects retrieved from a triplestore are stored in a list. Those objects are equally assigned to multiple threads. Each thread iterates over its spatial objects, searching neighbors and computing their distances. Generated \texttt{<source_object, neighbor_object, distance>} triples are stored in a per-thread temporary file. When all the threads complete their computation, the main process concatenates temporary files into a single N-triple file.
Table 5.2. Experiment Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Triples</th>
<th>#Geometries</th>
<th>#Clustered Geometries</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greek Administrative Geography</td>
<td>4,091</td>
<td>325</td>
<td>325</td>
<td>MULTIPOLYGON</td>
</tr>
<tr>
<td>Geonames</td>
<td>396,610</td>
<td>21,990</td>
<td>3,144</td>
<td>POINT</td>
</tr>
</tbody>
</table>

5.4 Experimental Evaluation

We evaluated the G-DBSCAN operation along with the optimization in order to check its clustering result and performance. In this section, we present the experimental environment and methodology, and evaluation results.

5.4.1 Experiment Setup

We evaluated the G-DBSCAN operation on a standalone server which has two Intel Xeon 2.4GHz E5-2630 processors each of which has eight hardware cores, 125GB RAM, and a 512GB SSD. We selected Parliament 2.7.10 [87] and GraphDB 8.0.2 free edition [70] as GeoSPARQL-enabled triplestores. Note that the optimized G-DBSCAN operation does not rely on GeoSPARQL primitives and, therefore, we evaluated the operation on Jena Fuseki 3.6.0 [5] as a general triple-store. For those triplestores, the maximum heap size for JVM was configured to 64GB. We used a crime dataset from the city of Raleigh, North Carolina provided by the US City Open Data Census [16]. It contains the crime information occurred from Jan 1st, 2005 where each occurrence is represented as a point geometry. The original dataset was converted into an RDF dataset and serialized into a N-triple file. There are 2,795,899 triples in the dataset whose size is 295.9MB. In this evaluation, we used a temporal predicate which retrieves crime data reported in 2005 in the initialization phase, which consist of 29,989 geometries. We additionally used Greek Administrative Geography (GAG) and Geonames from Geographica, a GeoSPARQL benchmark suite [28] to test clustering with non-spatial attributes. Table 5.2 provides detailed information about the datasets.

5.4.2 Evaluation Result

Performance Comparisons we compared the initial and the optimized G-DBSCAN operations in terms of execution time. For this experiment, we used three different parameter settings of \(\epsilon\) and \(MinPts\): \((\epsilon = 100 \text{ meters}, MinPts = 3)\), \((\epsilon = 200 \text{ meters}, MinPts = 3)\), and \((\epsilon = 200 \text{ meters}, MinPts = 7)\). For the optimized operation, we used 10 threads for distance computations. Figure 5.5 shows the execution times of the two operations. First, comparing the two operations, the optimized operation outperformed the initial operation. On Parliament, the optimized operation achieved about 39, 10, and 19 times speedups for the three settings, respec-
tively. The initial operation performs an expensive Cartesian product for computing pairwise distances, which contributed to around 91-97% of the execution times. On GraphDB, pairwise distance computations took an amount of time and we interrupted the initial operation after 10 hours. The optimized operation showed at least 87, 25, and 45 times speedups for the three parameter settings, respectively. The significance of the offloading distance computation are mainly two folds. First, the optimization creates an in-memory R-tree index on-the-fly and leverages it for computing pair-to-pair distances. It could reduce the time complexity of the distance computation from $O(n^2)$ to $O(n\log n)$, which made a large impact on the performance. Second, the Cartesian product in the initial distance computation query produces a large amount of intermediate data, which significantly affects the query processing performance [108]. On the other hand, the optimization avoids such an expensive operation and minimizes intermediate data by using a spatial index. Second, the initial operation relies on a GeoSPARQL primitive which computes distance between spatial objects. On the other hand, the optimized operation performs such distance computations out of triplestores, which makes the operation independent of the GeoSPARQL standard. One advantage of this approach is that the operation can run on any triplestores even though they are not GeoSPARQL-compatible. In Figure 5.5, we evaluated the optimized operation on Apache Jena. In our experiment, Jena showed around 1.35 - 1.75 times better performance than the two GeoSPARQL-compatible triplestores. We observed that Jena updated triples in the connected components operation faster than the others, which resulted in about 1.72 - 1.79 times faster the connected components operation on Parliament and 1.18 - 1.29 faster on GraphDB. Since the optimized operation could run on any triplestores, it can leverage recent high-performance triplestores such as uRiKA-GD [19] and uRiKA-GX [20].
Analysis of Connected Components  different parameter settings produce varying number of graph entities in input graphs to connected components operation. Such variations affect the performance of the operation. We analyzed the impact of different parameter settings on the operation. Table 5.3 presents the number of graph entities for each parameter setting along with the number of iterations for label propagation in connected components operation. In our setting, an increase in ε especially resulted in far more number of “directly density-reachable” edges along with a steep increase in number of iterations. On the other hand, an increase in MinPts did not reduce the number of edges a lot but significantly reduce the number of iterations. Those changes affected the performance of the connected components operation as shown in Figure 5.6. The time complexity of the connected components is a function of number of iterations and number of edges. We checked that the execution time of the operation exactly matches its time complexity.

Multi-threaded Distance Computation  we evaluated the impact of multi-threading for searching ε-neighbors in the optimized operation. In this evaluation, we measured the execution time of ε-neighbor searching on GraphDB by increasing the number of threads from 1 to 8 for three parameter settings where ε=200m, 400m, and 600m and MinPts=3. Note that we separate the distance computation process into two parts. The first builds an in-memory spatial index, computes ε-neighbor for all nodes by using the index, and merges partial output files.
The second part uploads the merged file into triplestores. We measured the execution time of the first part, which is shown in Figure 5.7. We observed that \( \varepsilon \)-neighbor searching benefits from multi-threading. In our setting, multiple threads showed 1.38-2.05, 1.56-3.10, and 1.64-3.83 times better performance than single-thread processing for \( \varepsilon = 200 \text{m}, 400 \text{m}, \) and \( 600 \text{m}, \) respectively. However, performance improvement tended to be saturated as more threads were involved since (1) \( \varepsilon \)-neighbor searching requires disk I/Os for storing partial outputs generated by threads and (2) it builds R-tree index sequentially rather than in parallel. We observed that R-tree construction part took about 1.5 seconds for the three parameter settings.
Figure 5.8. Clustering Result
(a) $\epsilon=100$, $MinPts=3$

(b) $\epsilon=200$, $MinPts=3$

(c) $\epsilon=300$, $MinPts=3$

(d) $\epsilon=100$, $MinPts=5$
(e) $\epsilon=100$, $MinPts=10$

(f) $\epsilon=100$, $MinPts=15$
Clustering Result  we compared clusters resulted from different parameter settings. First, Figures 5.8(a) to 5.8(c) show the clusters with different $\epsilon$ settings (100m, 200m, and 300m). Note that each color represents a cluster and black stars represent outliers. In G-DBSCAN, a longer $\epsilon$ distance increases not only the number of core points but also the number of density-reachable points from each data point with a same $MinPts$. Accordingly, those figures show that an increase on $\epsilon$ resulted in less outliers (note that black starts indicate outliers). Such an increase also produced fewer clusters where nearest clusters became merged as shown in the highlighted area. Next, Figures 5.8(d) to 5.8(f) describe how clusters change with different $MinPts$ settings (5, 10, 15). An increase on $MinPts$ makes the conditions of “directly density-reachability” and “core points” more strict, which, in turn, affects “density-reachability” and “outliers”. As a result, such an increase resulted that more data points were classified as outliers. It also divided a large cluster into a few smaller ones as highlighted in the zoomed region.

Clustering with Non-spatial Attribute  Figure 5.9 and Figure 5.10 show the clusters generated from two Geographica datasets, Greek Administrative Geography and Geonames, respectively, by using their population property as a clustering attribute. Regions of similar population sizes were grouped in a same cluster. Different from $K$-means and $K$-medoids clustering techniques, G-DBSCAN does not need to determine the exact number of clusters and automatically decides the optimal clusters based on the two parameters. It is also observed that different $\epsilon$ and $MinPts$ settings affect the results (e.g., the number of clusters and outliers).

5.5  Conclusion

In this chapter, we presented a graph mining operation that clusters graph entities by their spatial and non-spatial attributes. The delivered operation re-defines the density-based clustering problem into a graph-oriented problem. Then, it clusters graph entities by using connected components operation. In this work, we observed some limitations on the current GeoSPARQL standard and its compatible triplestores and proposed a solution which offloads expensive distance computation. It leverages in-memory processing, spatial index, and concurrent distance computation, which resulted in around 12 to 68 times better performance than the pure GeoSPARQL-based operation.
Figure 5.9. Clusters with A Non-spatial Attribute: Greek Administrative Geography
Figure 5.10. Clusters with A Non-spatial Attribute: Geonames
Chapter 6

Structural Clustering for Community Detection

6.1 Introduction

Real-world graphs are often scale-free [99] and their edge distribution is locally and globally inhomogeneous [27]. Those properties result in so called communities or clusters of nodes where high intra-connections within each community and low inter-connections between communities are observed. Finding such groups has a plethora of its applications, for example, grouping social entities who share similar interests [67], and identifying modules [47] and deriving phylogenetic information [4] from biological networks. In such a myriad of applications, detecting quality communities from graphs in an efficient manner is a core expectation of community detection algorithms.

As listed in a survey paper [27], there have been lots of community detection algorithms in different classes with their own features and applications. Many traditional algorithms partition graphs so that they can find communities that minimize inter-community edges [89, 27]. After defining the notion of modularity [66], several approaches have been developed to maximize modularity [65, 11, 103]. One of recent algorithms is Structural Clustering Algorithm for Network (SCAN) [105]. The algorithm borrows an idea from DBSCAN [26] in order to cluster graph entities with its own distance measure called structural similarity. Hence, it inherits the virtues of DBSCAN in that it can identify communities without any prior-knowledge about their structures and exceptional graph entities which connect communities or depart from them. However, the algorithm has a certain limitation that it does not exhibit good performance on recent large scale graphs [18, 90, 17]. Several MapReduce-based parallel extensions [18, 111] have been proposed. However, those approaches do not show optimal performance of parallel processing since they require multiple MapReduce jobs, resulting in recurring I/O and commu-
In this chapter, we introduce a parallel SCAN algorithm built atop a recent high-performance graph analysis framework called CombBLAS [15]. The algorithm is developed with parallelized matrix and vector operations of the framework. We also propose an optimization for (1) avoiding expensive matrix-matrix multiplication operation of CombBLAS and (2) reusing expensive structural similarity computation. The optimization computes structural similarity in a preprocessing phase by using hybrid parallel programming.

6.2 Related Work

Structural Clustering Algorithm for Network (SCAN) [105] provides additional features that can not only cluster graph entities without prior knowledge but also identify exceptional nodes such as hubs and outliers. In spite of such advantages, however, a limitation has been reported that it does not show optimal performance with large-scale graphs [18, 90, 17]. The limitation is mainly because of the high time complexity of computing structural similarity of node pairs. Recent optimizations [90, 17] could achieve far better performance of structural community detection by reducing the number of similarity computations. The optimization in SCAN++ [90] leverages a property of real-world graphs that a node and its two-hop-away nodes are likely to have a number of shared neighbors, and avoids structural similarity computations among them. The optimization proposed in pSCAN [17] more improved the community detection performance by early pruning of unnecessary similarity computations. However, those approaches still suffer from expensive structural similarity computations proportional to the scale of graphs. Moreover, pruned similarity computations depends on the two user parameters. In real-world scenarios, data scientists generally investigate communities with different settings of those parameters in order to understand implicit community structures in graphs. Unlike those approaches, we compute similarity for all pairs of connected pairs of nodes for reusing the computations. There have been parallel extensions of the SCAN algorithm to improve its performance [18, 111]. They leverage the MapReduce framework for processing large graphs efficiently. However, they requires multiple MapReduce jobs which result in incurring intermediate data materialization and communication overheads, affecting overall performance significantly as more jobs are processed. Different from those approaches, we developed a parallel scan algorithm based on hybrid parallel programming with multi-processing and multi-threading.

6.3 Background

This section presents detailed description about SCAN, the community detection algorithm we consider, and CombBLAS, a high performance graph analysis library we leverage for construct-
ing a parallel algorithm.

6.3.1 SCAN

Structural Clustering Algorithm for Network (SCAN) [105] is a graph community detection algorithm which borrows an idea from Density-based Algorithm for Discovering Clusters in Large Spatial Database with Noise (DBSCAN) [26]. While DBSCAN aims at clustering spatial objects, SCAN deals with graphs. Given a graph $G = \{V,E\}$, the algorithm defines the structure of node $u \in V$ as

$$\Gamma(u) = \{v \in V|(u,v) \in E\} \cup \{u\}$$

and uses the following structural similarity as its distance measure:

$$\sigma(u,v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{|\Gamma(u)||\Gamma(v)|}}$$

The algorithm receives two user-supplying parameters: $\mu$ and $\epsilon$ and defines a set of core terms with them. First, it defines $\epsilon$-neighborhood of node $u$ as

$$N_\epsilon(u) = \{v \in \Gamma(u)|\sigma(u,v) \geq \epsilon\}$$

which means all the 1-hop neighbors of $u$ whose structural similarity is greater than or equal to $\epsilon$. If a node has at least $\mu \epsilon$-neighbors, it is regarded as a core. If node $v$ meets the core condition, then it is said that $u$ is directly structure reachable from $v$ for node $u \in N_\epsilon(v)$. If node $u$ is reachable from any core point $v$ through a chain of cores, then $u$ is structure reachable from $v$. If node $u$ is not a core but direct structure reachable from any other node, it is called a border.

The algorithm starts by checking that each unclustered node in $G$ is a core. If a node is core, then it assigns a new cluster label to it and its $\epsilon$-neighbors, and expand the cluster by repeatedly finding directly structure reachable nodes from those neighbors and assigning the cluster label. In the end of the algorithm, any nodes that do not have cluster labels are further classified as hubs and outliers. If unclassified node $u$ is connected to more than or equal to two clusters, it is regarded as a hub, otherwise, an outlier. As discussed in [90], structural similarity computation is the most expensive operation in the SCAN algorithm. It computes structural similarity by iterating over edges in $G$ and counting the number of shared neighbors per edge, which results in time complexity of $O(|E|\text{min}(|\Gamma(u)|,|\Gamma(v)|))$. Since the average degree is $|E|/|V|$, the average time complexity is $O(|E|^2/|V|)$. Since we assume no multi-edges, the maximum degree of $G$ is $|V|$ when $G$ is a complete graph. Hence, the worst time complexity is $O(|V|^3)$. 67
### 6.3.2 CombBLAS

CombBLAS, Combinatorial BLAS, is a high performance software framework for graph analysis and mining [15]. It is frequently compared with modern graph frameworks for checking their performance and scalability [93, 74]. Among several recent graph libraries, we select it because (1) a full set of matrix primitives for developing our community detection algorithm, (2) its high performance and scalability over multi-core computers to shared-nothing clusters, (3) continuous updates, and (4) its open source license. The framework offers a set of data structures for matrices and vectors along with optimized linear algebra primitives. The data structures separately represent matrices and vectors according to their sparsity. Matrices and vectors are distributed across 2D and 1D grids of computing nodes, respectively. The primitives contain not only basic BLAS operations but also additional operations that act on their rows/columns and elements. Being aware of the distribution of data structures, each operation is optimized by using MPI primitives and multi-threading. We list a minimum set of primitives which are required for our algorithm. Note that we use more compact primitive names, which are different from their original names yet they are one-to-one mapped to each other.

**Primitives for Creating Data Structures**  
Load(path) loads a graph located by path into a distributed sparse matrix. It supports both a text format in Matrix Market Exchange (MTX) [68] and a binary format as defined in [6]. Copy(M or v) produces the same in-memory data structure from a given matrix or vector. SpParMat(r, c, v) generates a distributed sparse matrix from three vectors which contain row indexes, and column indexes, and element values. Ones(length) creates a vector of size length whose elements are set to one. Range(length, start) creates a vector of size length and set elements to numbers starting from start value and increasing by one.

**Primitives for Matrix and Vector Computations**  
SpGEMM(M1, M2) multiplies two distributed sparse matrices, M1 and M2. SpMV(M, v, f_add, f_mul) performs a matrix-vector multiplication. Two additional functions can be supplied each of which re-defines element-wise addition and multiplication. Add(M1, M2) performs element-wise additions of two input matrices. Set(v1, v2) set elements in vector v1 with corresponding non-zero elements in v2. EApply(M1, M2 or v1, v2, p, f) applies a lambda function f on any pairs of elements in M1 and M2 or v1 and v2 that satisfy a given predicate p. DApply(M, v, ROW or COL, f) applies a lambda function f on pairs of elements in each row or column of M and v.

**Primitives for Manipulation and Aggregation**  
Transpose(M) computes $M^T$ from a matrix M. RemoveDiagonal(M) and AddLoops(M) remove elements and set ones on the main diagonal of a matrix M, respectively. Prune(M or v, p) removes all the elements that meet a
predicate condition \( p \). \( \text{Apply}(M \text{ or } v, f) \) applies a lambda function \( f \) to each element in \( M \) or \( v \). \( \text{Reduce}(M, \text{ROW} \text{ or } \text{COL}, f, i) \) accumulates elements in \( M \) along each row or column. A lambda function \( f \) is supplied which has two arguments: each element value and an aggregated value. The aggregated value is initialized with \( i \). \( \text{NNZ}(v) \) counts the number of non-zero values in \( v \). \( \text{NumberOfRows}(M) \) and \( \text{NumberOfColumns}(M) \) return the number of rows and columns in \( M \), respectively.

The CombBLAS framework is written in C++. Its Python wrappers are available in \textit{Knowledge Discovery Toolbox (KDT)} [58]. At this point of time, the latest CombBLAS version is 1.6 and the latest KDT is 0.3 which is based on CombBLAS version 1.2.

### 6.4 Parallel SCAN

#### 6.4.1 Algorithm

We developed a parallel SCAN algorithm atop CombBLAS. Note that our algorithm is also built around other BLAS frameworks that provide the primitives listed in the previous section. The parallel SCAN algorithm is present in Algorithm 7. To summarize, the algorithm loads graph \( G \), computes structural similarity for all the edges in \( G \), finds core nodes and creates a new graph \( C \) whose edges represent direct structural reachability from the core nodes, and performs connected components operation to cluster nodes in \( C \). In the end, it classifies borders, hubs, and outliers.

**Algorithm 7** Parallel SCAN

```
1: function \( \text{SCAN}(\text{path}, \epsilon, \mu) \)
2: \( G \leftarrow \text{GRAPHLOAD}(\text{path}) \) \hfill \text{\( G \): input matrix}
3: \( D \leftarrow \text{COMPUTEEPSILONDISTANCE}(G, \epsilon) \) \hfill \text{\( D \): distance matrix}
4: \( (C, c) \leftarrow \text{FINDCORES}(D, \mu) \) \hfill \text{\( C \): core matrix, \( c \): core vector}
5: \( l \leftarrow \text{CONNECTEDCOMPONENTS}(C) \) \hfill \text{\( l \): component labels}
6: \( \text{FINDHUBSANDOUTLIERS}(G, D, c, l) \)
```

**Algorithm 8** Graph Loading

```
1: function \( \text{GRAPHLOAD}(\text{path}) \)
2: \( G \leftarrow \text{Load}(\text{path}) \)
3: \( G \leftarrow \text{ADD}(G, \text{Transpose}(G)) \)
4: return \( G \)
```
GraphLoad in Algorithm 8 reads an input graph and creates a distributed sparse matrix $G$ for the graph. We leverage a parallel I/O feature in CombBLAS to minimize I/O overhead. It adds $G$ and $G^T$ to generate edges in both directions.

**Algorithm 9** Computing $\epsilon$ Distance

1: function COMPUTE EPSILON DISTANCE($G, \epsilon$)  
2:   $A \leftarrow$ Copy($G$)  
3:   $B \leftarrow$ Copy($G$)  
4:   $D \leftarrow$ SpGEMM($G, G$)  
5:   $D \leftarrow$ RemoveDiagonal($D$)  
6:   $d \leftarrow$ Reduce($G$, COL, $f : e_G + \text{sum, 0}$)  
7:   $A \leftarrow$ DApply($A, d$, ROW, $f : e_A + e_d$)  
8:   $B \leftarrow$ DApply($B, d$, COL, $f : e_B + e_d$)  
9:   $A \leftarrow$ EApply($A, B, p : e_A \neq \emptyset \land e_B \neq \emptyset, f : \sqrt{e_A \times e_B}$)  
10:  $D \leftarrow$ EApply($D, A, p : e_D \neq \emptyset \land e_A \neq \emptyset, f : e_D / e_A$)  
11:  $D \leftarrow$ Prune($D, p : e_D < \epsilon$)  
12:  return $D$

COMPUTE EPSILON DISTANCE($G, \epsilon$) in Algorithm 9 computes structural similarity for every edge in $G$ with a user-defined $\epsilon$ parameter. $|\Gamma(v) \cap \Gamma(w)|$ of the structural similarity formula is equal to the number of shared 1-hop neighbors of $v$ and $w$ plus two, and it can be computed by counting elements in the intersection of $v$th row and $w$th row in $G$. Since CombBLAS currently does not support row-wise intersection, we compute the number of neighbors of all pairs of nodes by using SpGEMM. However, it has been reported that SpGEMM in CombBLAS consumes lots of memory space and does not work well with millions-edge-graphs [93]. Hence, we introduce a new approach that avoids the expensive operation in Section 6.4.1. Since structural similarity is defined over connected node pairs, we remove neighbor counts on disconnected node pairs computed by SpGEMM in line 6. At the same time, we increase each value by two. Next, the algorithm computes $\sqrt{|\Gamma(v)| |\Gamma(w)|}$. For this, it computes degrees of nodes, $d$, and assigns them to source nodes of the edges in matrix $A$ and target nodes of the edges in matrix $B$ to compute $|\Gamma(v)|$ and $|\Gamma(w)|$, respectively (line 7 - 9). Then, it performs element-wise multiplication and square-root over $A$ and $B$ (line 10) and produces structural similarity in line 11. In the end, it prunes any edges in $D$ which are not involved in $\epsilon$-neighborhood.

FIND CORES($D, \mu$) shown in Algorithm 10 retrieves core nodes by counting the number of $\epsilon$-neighbors for each node in $D$ with a user-supplying parameter $\mu$. First, it accumulates the number of $\epsilon$-neighborhood edges for each node along each row (line 2) and checks if it is greater
Algorithm 10 Finding Cores

1: function FINDCORES(D, \( \mu \))
2: \( c \leftarrow \text{Reduce}(D, \text{ROW}, f : (e_D > 0) \ ? \ \text{sum} + 1 : \text{sum}, 0) \)
3: \( c \leftarrow \text{Apply}(c, \text{func} : (e_c \geq \mu) \ ? \ 1 : 0) \)
4: \( A \leftarrow \text{Copy}(D) \)
5: \( A \leftarrow \text{Apply}(A, f : 1) \)
6: \( B \leftarrow \text{Copy}(A) \)
7: \( A \leftarrow \text{DApplay}(A, c, \text{ROW}, f : e_A \times e_c) \)
8: \( B \leftarrow \text{DApplay}(B, c, \text{COL}, f : e_B \times e_c) \)
9: \( A \leftarrow \text{Prune}(A, p : e_A = 0) \)
10: \( B \leftarrow \text{Prune}(B, p : e_B = 0) \)
11: \( C \leftarrow \text{EApplay}(A, B, p : e_A \neq \emptyset \land e_B \neq \emptyset, f : 1) \)
12: return \((C, c)\)

than or equal to \( \mu \) (line 3). Any nodes that meet the condition are marked as one in vector \( c \).
In line 5 - 11, it investigates whether each edge connects to any core node. Edges that connect to at least one core are marked as one in matrix \( C \). That is, it only retrieves edges that connect cores or cores and borders.

Algorithm 11 Connected Components Clustering

1: function CONNECTEDCOMPONENTS(C)
2: \( C \leftarrow \text{AddLoops}(C) \)
3: \( l \leftarrow \text{Range}(\text{NumberOfRows}(C), 0) \)
4: \( p \leftarrow \text{Copy}(l) \)
5: while \( \text{NNZ}(p) > 0 \) do
6: \( p \leftarrow \text{SpMV}(C, p, f_{add} : \max(e_p, \text{sum}), f_{mul} : e_p) \)
7: \( p \leftarrow \text{EApplay}(p, l, p : e_p \neq e_l, f : e_p) \)
8: \( l \leftarrow \text{Set}(l, p) \)
9: return \( l \)

CONNECTEDCOMPONENTS(C) in Algorithm 11 groups core and border nodes in \( C \) according to their structural reachability, using connected components operation. The connected components operation performs \textit{maximum label propagation} [51]. First, it initializes each node’s label with its ID (line 3). Then, each node sends its label to its neighbors and updates its label to the maximum label sent from its neighbors (line 6 - 7). Previous node labels are recorded in vector \( l \) and newly updated labels are stored in vector \( p \). The label propagation is repeated until there is no label update (line 5). Since we compare the labels of each node’s neighbors including its own label, we add self-loop edges into \( C \) (line 2).
Algorithm 12 Hubs and Outliers Assigning

1: function FindHubsAndOutliers($G, D, c, l$)
2:     $r \leftarrow$ Range(NumberOfRows($G$), 0)
3:     $o \leftarrow$ Ones(NumberOfRows($G$))
4:     $A \leftarrow$ SpParMat($c, r, o$)
5:     $f \leftarrow$ Reduce($A$, ROW, $f : e_A + sum$, 0)
6:     $f \leftarrow$ EApply($f, c, p : e_c \neq 1, func : e_f$)
7:     $s \leftarrow$ EApply($f, c, p : e_f = 1 \land e_c \neq 1, f : 1$)
8:     $l \leftarrow$ EApply($l, s, p : e_s \neq \emptyset, f : -1$)
9:     $D \leftarrow$ Apply($D, f : 1$)
10:    $D \leftarrow$ DApply($D, c, COL, f : e_D \times e_c$)
11:    $D \leftarrow$ DApply($D, l, COL, f : e_D \times e_l$)
12:    $D \leftarrow$ Prune($D, p : e_D = 0$)
13:    $b \leftarrow$ Reduce($D$, ROW, $f : \text{max}(e_d, sum)$)
14:    $l \leftarrow$ EApply($l, b, p : e_b > 0, f : e_b$)
15:    $s \leftarrow$ EApply($s, b, p : e_b < 0, f : 1$)
16:    $G \leftarrow$ DApply($G, s, ROW, f : (e_s > 0) ? e_G : 0$)
17:    $G \leftarrow$ Prune($G, p : e_G \leq 0$)
18:    $G \leftarrow$ DApply($G, l, COL, f : (e_G > 0) ? e_l : 0$)
19:    $G \leftarrow$ Prune($G, p : e_G < 0$)
20:    $n \leftarrow$ Reduce($G$, ROW, $f = \text{FUNC1}, -2$)
21:    $l \leftarrow$ EApply($l, n, p : e_p = -1, f : -3$)
22: function FUNC1($e, sum$)
23:    if $\text{sum} = -2$ then
24:        return $e$
25:    else if $\text{sum} = -1$ then
26:        return $\text{sum}$
27:    else
28:        if $e = -2$ then
29:            return $\text{sum}$
30:        else
31:            if $\text{sum} \neq e$ then
32:                return $-1$
33:            else
34:                return $e$
35: end function

72
In the end, \texttt{FindHubsAndOutliers}(G, D, c, l) in Algorithm 12 finds hubs and outliers and marks them in the label vector \( l \). First, it creates a matrix \( A \) so that \( i \)th row has all the nodes which have a label \( i \) (line 2 - 4). Then, it computes the frequency of each label by counting the number of nodes along each row (line 5). Such frequencies are store in a vector \( f \). Then, it finds solitary nodes which are not cores and form single clusters, and marks them as outliers (-1) in \( l \) (line 6 - 8). Next, it finds borders by checking they are connected to any cores (line 9 - 13) and set their labels to the maximum labels of their neighboring cores (line 14). In the end, all the hubs are retrieved by checking that they are connected to more than one clusters (line 15 - 19). Hubs are marked as -2 (line 20) and outliers are marked as -3 in \( l \).

### 6.4.2 Optimization of the Algorithm

As discussed in Section 6.4.1, the algorithm uses \texttt{SpGEMM} for computing structural similarity, which is an expensive operation and prohibitive when processing graphs of more than million edges. In real-world graph analytics, pre-processing graph data is often performed in order to reform their serialization formats since current graph analytics frameworks support different sets of limited formats [40]. For CombBLAS, graphs have to be represented in either MTX or a specific binary format it supports. Hence, we divide the algorithm into two steps: pre-processing and community detection and move the computation of counting shared neighbors into the former step. The advantages of this movement are three-folds. First, we can avoid the expensive matrix-matrix computation. Second, input graph data can be compacted by transforming them into the binary graph representation. Such graph compaction can reduce I/O overhead along with data parsing overhead. Last, the computation can be reused since data scientists frequently evaluate resulting communities with different settings of parameters \((\mu \text{ and } \epsilon)\). Moreover, computed shared neighbor counts are not only used in structural graph clustering algorithms [113] but also used for computing other graph measures such as edge embeddedness [24] in social network analysis.

We parallelize computations in pre-processing step by using hybrid MPI and OpenMP programming as shown in Figure 6.1. In this step, \( R \) communicators are configured and \( C \) processes are assigned to each communicator. Pre-processing is required especially for converting graphs in arbitrary serialization formats to a format suitable for a target graph analysis system and some graph serialization formats such as \texttt{GraphML} [13] and \texttt{RDF N-Triples} [9] have arbitrary length of each edge and metadata, which makes complicated to use parallel I/O. Therefore, each process in this step reads input graph \( G_m \) and loads only edges assigned to it. First, edges are divided into \( R \) partitions each of which is assigned to a single communicator. Hence, processes in a communicator store the same edge partition in memory. Nodes are equally divided into \( C \) partitions and each process in a communicator separately generates an \( n \times c \) sub-matrix.
from \( G_{in} \) so that it stores a fixed number of columns (\( c = \left\lceil \frac{n}{C} \right\rceil \) where \( n \) is the number of nodes and \( C \) is the number of processes in each communicator). Since graphs are sparse, we manage each sub-matrix as an array of sparse bitmaps. We leverage a sparse bitmap structure in Roaring [52], which provides compressed bitmap representation along with efficient bitwise operations. In each process, multiple threads are created and assigned some portion of edges. Each thread finds a corresponding pair of bitmaps per edge, intersects them, and counts the number of common elements. Computed counts are aggregated to every process in each communicator, and they perform parallel I/O at their exact offset in a binary file. This pre-processing step requires a modification of Algorithm 9 as shown in Algorithm 13.

**Algorithm 13** Computing \( \epsilon \) Distance

1: function \textsc{ComputeEpsilonDistance}(\( G, \epsilon \))
2: \( D \leftarrow \text{Copy}(G) \)
3: \( A \leftarrow \text{Copy}(G) \)
4: \( B \leftarrow \text{Copy}(G) \)
5: \( D \leftarrow \text{EApply}(D, G, p : e_G \neq \emptyset, f : (e_D \neq \emptyset) ? e_D + 2 : 2) \)
6: \( d \leftarrow \text{Reduce}(G, \text{COL}, f : e_G + \text{sum}, 0) \)
7: \( A \leftarrow \text{DApply}(A, d, \text{ROW}, f : e_A + e_d) \)
8: \( B \leftarrow \text{DApply}(B, d, \text{COL}, f : e_B + e_d) \)
9: \( A \leftarrow \text{EApply}(A, B, p : e_A \neq \emptyset \land e_B \neq \emptyset, f : \sqrt{e_A \times e_B}) \)
10: \( D \leftarrow \text{EApply}(D, A, p : e_D \neq \emptyset \land e_A \neq \emptyset, f : e_D/e_A) \)
11: \( D \leftarrow \text{Prune}(D, p : e_D < \epsilon) \)
12: return \( D \)
6.5 Evaluation

We evaluate the performance and the properties of our parallel SCAN algorithm. In this section, we describe the experimental environments along with the evaluation results.

6.5.1 Experiment Setup

We used Henry2, an in-house HPC Linux cluster in the North Carolina State University [98]. The cluster is based on IBM Blade Center hardware [21] and consists of 1,233 dual Xeon computing nodes. In this evaluation, we allocated 49 processors and 2 processors per a node. For software packages, we used CombBLAS [15] 1.6 with MPICH3 compiled with Intel compiler 16.0. We chose four undirected graph datasets from Stanford Network Analysis Platform (SNAP) which aims at evaluating high performance for large network analysis [53]. Table 6.1 shows a summary of the datasets. We set $\epsilon = 0.7$ and $\mu = 2$ according to the recommendation of the original paper.

Table 6.1. Evaluation Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Com-Amazon</td>
<td>334,863</td>
<td>925,872</td>
</tr>
<tr>
<td>Com-DBLP</td>
<td>317,080</td>
<td>1,049,866</td>
</tr>
<tr>
<td>Com-Youtube</td>
<td>1,134,890</td>
<td>2,987,624</td>
</tr>
<tr>
<td>Com-Orkut</td>
<td>3,072,441</td>
<td>117,185,083</td>
</tr>
</tbody>
</table>

6.5.2 Experiment Results

Table 6.2. Generated Dataset Size

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MTX (MB)</th>
<th>Binary (MB)</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Com-Amazon</td>
<td>13.9/19.4</td>
<td>11.1</td>
<td>20.1/42.7</td>
</tr>
<tr>
<td>Com-DBLP</td>
<td>15.7/22.0</td>
<td>12.5</td>
<td>20.3/43.1</td>
</tr>
<tr>
<td>Com-Youtube</td>
<td>44.4/71.7</td>
<td>35.8</td>
<td>19.3/50.0</td>
</tr>
<tr>
<td>Com-Orkut</td>
<td>2062.9/2812.4</td>
<td>1406.2</td>
<td>31.8/50.0</td>
</tr>
</tbody>
</table>
Comparison of Pre-processed Data  we first compare the size of the datasets represented in MTX and binary formats which are generated in the pre-processing step. In the pre-processing step, processes leverage parallel I/O to store edges and shared neighbor counts, which requires each process to compute its starting offset. However, when MTX serialization is used, each process cannot compute the offset since node IDs and counts have arbitrary length of digits. Hence, we stored IDs and counts as fixed length-digits in MTX graphs. We also manually generated graphs in MTX where each node ID and count is represented as its minimum length of digits for comparison. For binary representation, we serialized each node ID and count as 4-byte integers. Table 6.2 shows the graph size. Note that the first numbers in the MTX column mean the graph size with minimum length IDs and counts and the second numbers mean the graph size with fixed length. As shown in the table, fixed-length MTX serialization produced 1.3 - 1.6 times larger graphs than minimum-length serialization. On the other hand, binary serialization could reduce the graph size around 20 - 30% compared to minimum-length MTX serialization and 42 - 50% to fixed-length serialization. Such reduction could reduce I/O and communication overheads and result in up to 5.2 times better data loading time than fixed-length serialization and 2.2 times better than minimum-length serialization as shown in Figure 6.2.

Pre-processing Performance  we investigated the performance impact of different number of processes and threads on the pre-processing step. Figure 6.3(a) shows the pre-processing time
Figure 6.3. Pre-processing Performance

with varying number of processes. In the first place, we observed that pre-processing graphs that have roughly three million edges or less just takes a few seconds even on a single processor. For example, it spent 1.9, 3.0, and 8.3 seconds with Com-Amazon, Com-DBLP, and Com-Youtube, respectively. Second observation is that pre-processing time exponentially increases as more number of edges are present in graphs. Com-DBLP, Com-Youtube, and Com-Orkut have around 1.1, 3.2, and 126.5 times more edges than Com-Amazon, which resulted in ×1.6, ×4.4, and ×336.9 in pre-processing time. This is because finding shared neighbors of all pairs of connected nodes in each edge partition requires an iteration over edges and, for each edge, an intersection of two bitmaps whose time complexity is a function of their cardinalities [52]. Last, we observed that multi-processing in this step can greatly improve its performance. For example, on Com-Orkut, it showed ×5.8 performance improvement with 36 processes.

We checked the impact of multi-threading. In this evaluation, we ran a single process on Com-Orkut and increased the number of threads from 1 to 8. Figure 6.3(b) shows the result. We observed that multi-threading could a lot improve its performance. The pre-processing time could be improved up to ×2.7. The major computation in this step is repeating intersections of bitmaps for edges, and this causes frequent memory accesses, saturating memory bandwidth. Hence, as the number of threads increases, the impact of multi-threading becomes weak. We also observed that edges are not even distributed across multiple processes, which resulted in certain impacts on bitmap intersection. This problem should be addressed in the future.
Community Detection Performance we evaluated the performance of our community detection algorithm parallelized on CombBLAS. As shown in Figure 6.4, it could process three smallest graphs just in a few seconds even on a single processor: 1.9, 3.1, and 6.7 seconds with Com-Amazon, Com-DBLP, and Com-Youtube, respectively. With Com-Orkut, we checked that our community detection algorithm could achieve great gain in performance. With 16 processes, it showed $\times25$ performance improvement. In our settings, more than 16 processes could not improve its performance much.
Comparison of SpGEMM-based and Two-step Approaches

we compare the SpGEMM-based approach and the pre-processing approach in Figure 6.5. Note that we summed up pre-processing time and community detection time for the two-step approach. Also, note that execution time of the two-step approach includes pre-processing time and community detection time. First, the SpGEMM operation could not be processed even with 3 million edge-graph (Com-Youtube) because it rapidly consumed memory space. Next, even though the operation could be processed over Com-Amazon and Com-DBLP, the SpGEMM-based approach did not show any superior performance to the two-step approach with 1 - 9 processes. When 25 - 49 processes ran, the SpGEMM-based approach showed better performance than the two-step approach but it was mainly due to increased I/O and communication overheads in the pre-processing step. The original SCAN algorithm computes structural similarity for pairs of connected nodes rather than all-pairs of nodes. While CombBLAS provides a set of more extended primitive operations than other high-performance graph analysis frameworks. However, currently, it does not have a primitive for performing intersection of selected rows. Hence, we still need to rely on an expensive SpGEMM operation. In order to avoid the use of the operation, we moved the required computation into a separate process and could achieve better performance for our community detection operation.
Comparison of CombBLAS and KDT Implementations  

the features of CombBLAS are also available in Knowledge Discovery Toolbox (KDT) [58]. KDT provides major distributed data structures such as sparse/dense matrix and vectors along with a subset of primitive operations as python wrappers so that data scientists can easily perform graph analysis. At this time, the latest version is KDT 0.3, which is based on CombBLAS 1.2. We also implemented our parallel SCAN algorithm in KDT. Since there have been radical optimizations along with additional features on CombBLAS, it does not bring much information to compare our two implementations. Nevertheless, we compared the two implementations in order for data scientists to understand them and make better choices for conducting their graph analytic tasks. Figure 6.6 shows the execution time of those two implementations for community detection over Com-Orkut. First, when running on few processors, computation contributes to the majority of execution time but both implementations could reduce the computation time by using multi-processing. We observed that, as more processes are used, data loading becomes the main factor that affects the overall performance. Especially, the KDT implementation showed ×4 to ×8 more data loading time since the current KDT does not support parallel I/O even though its base CombBLAS framework supports parallel I/O when graphs are represented in the binary format. Next, the CombBLAS implementation showed ×2.5 - ×11 better performance than its KDT counterpart. The main reason for this difference is that the latest CombBLAS operations are more optimized than the one KDT uses. For example, each primitive leverages OpenMP for concurrent processing. Another reason is that the latest CombBLAS provides more extended
primitives. For example, since KDT does not provide $Prune(M \text{ or } v, p)$ and $AddLoops(M)$, we replaced those operations with a set of different primitives, which resulted in additional overheads and increased memory requirements.

6.6 Conclusion

In this work, we introduced a parallel structural clustering algorithm based on matrix operations and optimized on a recent high performance graph analysis framework. The proposed approach moves computationally costly similarity computations into a pre-processing phase. This could not only enable such computations to be reused with different parameter settings but also avoid an expensive matrix-matrix multiplication operation. We also parallelized the similarity computation by using hybrid parallel programming. The experiment results showed that both multi-threading and multi-processing highly improved the performance of similarity computation.
Chapter 7

Conclusion

As graph has gained its popularity, graph analytics positions its importance in modern data analytics. In data analytics, a routine yet key task is discovering important properties and valuable knowledge from databases. This is the primary goal of data analytics. Likewise, analyzing graph databases is at the core of graph analytics, which helps data scientists to understand and discover interesting properties and knowledge embedded in them. As presented in Chapter 2, graph pattern matching and mining are fundamental tasks in graph analytics. Having their unique roles, they are cooperatively used for extensive graph explorations.

Recently, various graph analysis platforms have come out with novel techniques. However, they provide limited features and capabilities insufficient to cover diversity existing in modern graph data models and requirements for analysis tasks. Hence, multiple systems are often used together to compensate their limitations. In graph pattern matching, those systems support graph pattern matching tasks in common, which gives many available options to data scientists. However, the different interfaces and computation and optimization techniques and their impacts have not been well compared across multiple data models. This makes it complicated to select suitable systems at the right place. Next, most of the systems provide limited number of built-in graph mining operations that are optimized by system developers. While those systems provide interfaces with which extended operations can be implemented, producing and optimizing such operations being aware of their properties is burdensome work to data scientists.

In this thesis, we made efforts to resolve the above-mentioned problems. In Chapter 3, we selected six recent graph processing systems across RDF and property graph data models, and evaluated them with graph pattern matching workloads. We discussed their advantages and limitations observed from our benchmark work in order to assist data scientists to understand those systems and give guidelines. Chapter 4-6 introduced several important graph mining operations built on some of recent systems. In Chapter 4, we especially focused on RDF-based systems because despite of its importance, RDF does not provide built-in graph mining
operations. We overcame that limitation by developing five graph mining operations optimized on triplestores. In Chapter 5-6, we introduced two cluster analysis operations. The operation introduced in Chapter 5 clusters entities in RDF graphs based on their spatial and non-spatial attributes. The operation presented in Chapter 6 is a structural clustering operation built on a high-performance matrix-based graph analysis framework. We admit that our graph mining operations are still not enough for extensive graph analysis. However, we reported and discussed several limitations we observed from developing those operations so that we can help to develop more extended operations and improve current systems.

Having investigated recent graph analysis systems, we observed that most of recent graph analysis systems enable their graph processing capabilities by leveraging general data processing systems. That is, they give a graph abstraction over general data processing systems, which bridges a gap between graph and general data models. For example, GraphX and Graphframes give a graph abstraction over relational data model of Apache Spark. Graph operations of those systems are built by assembling parallelized primitive operations of Apache Spark. For another example, KDT and CombBLAS are an abstraction over general matrix data model and their graph operations are composed of matrix and vector operations. We observed that such an abstraction can cause three problems which limit the performance and usability of recent graph systems. First, developing optimal graph operations typically requires good understanding of underlying general data analysis platforms. Second, the possibility of developing new graph operations and their optimality highly depend on provided data analysis primitives. Last, a gap between two different data models often degrades the optimality of graph operations. We expect that a top-down approach may be a solution to resolve those problems. In this approach, we investigate a set of common graph primitives required to build various graph analysis operations. Then, we compare different computation and data models to check their suitability and efficiency for building those primitives. Last, we develop a graph analysis system with selected models and optimized graph operations. We leave this as our future work.
REFERENCES


[22] DBpedia. Dbpedia facts & figures.


[60] Kristyn Maschhoff, Robert Vesse, and James Maltby. Porting the urika-gd graph analytic database to the xc30/40 platform. In *Cray User Group Conference (CUG15), Chicago, IL, 2015*.


[62] Harvey J Miller and Jiawei Han. *Geographic data mining and knowledge discovery*. CRC Press, 2009.


[80] QGIS. Qgis: a free and open source geographic information system.


[82] RDF4J. Openrdf sesame.


[107] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.


91


