Towards a Unified Framework for Network Analytics

Minjian Liu

A thesis submitted in partial fulfillment of the degree of
Master of Computing at
The Department of Computer Science
Australian National University

October 2015

Typeset in Palatino by TeX and LaTeX 2_{ε}.

Except where otherwise indicated, this thesis is my own original work.

Minjian Liu
25 October 2015

To my parents, for supporting me all the way.

Acknowledgements

First and foremost, an enormous thank you to my supervisor, Qing Wang for giving me an opportunity to do this research project with you, even though you know my bachelor degree is about medicine. Without your wealth of knowledge in network analytics and your dedicated assistance, I would never get the current achievement for this project. In this year, we met, we discussed, we argued and finally we consented for every point in this project. Thank you for imparting me knowledge like a teacher, thank you for giving me encouragement like a friend , and thank you for showing me patience like a sister.

Thank you to Peter Christen, Dinusha Vatsalan, Jeffrey Fisher, and Thilina Ranbaduge for spending time to listen to my presentation rehearsal and giving me so many great advices about the slides and the presentation skills.

Thank you to John Slaney for organising all honours cohort meetings and honours talks. It was interesting and exciting to communicate with other honours students about their projects.

Thank you to the trainers of Uplooking Technology Co,.Ltd for training and teaching me how to be a Linux Architect and Server Developer in the year before I came to ANU. Without this training, it is difficult for me to survive in the computer science field with a medicine background.

Thank you to my fiancee, Wen, for making delicious food for me when I was tied up with the project and giving me courage when I was stressed and frustrated.

Finally, thank you to my parent and my sister for understanding me to give up five-year medicine study to pursue my real interest and dream in computer science field and giving me supports all the way.

Abstract

Network analytics has started to become increasingly popular and various specialised graph systems for network analytics have been proposed in recent years. However, most network data is still collected and managed in relational databases and the use of relational databases for network analytics is largely ignored.

This situation then raises a question of whether or not relational databases have limitations for network analytics. The relational model is indeed inefficient for some network analysis tasks which often require multiple expensive joins for tables and the SQL query language also makes it difficult to express network analysis operations. Even so, relational databases are already used for a variety of other analysis tasks and they are filled with many great features, such as query optimisation, fault tolerance, secure transaction, integrity constraints and so on.

In this thesis, we present a unified framework for network analytics, which provides a data model that extends relational databases with network analysis capability and a query language to manipulate data for relational analysis, network analysis or a mix of them. In addition, this unified framework also includes a query engine that is built with an open-source relational database (PostgreSQL) for processing queries that are written in the query language of this framework. The experimental result indicates the query engine is flexible to process different types of queries and is able to achieve comparable or better performance in most cases.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

”Network analytics” is a broad term that is widely used in various areas such as social networks, transportation systems, bioinformatics, communication networks and so on. From the computer science perspective, it can be subsumed under ”applied graph theory”, since the structural and algorithmic aspects of abstract graphs are the prevalent methodological determinants in many applications of network analytics [26].

Nowadays, more and more large networks become available. Analysing these networks to derive key insights for business is critical for many enterprises and organisations. As a result, in recent years, network analytics has started to become increasingly popular. In response to the growing popularity for network analytics, a deluge of specialised graph systems have been developed, including Pregel [39], Giraph [6], GraphLab [38], Giraph++ [49], NScale [44], AllegroGraph [2], and Neo4j [14].

For many enterprises and organisations, these specialised graph systems are typically used in conjunction with relational databases because network data are often stored and managed in relational databases in the first place. As a result, within two separate systems, a common usage pattern for network analytics is described as follows: (1) exporting data from a relational database to text files (e.g. CSV, XML, TXT), (2) importing those text files into graph systems, (3) running analysis and getting results from those graph systems, (4) possibly reloading results into relational databases for further processing [36]. In this pattern, data analysts need to move data around, which is an expensive step. It is also cumbersome to learn and maintain two separate systems.

Currently, most network analysis tasks follow this pattern. This is because relational databases have limitations for network analytics. For example, it is difficult to use SQL, the query language of relational databases, to express network analysis operations. Even for simple operations such as neighbourhood accesses, a SQL query would require multiple joins and become complex. Moreover, even if we can write an SQL query for network analysis operations, relational databases are inefficient for running iterative algorithms (e.g. PageRank, finding shortest paths) [36].

However, in real-world networks, vertices and edges are often accompanied by some attributes. For example, in a social network, vertices may have attributes to describe the properties of each person, such as name, gender and location. Edges may also be of different types, such as friends, classmates and colleagues. Accessing these attributes is typically about relational analysis.

Therefore, we come up with a question: "what if we can perform network analytics directly with relational databases?". If it is convenient and efficient to perform network analytics with relational databases, the following benefits can be derived:

- We do not need to export or import data between two kinds of systems.
- We can combine network analysis and relational analysis to retrieve more valuable and interesting information.
- We can inherit many great features of relational databases, such as query optimisation, fault tolerance, secure transaction, integrity constraints and so on.

Furthermore, some existing works indicate relational databases, via using some optimisation techniques, can achieve a better or comparable performance than specialized graph systems for some network analysis tasks, such as triangle counting [36], subgraph pattern matching [35], and weakly connected component [32].

Therefore, unlike those graph systems, the motivation of this thesis is to develop a unified framework which is able to extend relational databases with network analysis capability.

1.1 Objectives

The goal of this thesis is to develop a unified framework for network analytics. This framework aims to provide users a unified method to deal with network analysis tasks, relational analysis tasks, and even a mix of them. The specific objectives are described as follows:

- Develop a data model that supports data analysis over both relations and graphs.
- Design a query language that enables users to write queries for network analysis operations, relational analysis operations and even a mix of them.
- Implement an efficient query engine that is able to efficiently process different types of queries.

1.2 Contributions

This thesis has four main contributions:

- We have developed a new data model for network analytics, called Relation-Graph (RG) model. This RG model takes a relational core in the center and the relation core is surrounded by a number of graphical views. Between the relational core and the graphical views, there are a number of Relation-Graph mappers (RG mappers) that take a number of relations to generate a graph. Using the RG model, users are able to manage data in a relational database and perform network analytics with it.

- We have designed a SQL-like query language for network analytics, called Relation-Graph Structured Query Language (RG-SQL). It extends SQL with ranking, clustering, path finding and graph constructing operations. In essence, RG-SQL is a relation-graph interactive query language. Users can use traditional SELECT-FROM-WHERE statements to extract a sub-graph or use aggregate and join operations for further processing network analysis results. It also supports nested queries for advanced network analysis tasks that involve analysis over both graphs and relations.

- We have designed an implementation architecture for a query engine, called RG engine, and have implemented it with an open-source relational database (PostgreSQL). This architecture allows us to incorporates different graph analysis tools as plug-ins for supporting network analysis algorithms. It is flexible to add, modify or delete algorithms within this architecture.

- We have conducted two experiments. One experiment is to evaluate the performance of three existing graph analysis tools (SNAP [21], NetworkX [16], Graph-tool [7]). In this experiment, we use the Erdos-Renyi methods [31] to create random graphs as inputs, run different network analysis algorithms using these tools and evaluate their time performance and memory performance. Another experiment is to compare the RG engine with the query engines of a relational database (PostgreSQL) and a graph database (Neo4j) to indicate the efficiency of the RG engine.

1.3 Outline

The rest of this thesis is divided into the following 6 chapters:

- Chapter 2 introduces three typical types of existing systems for network analytics. We discuss the advantages and limitations of these existing systems and explains why a unified framework is needed.

- Chapter 3 presents the formal definition of the RG model and introduces the main features of RG-SQL. We use the ACM bibliographical network as an example to illustrate the key concepts of our data model and to demonstrate how to write queries using RG-SQL.

- Chapter 4 discusses the main phrases in the query processing, presents the architecture of our query engine and proposes some query optimization strategies that can be incorporated into the implementation of the query engine.
- Chapter 5 presents our experimental results. One experiment we have conducted is to evaluate the performance of three graph analysis tools. Another experiment is to compare our query engine with the query engines of a relational database (PostgreSQL) and a graph database (Neo4j).
- Chapter 6 concludes the thesis and discusses the future work.

Chapter 2

Background and Related Work

In this chapter, we introduce three types of systems that have been proposed in the past few years. In Section 2.1, we first present vertex-centric systems (e.g. Pregel [39], Giraph [6], GraphLab [38]) and neighbourhood-centric systems (e.g. Giraph++ [49], NScale [44]). These two kinds of systems are closely related because neighbourhood-centric systems are developed upon the concepts of vertex-centric systems. In Section 2.2, we introduce the embryonic-but-growing-significantly graph databases such as Neo4j [14] and AllegroGraph [2]. Then Section 2.3 describes two SQL-based systems, GraphiQL [36] and Grail [32], which are built upon the traditional relational databases. We will discuss how our work is different from these SQL-based systems. A summary for different types network analysis systems is given in Section 2.4.

2.1 Vertex-centric and Neighbourhood-centric Systems

Vertex-centric systems were developed for efficiently processing large-scale graphs in a distributed environment. In vertex-centric systems, generally, a large-scale graph is divided into several partitions. Each of them has vertices and outgoing edges that are stored distributively. Figure 2.1 shows an example data model used in vertex-centric systems. In Figure 2.1, an input graph is divided into three partitions (P1, P2, P3) and each partition contains a set of vertices. One vertex has a unique ID (e.g. V1), a set of values (a vertex has one value about out-degree in this example) and a set of outgoing edges for finding targets to pass messages.

In vertex-centric systems, each vertex is considered as an independent computing unit and users are required to express their network analysis algorithms in the so-called ”thinking like a vertex” programming mode [39]. The algorithm computation is processed at the vertex level but the computation models of different systems are slightly different. The representative vertex-centric systems include Pregel [39], Giraph [6] (an open source implementation of Pregel) and GraphLab [38]. For Pregel and Giraph, their computation models are both based on message passing which enables vertices to be computed in parallel. Each vertex is associated with two states – **active** and **inactive**. At the beginning, all vertices are active. Then following a sequence of iterations, called **supersteps**, messages are passed from one vertex to anther vertex. In

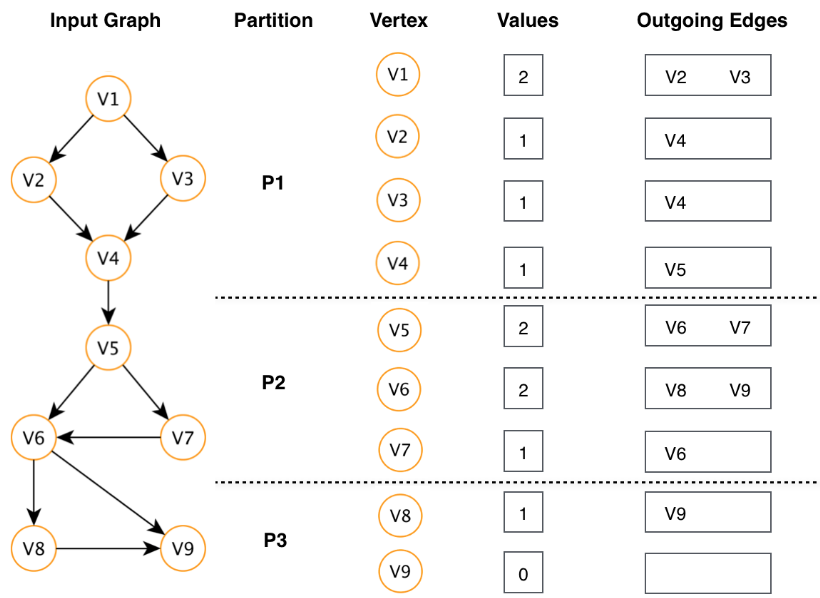


Figure 2.1: Data Model Example for Vertex-centric Systems

a superstep i, each active vertex receives messages from other vertices in the superstep i-1, updates its values and sends messages to other vertices in the superstep i+1. When passing messages among vertices, the states of vertices will be changed from active to inactive. When all vertices become inactive, the overall program terminates. For GraphLab, unlike Pregel, the computation is a stateless function that operates on the values of vertices which are associated with small neighbourhood in a graph. A vertex reads and updates its values or values of its neighbours. Hence, without passing message, GraphLab allows asynchronous iterative computation. Moreover, GraphLab requires the graph structure to be static while Pregel supports graph mutation during computation. In addition to the systems mentioned above, there are other vertex-centric systems such as Trinity [48], GRACE [50], Kineograph [28] and so on.

Neighbourhood-centric systems were developed soon after vertex-centric systems were proposed. This is because the vertex-centric model hides the subgraph information via using a collection of unrelated vertices instead of a proper subgraph of the original input graph. So the vertex-centric model restricts optimization for some algorithms (e.g. connected component and PageRank) [49]. The typical neighbourhood-centric systems include Giraph++ [49] (developed upon Giraph) and NScale [44]. Figure 2.2 shows an example data model for neighbourhood-centric systems based on the concepts of Giraph++. In Figure 2.2, the neighbourhood-centric model divides the original input graph into partitions as subgraphs (G1, G2, G3). The subgraph stores the information about vertices and their connections. Each vertex has a unique id (e.g. V1) and a set of values (this example considers the out-degree value). The model categorises vertices into two types – **internal vertices** and **boundary vertices**. The vertices that are used to divide the input graph are the **boundary vertices** (V4 in G2 and V6 in

G3 are boundary vertices). A vertex is an **internal vertex** in an exactly one subgraph and this subgraph is called the **owner** of the vertex (G1 is the owner of vertex V4 and G2 is the owner of vertex V6), but this internal vertex can be a boundary vertex in zero or more subgraphs. The vertices V1, V2, V3 and V4 are the internal vertices in G1, The vertices V5, V6 and V7 are the internal vertices in G2 and the vertices V8, V9 are the internal vertices in G3. For all internal vertices in a subgraph, the owner subgraph stores all the values. But for a boundary vertex, the vertex value is just a temporary local copy and its primary information resides in its owner subgraph.

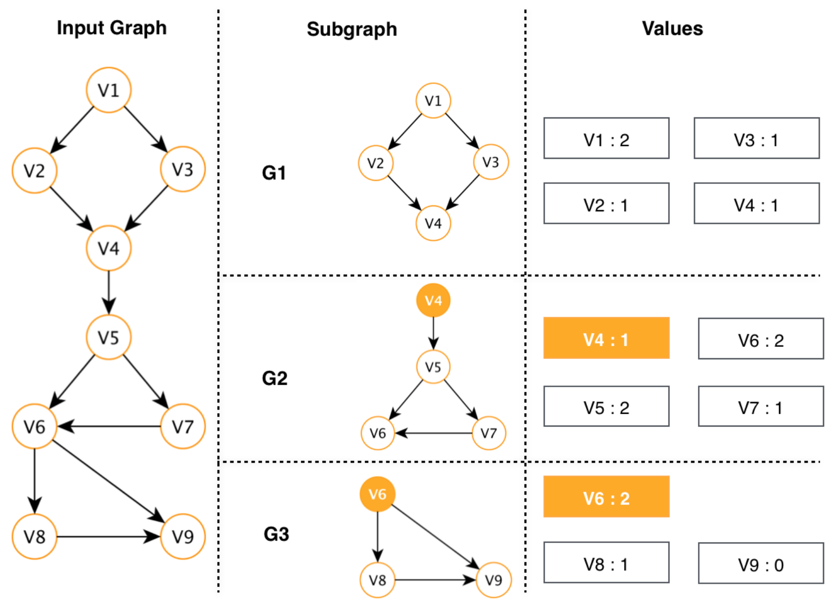


Figure 2.2: Data Model Example for Neighbourhood-centric Systems

In terms of the computation model of neighbourhood-centric systems, it is similar to the message passing model, but the messages are only sent from boundary vertices to their corresponding internal vertices. As message passing through internal vertices is cheap and immediate, this model can reduce the number of messages passing through cross-partition edges so as to improve the efficiency.

The vertex-centric model is simple-to-use for programming and has been proved to be useful for many network analysis algorithms. The neighbourhood-centric model is not intended to replace the vertex-centric model, instead, it can be implemented in the same system such as Giraph and Giraph++ for achieving better performance. Our concern for both vertex-centric and neighbourhood-centric systems is that they require users to do imperative programming as they do not provide any declarative languages for querying data. Moreover, some recent works indicate that simply using a SQL-based system can achieve a better or comparable performance than vertex-centric systems for some network analysis tasks, such as PageRank, triangle counting, connected components and single source shortest path [32] [36].

2.2 Graph Databases

Graph databases emphasise on efficiently managing and processing data as graphs for network analytics. For example, for the network analysis tasks like finding friends of friends, relational databases need to use expensive join operations on tables. The key idea of data model in graph databases is to include all connections between objects so as to generate a cohesive picture of the whole data. As a result, there are two typical data models used in graph databases – **Property graphs** and **RDF triple stores**.

Property graphs are often said to be "whiteboard-friendly" by data analysts because when they draw a picture to describe data, it is often naturally a property graph [40]. Figure 2.3 shows an example property graph. A standard graph structure consists of vertices and edges, denoted by $G = (V, E)$ where V represents vertices and E represents edges. However, a current popular property graph structure also contains properties in addition to vertices and edges, denoted by $G = (V, E, \lambda)$ where λ represents **properties**. In Figure 2.3, vertices contains properties in the form of arbitrary key-value pairs where keys (e.g. T1, U5) are strings and values (e.g. Name, State, Comment Count) have various data types (e.g. string, integer). An edge (e.g. Tweets, Follows, Re-tweets) that connects two vertices is directed and labelled. Like vertices, edges can also have properties (e.g. Date, Time) which is useful for providing extra metadata for network analysis algorithms and adding semantics to relationships such as quality and weight [46]. Some typical graph databases that are using property graphs include Neo4j [14], Titan [24] and OrientDB [15]. Although these graph databases use the same data model, they have different query languages for data manipulation. Neo4j has its exclusive Cypher query language for graph traversal and Titan uses Gremlin as its graph traversal language. As OrientDB supports both schemaless (OrientDB graph model) and schema-based model (OrientDB document model), it not only uses Gremlin for graph traversal but also uses SQL on top of Gremlin for querying structured data.

RDF (Resource Description Framework) **triple stores**, created in 1999 [41], were designed to support the semantic web by adding semantic markup to the links that connect web resources. In fact, a typical RDF triple is a **subject-predicate-object** data structure and RDF databases do not store data as a graph. So RDF databases do not support index-free adjacency [40]. As noted in [40], the reason why RDF triple stores fall under the category of graph databases is that they do offer optimised graph query capabilities when connected structures are created for different independent triples (refer to Figure 2.4 [1]). Some representative graph databases include AllegroGraph [2], Stardog [22], and Apache Jena [3] and SPARQL is the standard query language for RDF triple store.

Unlike vertex-centric and neighbourhood-centric systems, graph databases provide different kinds of declarative query languages to retrieve information. In some net-

[1] source: http://franz.com/agraph/support/documentation/current/agraph-introduction.html

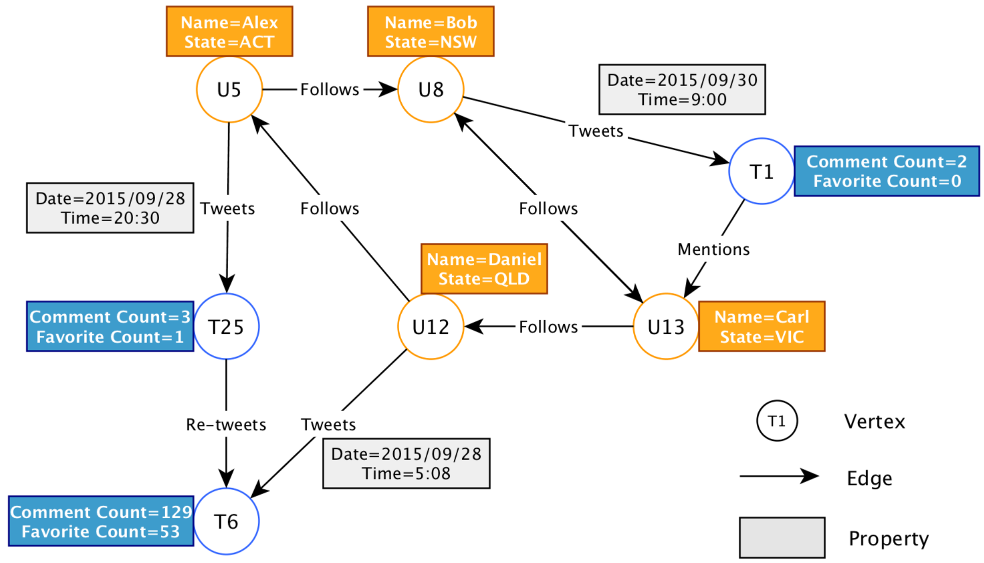


Figure 2.3: Data Model Example for Property Graph

work analysis tasks, particularly in ”friends of friends” queries [30], they are able to achieve far better performance than relational databases that have to use expensive multiple joins on tables. However, relational databases are still widely used by enterprises or organisations and they provide a number of sophisticated optimization technologies (e.g. indexing, materialised views) for managing and processing schema-based data. So relational databases are still our preference for some tasks such as accessing attributes of entities, using aggregate functions and so on.

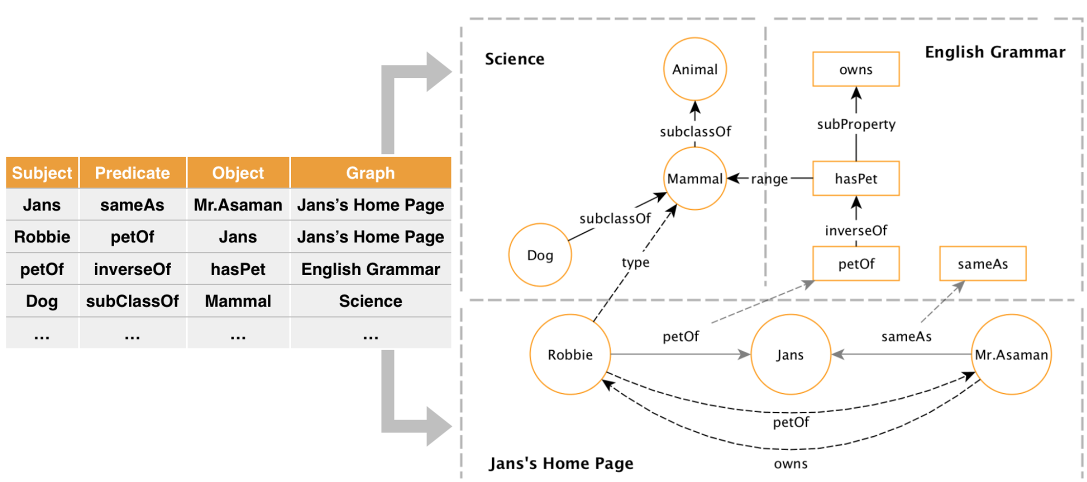


Figure 2.4: RDF Triple Store Model

2.3 SQL-based Systems

As various specialised systems for network analytics have been created in recent years, the use of SQL-based systems for network analytics is largely ignored since users have an impression that systems with a graph model (graph systems) are in the nature of better performance for network analysis tasks. Then some researchers come up with a natural question – "Is it really bad to simply use a SQL-based relational system for both managing and processing network data?". Recently, using SQL-based relational systems for network analytics becomes popular in the research field and some papers demonstrate SQL-based relational systems, compared with graph systems, do have better or competitive performance in some network analysis tasks. The work in [51] shows that Oracle database can achieve better performance for finding shortest paths. The work in [35] proposes query optimization techniques for efficient subgraph pattern matching in PostgreSQL. The works in [37] and [32] both indicate that SQL-based systems are competitive in queries for PageRank, finding single source shortest paths and calculating connected components.

Vertex

id	data	val
A	...	100
B	...	100
C	...	100
D	...	100

Edge

src	dest	data	val
A	B	...	1
A	C	...	2
B	D	...	2
C	D	...	3

(a)

Graph Table

id	type	property1	property2	...
V1	VERTEX	Alice	ACT	...
V2	VERTEX	Bob	NSW	...
V3	VERTEX	Carl	VIC	...
E1	EDGE	V1	V2	...

(b)

Figure 2.5: Data Model Example for SQL-based Relational Systems

Figure 2.5.(a) shows an example data model for **Grail** [32], one SQL-based relational system with a syntactic layer for network analytics. This data model consists of a **vertex table** and an **edge table**. In Figure 2.5.(a), *id* (e.g. V1, V2) in the vertex table represents the unique identifier of a vertex, *src* and *dest* in the edge table respectively represent the source vertex id and the destination vertex id, *data* in both tables contain vertex or edge properties that are irrelevant to the computation and *val* in both tables represents the properties that are relevant to the computation.

Then Figure 2.5.(b) shows an example data model for **GraphiQL** [36], another SQL-based system with a graph intuitive query language. Unlike the data model of Grail, GraphiQL includes all graph elements in one table called **Graph Table** with a purpose that helps users to easily access neighbourhood of vertices and edges without joining tables. In Figure 2.5.(b), every element (either vertex or edge) in a graph table has the default properties *id* (e.g. V1, V2) and *type* (e.g. VERTEX, EDGE) and a number of associated *properties* (e.g. property 1, 2 for vertices respectively relate to name and state whilst for edges they respectively relates to the source vertex and the destination vertex.).

In terms of the computation model of these systems, they are similar but with different implementation methods. Computation of Grail and GraphiQL are vertex-centric with the message passing model (refer to Section 2.1). They translate a vertex-centric program to SQL by creating some intermediate tables and using different relational operators to implement the program. For Grail, it creates temporary tables, such as **next table** and **message table**, to simulate the message passing model. Next table contains id and values for vertices in the next superstep and message table contains id of the target vertices and messages that change vertices' values. For GraphiQL, it creates computation tables that store computation values for vertices and edges, but they are not temporary. In each superstep of the message passing model, old computation tables are replaced by new computation tables with latest values.

In essence, these SQL-based systems (e.g. Grail and GraphiQL) are vertex-centric but they provide declarative query languages for users to do vertex-centric programming and then translate the program into SQL. As these systems need to translate their query languages into SQL, there is a gap between two levels of query languages, which indicates these query languages lack of capability to well interact with SQL, such as using SQL joins or aggregate functions for further querying. In addition, since they use SQL and relational operators for vertex-centric programming, it should have limitations or poor performance for running some network analysis tasks (e.g. find friends-of-friends) which are inefficient via using relational systems.

2.4 Summary

In this chapter, we have introduced three types of systems for network analytics. For vertex-centric and neighbourhood-centric systems, they do not provide declarative languages for users to retrieve data easily. In terms of graph databases, we have demands on not only querying data in graphs but also querying schema-based data. Moreover, most of applications are still using relational databases to manage and process data. As a result, we want a system which is SQL-based, provides a declarative query language and has competitive performance for network analysis tasks. Currently, existing SQL-based relational systems still have limitations: (1) the query languages lack of capability to interact with SQL so we want a declarative query language that is able to well interact with SQL (e.g. using SQL to create graphs or subgraphs, combining the analysis results with SQL joins and aggregate functions to get more information). (2) they can achieve competitive performance for only a few network analysis tasks so we want a flexible way to cope with most of network analysis tasks (e.g. for some tasks we can leverage the graph model and graph computing engines to efficiently get the results, for other tasks we can take advantage of SQL optimization techniques to achieve better performance). Therefore, we propose the our data model and query language in Chapter 3 to meet these requirements.

Chapter 3

Data Model and Query Language

In this chapter, we describe our data model and query language. In Section 3.1, we first define our data model. Then based on our data model, in Section 3.2, we introduce a new query language for network analytics. A summary of our data model and query language is given in Section 3.3.

3.1 Data Model

Our data model consists of a **relational core**, **graphical views** and **relation-graph mappers**. A relational core that contains different relations is in the center of our data model and surrounded by a number of graphical views. Relation-graph mappers are used to map relations to graphical views. As our data model allows to build graphs upon relations, we call it **Relation-Graph data model (RG model)**. Figure 3.1 gives an overview of the RG model based on the ACM bibliographical network [1].

3.1.1 Relational Core

In the RG model, a **relational core** consists of a collection of **relations**. Each relation is described by a **relation schema**, and contains a number of **tuples**. Each tuple represents a fact about objects in real-life applications. Now, we define the following concepts for the relational core.

- Let $\mathcal{D} = \{D_i\}$ where $i \in \mathbb{N}$ be a family of possibly infinite domains and each D_i is referred to one **domain**. For instance, we could have domains such as string, integer, boolean and so forth.

- A **relation schema** R consists of a relation name R and a finite set of attributes $\{A_1, \ldots, A_n\}$ together with an assignment of domains, $dom : R \to \mathcal{D}$, such that each A_i is associated with a domain $dom(A_i)$ where $i \in [1, n]$. We use $attr(R)$ to refer to the set of attributes of R, i.e., $attr(R) = \{A_1, \ldots, A_n\}$.

[1] Provided by ACM Digital Library (http://dl.acm.org/)

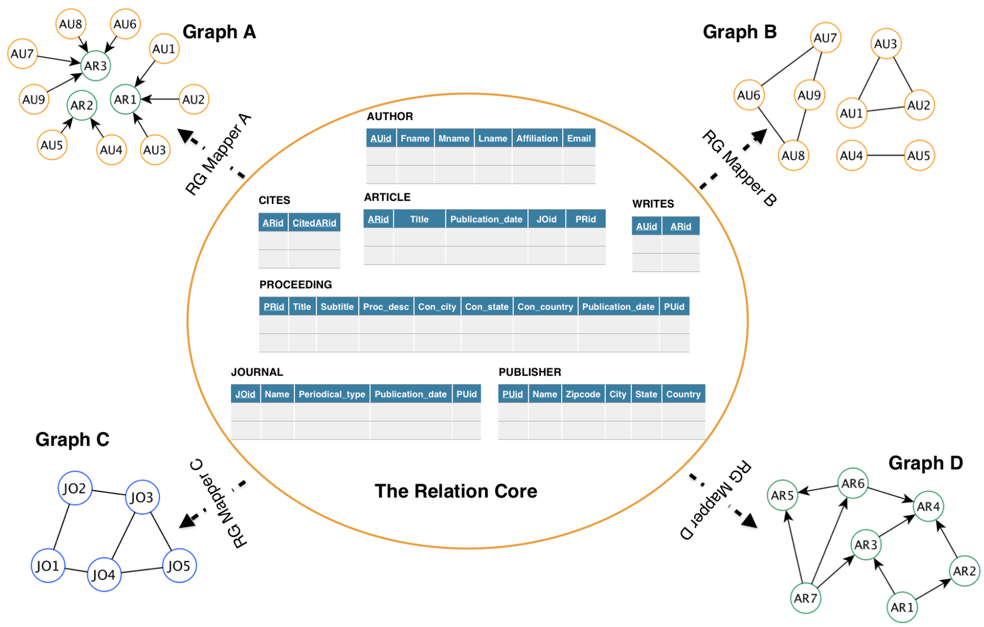


Figure 3.1: Overview of Data Model

- A **tuple** over R (or an R-*tuple* for short) is a mapping, $t : R \to \mathcal{D}$, with $t(A) \in dom(A)$ for all $A \in attr(R)$. We use $t(A)$ indicates the value that corresponds to the attribute A in tuple t.
- A **relation** over R (or an R-*relation* for short) is a finite set of R-*tuples*.
- A **relational core** C is a set of relation schemas, i.e., $C = \{R_1, R_2, \ldots, R_m\}$.

In the relation core, there are two types of domains: $\mathcal{D}_{id} \subseteq \mathcal{D}$ is a set of **identifier domains** and $\mathcal{D}_{va} \subseteq \mathcal{D}$ is a set of **value domains** with $\mathcal{D}_{id} \cap \mathcal{D}_{va} = \emptyset$ and $\mathcal{D}_{id} \cup \mathcal{D}_{va} = \mathcal{D}$. An identifier domain contains a set of entity identifiers. A value domain contains a set of permissible values. All identifier domains in $\mathcal{D}_{id}$ are **pairwise disjoint** (the reason will be described in Section 3.1.2). The following example illustrates these concepts of the relational core.

Example 3.1.1 The Association for Computing Machinery (ACM) is an organization for academic and scholarly interests in computing. It manages a large bibliographical network data. In the ACM bibliographical network, each article is written by one or more authors, an article is published in a conference proceeding or a journal, one article may cite a number of other articles, and each journal or conference proceeding is published by a publisher. Figure 3.2 shows a relational core for the ACM bibliographical network $ACM = \{AUTHOR, ARTICLE, PROCEEDING, JOURNAL, PUBLISHER, WRITES, CITES\}$. The underlined attributes represent primary keys and each directed arc represents a foreign key. Each relation schema has one or more attributes with an identifier domain. In this case, we have $\mathcal{D}_{id} = \{dom(AUid), dom(ARid), dom(CitedARid), dom(JOid), dom(PRid), dom(PUid)\}$.

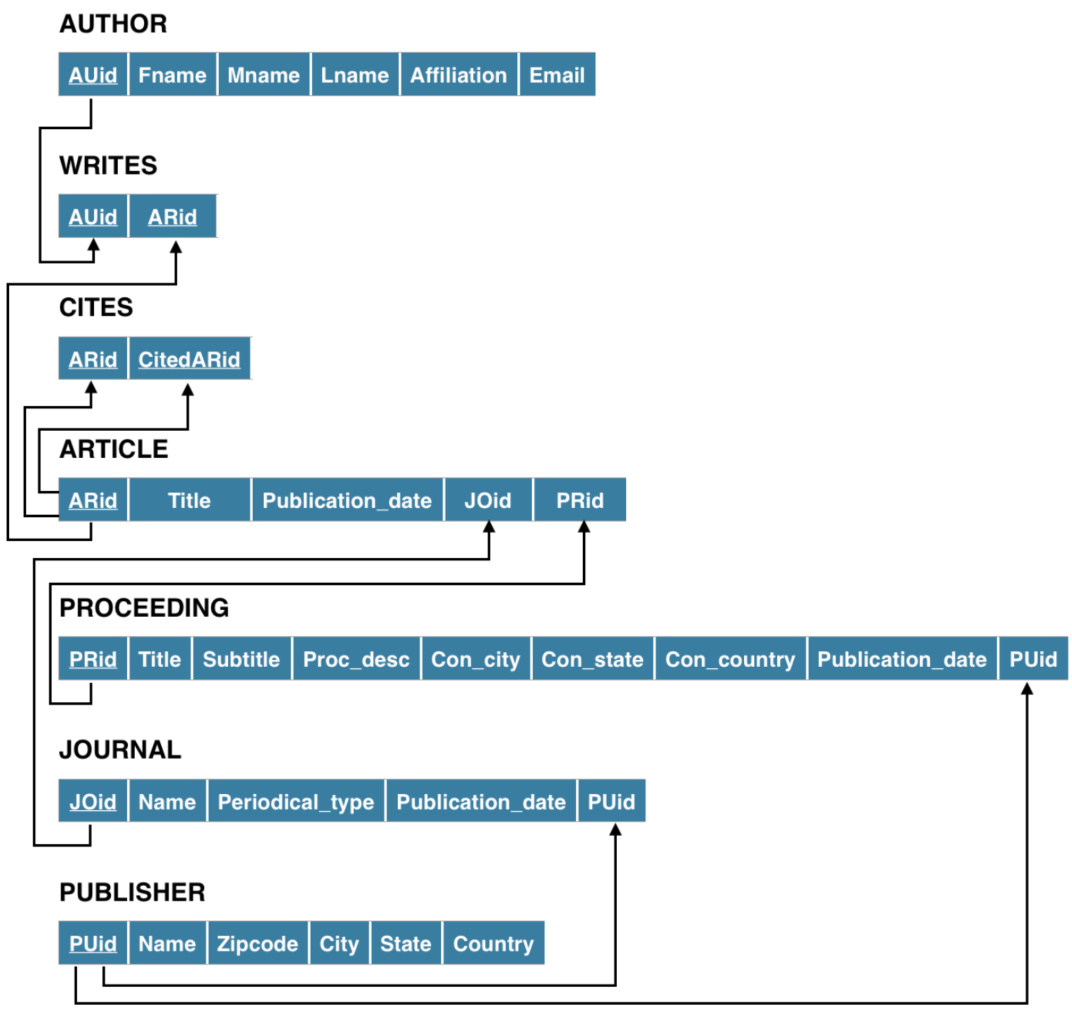


Figure 3.2: The Relational Core of ACM Bibliographical Network

3.1.2 Graphical Views

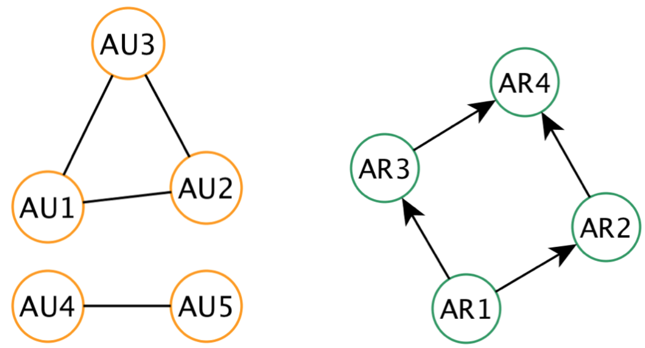
Based on a relational core, a number of graphical views can be established in the RG model. Each graphical view is a graph in which a vertex represents an entity and an edge represents a link between two entities. Each graph can be described by a graph schema. Informally, a graph schema describes what kinds of entities the vertices of a graph may represent and the connections of such entities represented by the edges. In this work, we use entity class to describe one kind of entities and link class to describe a type of connection between entities.

Formal definitions are presented as follows:

- An **entity class** $\mathcal{E}$ describes a set of (physical or abstract) entities that have the same behaviour and characteristics. In the RG model, each entity class $\mathcal{E}$ contains a set of entity identifiers from the same identifier domain.
- A **link class** $\mathcal{L}$ describes relationships among two (possible same) entity classes $\mathcal{E}_1$ and $\mathcal{E}_2$. For a link class $\mathcal{L}$ and any two entities $\mathcal{E}_1$ and $\mathcal{E}_2$, $\mathcal{L}$ is **symmetric** if it satisfies a condition: whenever $(\mathcal{E}_1, \mathcal{E}_2) \in \mathcal{L}$, then we must have $(\mathcal{E}_2, \mathcal{E}_1) \in \mathcal{L}$. A link class is **asymmetric** if it is not symmetric.
- A **graph schema** $\mathcal{G}$ consists of two entity classes and one link class, denoted by $\mathcal{G} = \langle \mathcal{E}_1, \mathcal{L}, \mathcal{E}_2 \rangle$, where the link class $\mathcal{L}$ is defined as $\mathcal{L} \subseteq \mathcal{E}_1 \times \mathcal{E}_2$. If $\mathcal{L}$ is symmetric, then graphs over this graph schema $\mathcal{G}$ are undirected graphs. Otherwise, graphs are undirected.
- A **graph** $G = (V, E)$ over $\mathcal{G} = \langle \mathcal{E}_1, \mathcal{L}, \mathcal{E}_2 \rangle$ consists of a set of vertices $\mathcal{E}_1 \cup \mathcal{E}_2$ and a set of edges $E \subseteq \mathcal{L}$.

A standard graph structure, $G = (V, E)$, consists of vertices and edges. V is a set of vertex identifiers and E is a set of vertex identifier pairs. Therefore, in our data model, only entity identifiers are stored in graphs. Other information are stored in the relational core. We also require all identifier domains in $\mathcal{D}_{id}$ must be **pairwise disjoint** so as to guarantee one vertex in a graph represent exactly one entity.

Example 3.1.2 In the ACM bibliographical network, we may have two entity classes – $\mathcal{E}_{au}$ for authors and $\mathcal{E}_{ar}$ for articles. $\mathcal{E}_{au}$ contains the entity identifiers in $dom(AUid)$ in the relation schema $AUTHOR$ and $\mathcal{E}_{ar}$ contains the entity identifiers in $dom(ARid)$ in the relation schema $ARTICLE$. For example, the article $AR1$ is written by three authors $AU1$, $AU2$ and $AU3$. The article $AR2$ is written by two authors $AU4$ and $AU5$. In Figure 3.3(a), we have an undirected graph over a graph schema $\mathcal{G} = \langle \mathcal{E}_{au}, \mathcal{L}_{coauthorship}, \mathcal{E}_{au} \rangle$ where $\mathcal{L}_{coauthorship}$ is symmetric and indicates that two entities are linked if they have co-authored at least one article. For another example, the article $AR1$ cites two articles $AR2$ and $AR3$. Both $AR2$ and $AR3$ cite the article $AR4$. In Figure 3.3.(b), we have a directed graph over a graph schema $\mathcal{G} = \langle \mathcal{E}_{ar}, \mathcal{L}_{citation}, \mathcal{E}_{ar} \rangle$ where $\mathcal{L}_{citation}$ is asymmetric and indicates that two entities are linked if one cites another one.



(a) Co-authorship Graph over $\langle \mathcal{E}_{au}, \mathcal{L}_{coauthorship}, \mathcal{E}_{au} \rangle$

(b) Citation Graph over $\langle \mathcal{E}_{ar}, \mathcal{L}_{citation}, \mathcal{E}_{ar} \rangle$

Figure 3.3: Graphs of ACM Bibliographical Network

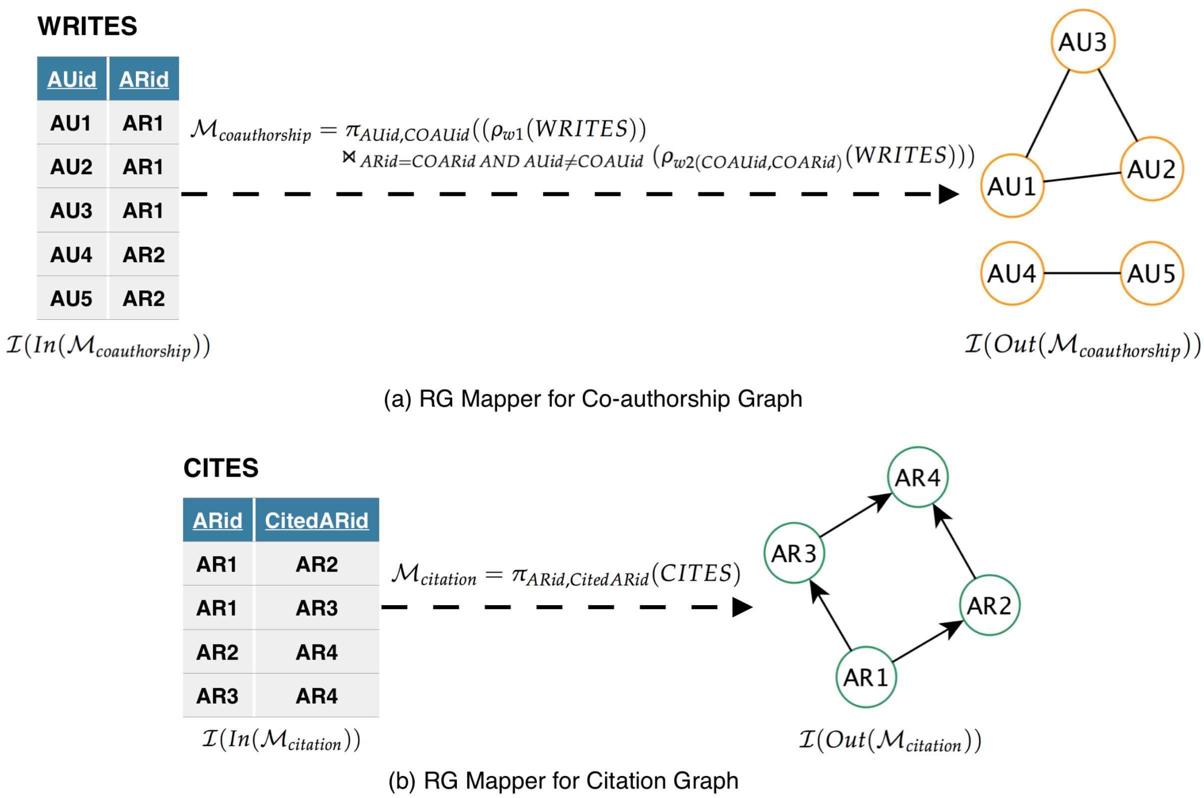
3.1.3 Relation-Graph Mappers

In the RG model, we define **relation-graph mappers** (RG mappers), each of which takes a set of relations as input and generate a graph as output.

We define the following related concepts for RG mappers.

- An **input schema** $In_{\mathcal{M}}$ is a set of relation schemas, $In_{\mathcal{M}} = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_m\}$. A set of relations over the relation schemas in $In_{\mathcal{M}}$ is denoted by $\mathcal{I}(In_{\mathcal{M}})$.
- An **output schema** $Out_{\mathcal{M}}$ is a graph schema, i.e.,$Out_{\mathcal{M}} = \langle \mathcal{E}_1, \mathcal{L}, \mathcal{E}_2 \rangle$. A graph over the graph schema is denoted by $\mathcal{I}(Out_{\mathcal{M}})$.
- An **RG mapper** $\mathcal{M}$, which is a mapping, maps a set of relations over an input schema $In_{\mathcal{M}}$ to a graph over an output schema $Out_{\mathcal{M}}$, i.e., $\mathcal{I}(In_{\mathcal{M}}) \rightarrow \mathcal{I}(Out_{\mathcal{M}})$.

Example 3.1.3 Figure 3.4.(a) presents two RG mappers $\mathcal{M}_{coauthorship}$ and $\mathcal{M}_{citation}$. In Figure 3.4.(a), the RG mappers $\mathcal{M}_{coauthorship}$ generates the co-authorship graph over the graph schema $\langle \mathcal{E}_{au}, \mathcal{L}_{coauthorship}, \mathcal{E}_{au} \rangle$ from a relation over the relation schema $WRITES$, so $In_{\mathcal{M}_{coauthorship}} = \{WRITES\}$ and $Out_{\mathcal{M}_{coauthorship}} = \langle \mathcal{E}_{au}, \mathcal{L}_{coauthorship}, \mathcal{E}_{au} \rangle$. In Figure 3.4.(b), another RG mappers $\mathcal{M}_{citation}$ generates the citation graph over the graph schema $\langle \mathcal{E}_{ar}, \mathcal{L}_{citation}, \mathcal{E}_{ar} \rangle$ from a relation over the relation schema $CITES$, so $In_{\mathcal{M}_{citation}} = \{CITES\}$ and $Out_{\mathcal{M}_{citation}} = \langle \mathcal{E}_{ar}, \mathcal{L}_{citation}, \mathcal{E}_{ar} \rangle$.



(a) RG Mapper for Co-authorship Graph

(b) RG Mapper for Citation Graph

Figure 3.4: RG Mappers of ACM Bibliographical Network, we use relational algebra to represent an RG mapper in this section.

3.2 Query Language

In this section, we present a query language that is based upon the RG model. Our query language, called **RG-SQL**, extends the traditional SQL (Structured Query Language) with the following main features..

- **graphical views** providing flexible choices for building graphs on-the-fly or materialising graphs.
- Incorporating **graph operators** to support common graph algorithms for network analytics, such as vertex centrality, community detection, reachability and shortest path.

Here, we discuss three types of graph operations which are ranking, clustering and path finding. We also demonstrate how such operators can be incorporated into SQL to provide a unified data analysis framework for relational analysis, network analysis or a mix of them.

Below is the basic syntax (SQL-style syntax) of graph queries in our query language (we will provide more details in the following subsections):

SELECT <attribute list>
FROM <graph operator>
WHERE <condition>;

- <attribute list> is a list of attribute names of a relation that contains the result generated by a graph operation.
- <graph operator> indicates which operator (RANK, CLUSTER or PATH) a user wants to use.
- <condition> in the WHERE clause is optional for ranking and clustering operations to construct a graph on-the-fly, but it is required for path finding operation to specify vertex condition.

3.2.1 Create Graphical Views

In our data model, graphs can be constructed over a relational core using RG mappers. Thus, graphs are supposed to be dynamic, i.e., graphs change if we modify the tuples of the relational core. In general, there are two approaches to specify graphs in our work.

Graphs On-the-fly The first approach is to create graphs on-the-fly. In this case, graphs are not persistently stored in the database, which provides us a flexible way to create small graphs or different subgraphs of a large one. For graphs that are created on-the-fly, they are stored in the main memory, so the I/O cost can be significantly reduced. However, this approach is also limited by the size of a graph and the size of available main memory. If a graph is too large, then it may not be able to fit into the main memory, and fails to be created on the fly. Another disadvantage of this approach is that it is inefficient for a frequently-used graph when a RG mapper is a complex query that is time-consuming to execute. The syntax of creating a graph on-the-fly is defined by:

SELECT <attribute list>
FROM <graph operator
WHERE <graph name> **IS** <graph type> **AS** (RG mapper);

<graph type> := UNGRAPH | DIGRAPH

If users want to create a graph on-the-fly, they need to specify the graph name, the graph type (UNGRAPH means undirected graph, DIGRAPH means directed graph) and the RG mapper in the **WHERE** clause. In *Example 3.2.1*, it shows how to create the citation graph on-the-fly, where the citation graph is generated by an RG mapper (SELECT ARid, CitedARid FROM CITES). Details about the VertexID, Value and RANK operator are given in the next subsection.

Example 3.2.1 The following citation graph is created on the fly.

```
SELECT VertexID, Value
FROM RANK (citation, indegree)
WHERE citation IS DIGRAPH AS
(
    SELECT ARid, CitedARid FROM CITES
);
```

Materialised Graphs The second approach is called graph materialisation which persistently creates a graph in the database. The same as materialised views in relational databases, incremental update is the technique that keeps the graph up-to-date [25]. This approach is efficient when a graph query needs to be executed multiple times, or a graph query provides results that can be further analysed. However, we need space to store materialised graphs. The syntax of creating a materialised graph is defined by:

CREATE <graph type > <graph name> **AS** (RG mapper);

<graph type> := UNGRAPH | DIGRAPH

Users can use the **CREATE** command to create a materialised graph in the database. As same as creating a graph on-the-fly, users are required to specify the graph type, the graph name and the RG mapper. We take the coauthorship graph and the RG mapper $\mathcal{M}_{coauthorship}$ mentioned in the previous section as an example to demonstrate the syntax of creating a materialised graph:

Example 3.2.2 The following creates a coauthorship materialised graph.

```
CREATE UNGRAPH coauthorship AS
(
    SELECT w1.AUid AS AUid, w2.AUid as CoAUid
    FROM WRITES as w1, WRITES as w2
    WHERE w1.ARid = w2.ARid AND w1.AUid != w2.AUid
);
```

If we do not need a materialised graph any more, we can use the **DROP** command to dispose of it. We define the following syntax of dropping a materialised graph along with an example of dropping the coauthorship graph.

DROP <graph type > <graph name>;

<graph type> := UNGRAPH | DIGRAPH

Example 3.2.3 The following drops the coauthorship materialised graph.

```
DROP UNGRAPH coauthorship;
```

3.2.2 Use Graph Operators

In our query language, graph operations are provided as building blocks in the **FROM** clause for expressing queries over graphs. We have incorporated three typical operations – ranking, clustering and path finding.

Ranking In network analytics, we are interested in **vertex centrality** which indicate the importance of vertices within a graph. A number of measures have been previously proposed to determine the importance of vertices such as degree, betweenness, closeness, pagerank and so forth [26]. We develop a graph operator **RANK** to specify the ranking operation with the following syntax:

RANK (<graph name>, <measure>)

<measure> := degree | indegree | outdegree |
betweenness | closeness | pagerank

Note that different measures support different graph types. When creating a graph, we are required to specify the type of the graph. We will check the measures with the graph type when running ranking operations on a graph. Table 3.1 shows all measures that have been incorporated into our query language, along with their supporting graph types.

Operator	Measures	Supporting Graph Types Undirected graph	Directed graph
RANK	degree	√	
	indegree		√
	outdegree		√
	betweenness	√	
	closeness	√	
	pagerank	√	√

Table 3.1: Measures of the RANK Operator

After running a ranking operation over a graph, the results are stored in a temporary table which consists of two attributes – ”VertexID” and ”Value”. The value of the ”VertexID” attribute in a tuple is an entity identifier of the graph. The value of the ”Value” arrtibute in a tuple is the ranking score of the vertex corresponding to the entity identifier in ”VertexID”. The results are sorted by a descending order of ”Value”. We can also add the **LIMIT** clause to return only the top k results. In the following, we show a query that is based on the data model of the ACM bibliographical network mentioned in the previous section.

Example 3.2.4 The following query is to find the top 3 influential articles according to their citation counts.

```
SELECT VertexID, Value
FROM RANK (citation, indegree)
WHERE citation IS DIGRAPH AS
(
    SELECT ARid, CitedARid FROM CITES
)
LIMIT 3;
```

Clustering A large number of clustering algorithms have been developed for solving problems in different application areas [26]. In network analytics applications, two typical clustering-related tasks are: **community detection** and finding **connected components**. In real-life networks, the distribution of edges normally is locally inhomogeneous, which means high concentrations of edges with special groups of vertices and low concentrations between these groups. This feature is called community structure [33]. In addition to finding community, we often want to find the biggest connected component or find all strongly connected components in a network. We develop a graph operator **CLUSTER** to specify a group of vertices by using algorithms for connected components and community detection. For algorithms, we use five keywords including CC for the algorithm of finding connected components [26], SCC for the algorithm of finding strongly connected components [26], GN for Girvan-

Newman algorithm [34], CNM for Clauset-Newman-Moore Algorithm [29] and MC for Peixoto's modified Monte Carlo Algorithm [42]. The syntax of the clustering operation is defined by:

CLUSTER (<graph name>, <algorithm>)

<algorithm> := CC | SCC | GN | CNM | MC

As same as the ranking operation, clustering algorithms support different graph types. Table 3.2 shows all algorithms along with their supporting graph types.

Operator	Algorithms	Supporting Graph Types Undirected graph	 Directed graph
CLUSTER	CC	√	√
	SCC		√
	GN	√	
	CNM	√	
	MC	√	√

Table 3.2: Measures of the CLUSTER Operator

The result generated by a clustering operation over a graph is stored in a temporary table which consists of three attributes – "ClusterID", "Size" and "Members". Users can add the **ORDER BY** clause with the "Size" attribute to get the biggest connected component or community. The value of the "Members" attribute in a tuple is an array of entity identifiers, which indicates who are in this tuple's cluster. Assume that we have already created a materialised graph called coauthorship mentioned in *Example 3.2.2*. *Example 3.2.5* shows how to find the biggest communities of authors in the ACM bibliographical network.

Example 3.2.5 The following query is to find the biggest communities that consist of authors who collaborate with each other to publish articles together.

```
SELECT ClusterID, Size, Members
FROM CLUSTER (coauthorship, GN)
ORDER BY Size DESC
LIMIT 1;
```

Path Finding A path is a sequence of pairwise disjoint vertices $V_1, \ldots, V_n$ where (V_i, V_{i+1}) is an edge for $i = 1, \ldots, n-1$. Finding paths is also one of typical tasks in network analytics and it includes two primary problems – **reachability** and **shortest path**. In addition, users often want to add more conditions on a path such as finding a path with a specific length or with a specific vertex in the middle of it. The syntax of **PATH** graph operator is defined by:

PATH (<graph name>, <path expression>)

<path expression> := **.** | V | <path expression>/ <path expression>|
<path expression>// <path expression>

where V is a vertex expression defined by conditions in the WHERE clause (refer to the basic syntax of graph queries at the beginning of Section 3.2) and "**.**" is the **do-not-care** symbol which indicates any vertex is allowed.

A path expression is **valid** if it contains a vertex expression in the first and last positions. In path expression, "/" represents one edge and "//" represents any number of edges. Table 3.3 shows some examples about path expression.

Operator	Path Expression
PATH	V1 /. /. / V2
	(paths between V1 and V2, where the length is 3)
	V1 // V2
	(paths between V1 and V2 with any length)
	V1/. / V2 /. / V3
	(V2 in the 3rd position of paths between V1 and V3, where the length is 4)
	V1 // V2 // V3
	(V2 in the middle of paths between V1 and V3 with any length)

Table 3.3: Examples of Path Expression

When using path finding operation, users are required to specify vertex expressions in the **WHERE** clause. A temporary table that stores the results of a path finding operation over a graph consists of three attributes – "PathID", "Length" and "Path". Users can add the **ORDER BY** clause with the "Length" attribute to get the shortest path and the "//" symbol is for reachability problem between two vertices. The value of "Path" attribute in a tuple is an array of entity identifiers, which demonstrates the sequence of vertices in the path. Still, assume that we already have the coauthorship materialised graph and we use *Example 3.2.6* and *Example 3.2.7* to illustrate queries about reachability and finding shortest path in the ACM bibliographical network.

Example 3.2.6 The following query is to find two authors V1 and V2, where V1 and V2 are connected by a path of any length, the author V1 is affiliated at ANU (Australian National University) and the author V2 is affiliated at Microsoft.

```
SELECT PathID, Length, Path
FROM PATH (coauthorship, V1//V2)
WHERE V1 AS
(
    SELECT AUid FROM AUTHOR
    WHERE Affiliation like '%ANU%'
) AND V2 AS
(
    SELECT AUid FROM AUTHOR
    WHERE Affiliation like '%Microsoft%'
);
```

Example 3.2.7 The following query is to find shortest paths between two authors V1 and V3, where in the middle of the shortest path there is an author V2 who is affiliated at Microsoft. Author V1 is affiliated at ANU and Author V3 is affiliated at NICTA (National ICT Australia).

```
SELECT PathID, Length, Path
FROM PATH (coauthorship, V1//V2//V3)
WHERE V1 AS
(
    SELECT AUid FROM AUTHOR
    WHERE Affiliation like '%ANU%'
) AND V2 AS
(
    SELECT AUid FROM AUTHOR
    WHERE Affiliation like '%Microsoft%'
) AND V3 AS
(
    SELECT AUid FROM AUTHOR
    WHERE Affiliation like '%NICTA%'
)
ORDER BY Length ASC;
```

3.3 Summary

In this chapter, we have presented our data model (RG model) and query language (RG-SQL). The RG model is a hybrid model with relations and graphs. It consists of a relational core, graphical views and relation-graph mappers (RG mappers). A relational core is similar to the relational data model in traditional relational databases and the entity identifiers from identifier domains are used to specify the vertices of graphs. Therefore, all identifier domains in the relational core must be pairwise disjoint so as to guarantee each vertex in a graph can only represent exactly one entity. An RG mapper is a query that is used to map a set of relations to one graph. In the RG model, a relational core provides a basis for a number of graphical views that are generated by using a number of RG mappers. Based upon the RG model, we propose a query language (RG-SQL) for data manipulation. RG-SQL extends traditional SQL with creating/dropping graphs, and conducting queries over graphs. Users can use RG-SQL to create graphs on-the-fly or materialised graphs. The ranking operation is to sort vertices in a graph according to certain measure of vertex centrality. The clustering operation is to find a group of vertices and the path finding operation is to find a sequence of vertices in a graph.

Chapter 4

Query Engine

In this chapter, we describe the query engine developed for the RG model and RG-SQL. As we develop our query engine with PostgreSQL, we follow the PostgreSQL concepts when describing the query processing and each component of the query engine. In Section 4.1, we first demonstrate how queries written in RG-SQL are processed in our query engine. In Section 4.2, we present the architecture of our query engine and give more details about its components. Then we propose some query optimisation strategies in Section 4.3. Section 4.4 gives a summary of the query engine.

4.1 Query Processing

Similar to relational query processing, a query written in the RG-SQL is processed to follow a parser-optimiser-executor pattern. An RG-SQL query created in the query console is first validated by the query parser and then converted into a plan tree. The query optimiser enumerates alternative plan trees, estimates their cost and determines the best plan tree for execution. A plan tree (refer to Section 4.2 for more details) consists of different types of operation nodes including graph operation nodes (rank operation, cluster operation and path operation) and other relational operation nodes (selection operation, join operation, aggregate operation). The query optimiser will extract graph operations from the plan tree and pass them to the graph executors.

For all graph operations, they are executed by three graph executors: (1) rank executor is for rank operations, (2) cluster executor is for cluster operations and, (3) path executor is for path operations. During these executions, the graph executors need to retrieve the graph data from the data storage to generate graphs and run algorithms over those graphs. After graph operations are executed, their corresponding executors will store the results into the data storage as the network analysis results.

After the query optimiser determines the best plan tree, the plan executor executes the plan tree by processing its operation nodes from the bottom to the top. During the execution, the plan executor needs to retrieve the network analytics results and the relational data from the data storage. After the execution, the plan executor will return the query result to the query console.

Next section will give more details about each component of our query engine.

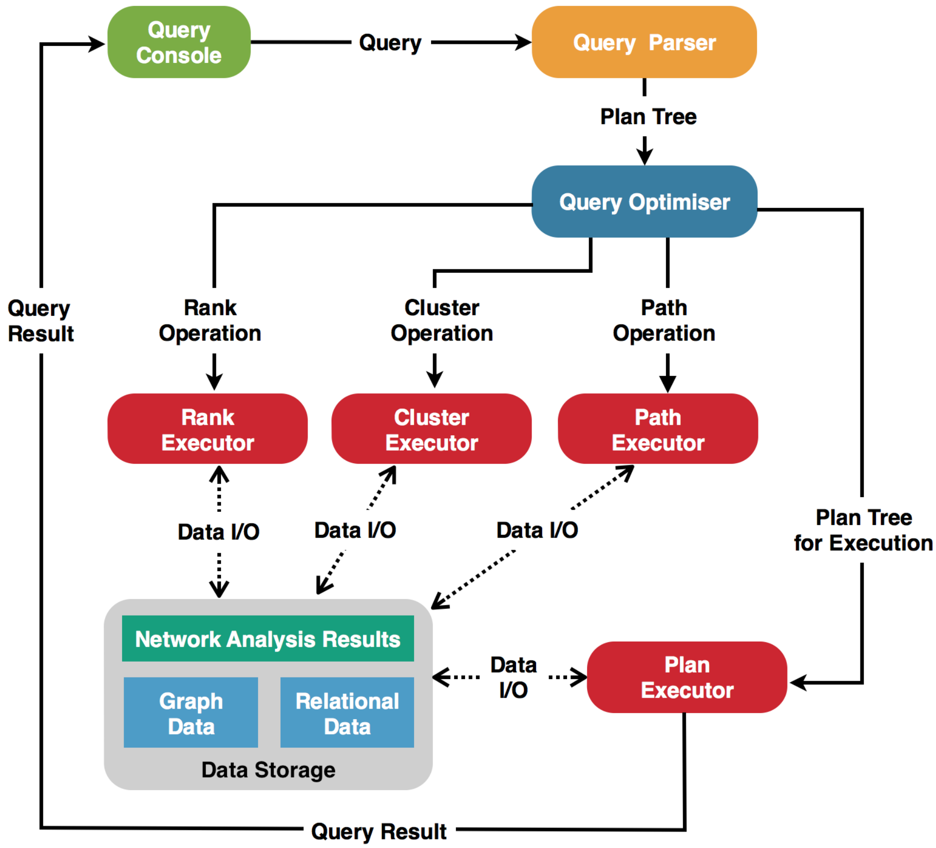


Figure 4.1: RG-SQL Query Processing

4.2 Architecture

Our query engine, called **RG Engine**, is built for processing RG-SQL queries that contain graph sub-queries (queries with graph operators) and relational sub-queries. The RG engine is developed in Python programming language with the official PostgreSQL client library – libpq [11]. We use Psycopg [17], the current mature wrapper for the libpq, as the PostgreSQL adapter for our query engine. Figure 4.2 shows the main components of the RG Engine.

Query Console

The query console is a user interface that allows users to submit RG-SQL queries. The same as traditional SQL queries, each RG-SQL query ends with a semicolon. The console also displays query result and error messages, such as graph type error, path expression error, and so on.

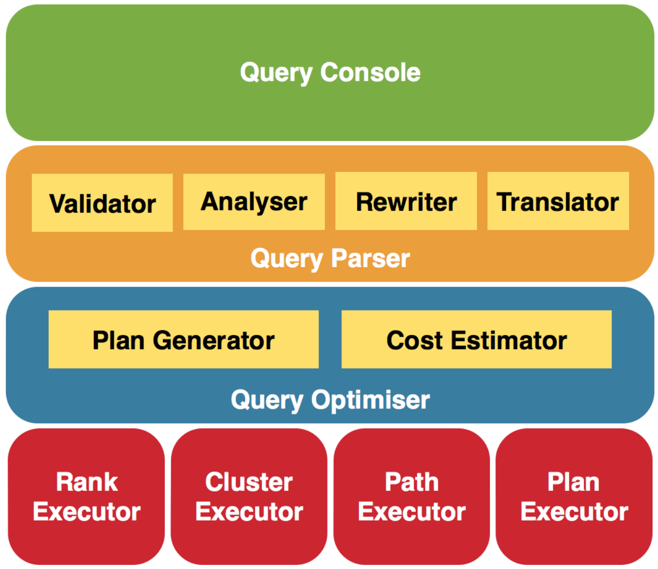


Figure 4.2: Architecture of the RG Engine

Query Parser

The query parser consists of four main sub-components – Validator, Analyser, Rewriter and Translator. Given an RG-SQL query, the validator first checks whether or not the query syntax is correct, such as checking the keyword's spelling, checking the number of parentheses, checking if path expressions are in correct format and so forth. Then, the validator is involved with the system catalog to validate the query. The system catalog is the place where PostgreSQL stores schema metadata, such as information about tables, attributes, operators, data types and other internal information [18]. We add a schema metadata about materialised graphs into the system catalog – the pg_matgraph. The following describes some typical query validation tasks:

- To check whether or not the graphs and tables of the query are registered in the system catalog. The corresponding schema metadata contain the pg_matgraph, the pg_table, the pg_matviews and the pg_views.

- To ensure that the attribute references are correct. The corresponding schema metadata is the pg_attribute.

- To examine if the operators used in the query are consistent with data types. The corresponding schema metadata contain the pg_operator and the pg_type.

After a query is validated, the analyser starts to differentiate graph sub-queries and relational sub-queries. There will be a query tree that indicates the query execution order (refer to Figure 4.3, queries at the bottom will be executed first). In Figure 4.3, the "Graph Sub-query 1" retrieves a materialised graph and the "Graph Sub-query 2" with a relational sub-query retrieves a graph that is created on-the-fly.

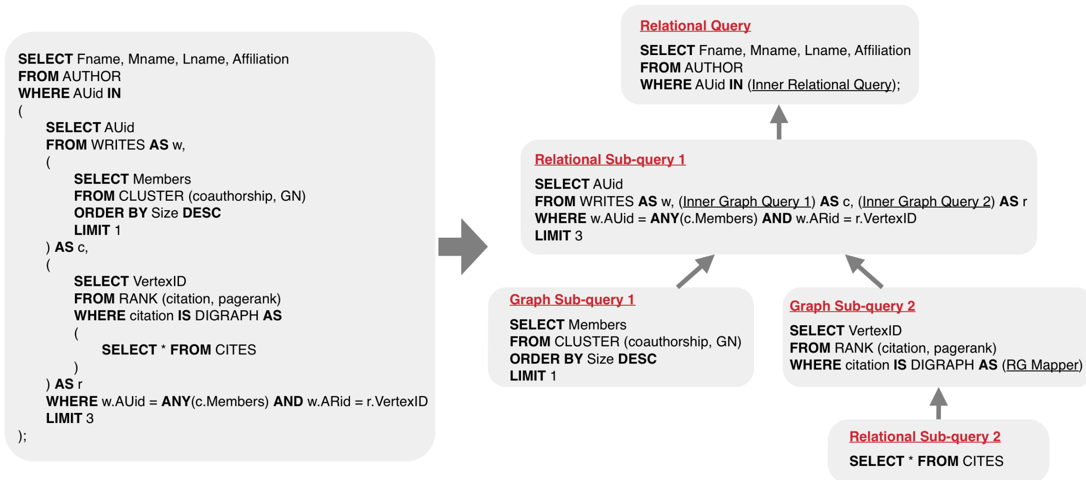


Figure 4.3: Query Tree Example for a Query of the ACM Bibliographical Network

For all the graph sub-queries, the rewriter replaces the graph operators with some specific table names. These table names will be used for temporary tables to store results after executing graph operations. For example, the ”Graph Sub-query 1” in Figure 4.3 will become ”SELECT Members FROM *cluster_coauthorship_1* ORDER BY Size DESC LIMIT 1;”. These table names follow a specific format:

<graph operator>_<graph name>_<graph operator ID>

<graph operator>:= rank | cluster | path
<graph name> is the name of the graph stored in the data storage.
<graph operator ID> corresponds to the order that graph operators occur in the query.

In our query engine, the text string of graph operators and the specific table names are stored in the data dictionary. If a graph operator contains a graph that is created on-the-fly, then the graph operator and its corresponding relational sub-query will be rewritten to one specific table name. For example, the ”Graph Sub-query 2” and the ”Relational Sub-query 2” in Figure 4.3 will become ”SELECT VertexID FROM *rank_citation_2*;”.

After all these steps, the translator converts the query into an internal format of the query (i.e. a plan tree) that will be passed on to the query optimiser for optimisation [18]. A plan tree can be represented by a relational algebra expression.

Query Optimiser

In general, what the query optimiser does are: (1) enumerating alternative plan trees based on the plan tree that is received from the query parser (done by the Plan Generator); (2) estimating the cost for the alternative plan trees (done by the Cost Estimator); (3) choosing the plan tree with the lowest cost for execution.

In order to identify alternative plan trees (typically a subset of all possible plan trees), one important method used by a query optimiser is using heuristic rules that transform a relational algebra expression (RA expression) into another equivalent-but-more-efficient RA expression. Some typical transformation rules include: to deconstruct conjunctive select operations into a sequence of individual selection, to combine selections and cross-products into joins, to push selections and projections ahead of joins and so forth [20][45]. After identifying the alternative plan trees, the query optimiser estimates costs of each plan tree in terms of disk page fetches (I/Os) and CPU time [1]. Then it determines the best plan tree for execution. There is one more thing: the query optimiser extracts all graph operations from the execution plan tree, passes them to the three graph executors.

Graph Operation Executors

All graph operations will be executed by three graph operation executors according to their operation types, which rank executor is for rank operations, cluster executor is for cluster operations and path executor is for path operations. For the graph operation executors, we use three graph analysis tools as algorithm support, including SNAP [21], NetworkX [16] and Graph-tool [7]. Based on the performance evaluation for these three graph analysis tools (refer to Section 5.2), we make decisions about algorithm support as follows:

- Rank Executor: choose SNAP to support algorithms for four ranking measures (i.e. degree, indegree, outdegree and pagerank) and Graph-tool to support algorithms for other two ranking measures (i.e. closeness and betweenness).

- Cluster Executor: choose SNAP to support four clustering algorithms (i.e. finding connected components [26], finding strongly connected components [26], the Girvan-Newman algorithm [34] and the Clauset-Newman-Moore algorithm [29]) and Graph-tool to support the Monte Carlo algorithm [42].

- Path Executor: choose NetworkX to support the path finding algorithm.

Table 4.1 shows the methods that are used in our graph operation executors. More details about those methods refer to the reference manuals of these graph analysis tools[2]. After executing the corresponding operations, three graph operation executors will store the results into temporary tables with specific table names (mentioned in the Query Parser). These temporary tables will be stored in the data storage as the network analysis results before the query processing terminates. The rank executor will first sort the results according to the measure values and then store the results into a table that consists of two columns – VertexID and Value. Likewise, the cluster executor will store the results into a table with three columns (i.e. ClusterID, Size and Members) and the path executor will create a table that also consists of three columns (i.e. PathID, Length and Path).

Algorithms	Methods	Tools
Degree	GetDegreeCentr( )	SNAP
Indegree	GetNodeInDegV( )	
Outdegree	GetNodeOutDegV( )	
Pagerank	GetPageRank( )	
Connected Component	GetWccs( )	
Girvan-Newman	CommunityGirvanNewman( )	
Clauset-Newman-Moore	CommunityCNM( )	
Betweenness	centrality.betweenness( )	Graph-tool
Closeness	centrality.closeness( )	
Monte Carlo	community.minimize_blockmodel_dl( )	
Path Algorithm	all_simple_paths( )	NetworkX

Table 4.1: Algorithm Support

Plan Executor

The basic idea of the plan executor is to execute the plan tree chosen by the query optimiser, to extract the required set of tuples, and to return the tuples as a query result to the query console. The plan tree is a pipelined demand-pull graph with different types of operation nodes and these nodes will be recursively processed by the plan executor [18]. The bottom-level nodes produce tuples as the input for the upper-level nodes. In general, the bottom-level nodes often relate to selection and projection operations which require the executor to scan physical tables (e.g. sequential scan for non-index tables and index scan for tables with index attributes) and the upper-level nodes often relate to join operations (e.g. nested-loop, merge join and hash join). There are other special-purpose operation nodes, such as sorting and aggregate operations [1]. Figure 4.4 shows an example about how an plan tree is processed for a query to find the affiliations of top 10 influential authors in the co-authorship network.

[2]SNAP's manual: http://snap.stanford.edu/snappy/doc/reference/index-ref.html;
NetworkX's manual: http://networkx.github.io/documentation/networkx-1.9.1/;
Graph-tool's manual: http://graph-tool.skewed.de/static/doc/index.html

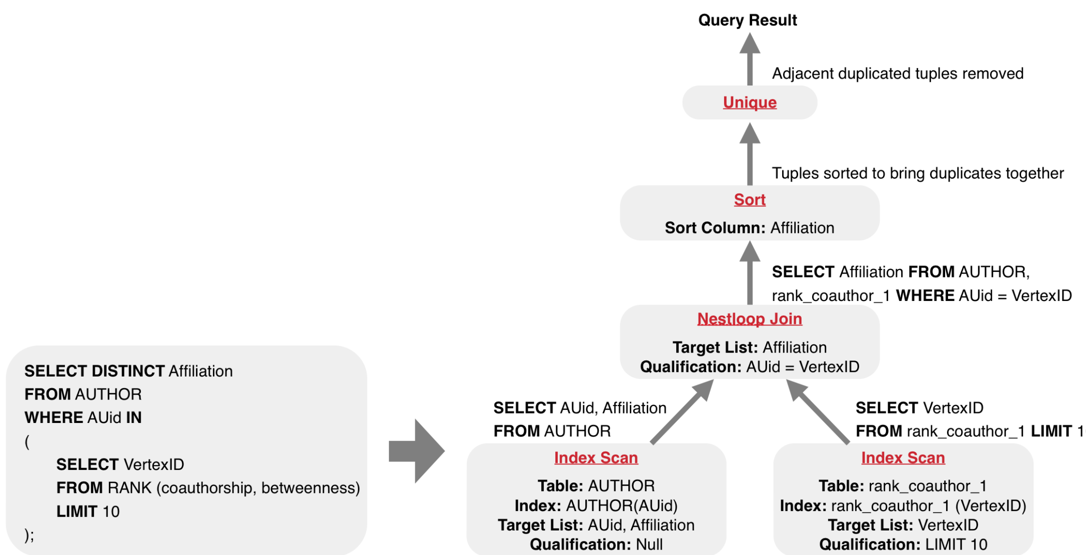


Figure 4.4: Plan Tree Processing for a Query of the ACM Bibliographical Network

4.3 Query Optimisation

In this section, we propose some query optimisation strategies for our query optimiser. As we adopt the query optimiser of PostgreSQL in our query engine, the specific optimisation techniques of PostgreSQL have already been used in our query engine, such as the transformation rules for relational algebraic equivalence, the genetic optimisation algorithm for searching alternative plan trees and so forth [18]. However, our RG-SQL queries may have graph sub-queries and relational sub-queries. How to optimise those graph sub-queries and relational sub-queries in a unified framework is the focus of this section. We divide our optimisation strategies into two groups – Sub-query equivalence and Query caching.

Sub-query equivalence A complex RG-SQL query always contains a number of graph sub-queries and these graph sub-queries often contain a number of relational sub-queries. Figure 4.5 shows a query tree example for a path finding query. In Figure 4.5, the relational sub-queries 1,2 3 are very similar, which are to select author identifiers from the AUTHOR relation. Moreover, for the relational sub-query 1 and the relational sub-query 3, the results of them are very close because many authors who work in NICTA are researchers in ANU.

The basic idea of sub-query equivalence is to decompose an RG-SQL query Q into a set of sub-queries $\{q_1, q_2, \ldots, q_n\}$, i.e. $Q \Rightarrow \{q_1, q_2, \ldots, q_n\}$. Then we reduce or rewrite the equivalent sub-queries to make the sub-query set smaller so as to improve efficiency.

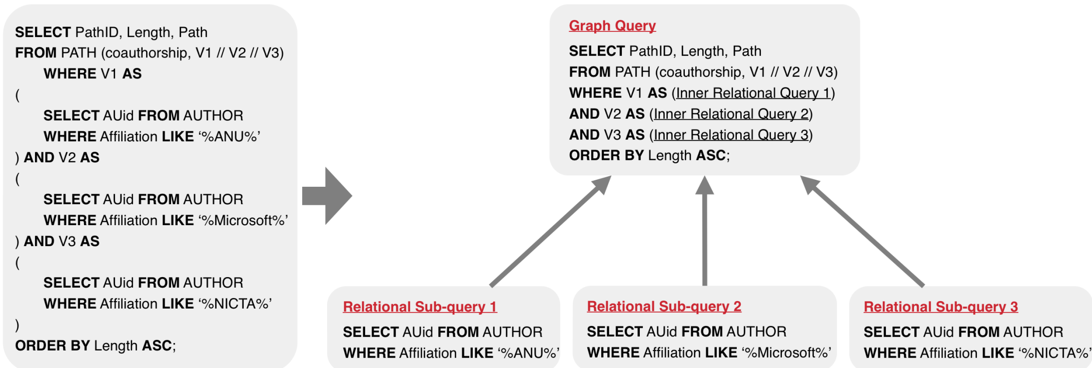


Figure 4.5: Query Tree Example for Sub-query Equivalence

Query caching Similar to some existing works for caching results of relational queries [27] [43], we can also cache the query results so as to avoid repeated computation. Given a complex RG-SQL query, its sub-queries can be transformed into a number of equivalent queries using different cached results and then this revised query is fed to the query optimiser to generate an optimal execution plan. However, there are some issues that need to be solved during the implementation including:

- Cache replacement strategy: we need to decide what kind of caches should be replaced when the cache space is full. Should we replace the caches that are the least recently used, or the caches that are the least frequently used, or the caches that require the largest cache space?

- Cache update strategy: we need to decide how to update the outdated caches. Should we update the caches once their base relations are changed (immediate update), or according to certain periods (periodical update), or when the caches are on demand (on-demand update)?

- Query matching strategy: we need to decide the requirements for two queries that can be considered as equivalent queries. If two queries are exactly the same, they certainly are equivalent. How about one query contains another query or two queries are overlapped? In these situations, can we still reuse the cached results?

4.4 Summary

In this chapter, we have described the RG-SQL query processing and the architecture of the RG engine. RG-SQL queries typically go through a parser-optimiser-executor pattern in the query engine. In the query parser of the RG engine, we have a validator to check and validate queries, a analyser to differentiate graph sub-queries and relational sub-queries, a rewriter to rewrite all graph sub-queries and a translator to convert queries into plan trees. Given a plan tree from the query parser, the query optimiser enumerates alternative plan trees, estimates their cost and determines the plan tree with the lowest cost to be executed. Then three graph operation executors execute the graph operations extracted from the execution plan tree and the plan executor processes each operation nodes of the plan tree from bottom to top. At last, the plan executor returns the query result to the query console. In addition, we also propose two query optimisation strategies for RG-SQL queries including sub-query equivalence and query caching.

Since the RG engine is developed with the PostgreSQL, it takes advantage of the existing PostgreSQL components to process queries including the query parser, the query optimiser and the plan executor. The source code of the RG engine refers to https://gitlab.com/RG_Framework/RG_Engine. We extend the PostgreSQL components with capability of processing RG-SQL queries, but we have not yet incorporated those query optimisation strategies with the RG engine. We take the implementation for query optimisation as one of our future work.

Chapter 5

Performance Evaluation

In this chapter, before showing the results of our performance evaluation experiments, we first describe our experimental environment including the hardware and software information in Section 5.1. We conduct two experiments in this chapter. The first one, in Section 5.2, is about the performance of the graph analysis tools that we use as the RG engine's algorithm support (i.e. SNAP [21], NetworkX [16], Graph-tool [7]). In Section 5.3, the second experiment, we compare our RG engine with the query engines of a relational database (PostgreSQL [19]) and a graph database (Neo4j [14]) through running different types of queries. A summary is given in Section 5.4.

5.1 Experimental Environment

Hardware Information

All of our experiments were performed on the Dell Optiplex 9020 desktop computer with the Intel(R) Core(TM) i7-4790 CPU 3.6GHz 8 cores processor, 16 GB of memory and the 256GB SAMSUNG SSD PM851 disk.

Software Information

The experiment-related software information is presented in Table 5.1.

Operating System	Ubuntu 14.04 LTS with Linux kernel 3.16.0-50 generic
Programming Language	Python 2.7.6
Relational Database	PostgreSQL 9.4.4
Graph Database	Neo4j community 2.2.5
Graph Analysis Tools	Snap.py 1.2 NetworkX 1.10 Graph-tool 2.9
PostgreSQL Adapter	psycopg2 2.6.1
Time Measure Package	timeit 2.6
Memory Measure Package	psutil 3.2.1

Table 5.1: Software Information

5.2 Performance of Graph Analysis Tools

As we mentioned in Section 4.2, for the graph operation executors, we use three graph analysis tools as algorithm supports, including SNAP [21], NetworkX [16] and Graph-tool [7]. Table 5.2 shows that all three graph analysis tools can support the first six algorithms including "Degree", "PageRank", "Betweenness", "Closeness", "Connected Component" and "Strongly Connected Component". However, for the other five algorithms, each algorithm can be supported by only one tool. Therefore, we choose SNAP to support the "Girvan-Newman" and "Clauset-Newman-Moore" algorithms, Graph-tool to support the "Monte Carlo" algorithm and NetworkX to support path finding algorithms.

	Algorithms	SNAP	NetworkX	Graph-tool
Ranking	Degree	$\surd$	$\surd$	$\surd$
	PageRank	$\surd$	$\surd$	$\surd$
	Betweenness	$\surd$	$\surd$	$\surd$
	Closeness	$\surd$	$\surd$	$\surd$
Clustering	Connected Component	$\surd$	$\surd$	$\surd$
	Strongly Connected Component	$\surd$	$\surd$	$\surd$
	Girvan-Newman	$\surd$	–	–
	Clauset-Newman-Moore	$\surd$	–	–
	Monte Carlo	–	–	$\surd$
Path Finding	Shortest Path	$*$	$\surd$	$***$
	Path with Specific Length	$**$	$\surd$	–

Note:
$*$ SNAP has the snap.GetShortPath( ) method but only returns the length of the path.
$**$ SNAP has the snap.GetNodesAtHop( ) method but only returns vertex identifiers of the destination vertices.
$***$ Graph-tool has the graph_tool.topology.shortest_path( ) but only returns one of all shortest paths.

Table 5.2: Algorithm Support of Graph Analysis Tools

We have first conducted an experiment to evaluate the time performance and memory performance of the three graph analysis tools through running the first six algorithms. For the experiment input, we used the graph generator of SNAP to create twelve Erdos-Renyi random graphs [5] [31], rather than graphs of a specific type of network. Table 5.3 shows the details of these Erdos-Renyi random graphs.

In the experiment, we ran each of these six algorithms over the twelve random graphs using the three graph analysis tools. Note that Graph-tool performs some algorithms (e.g. PageRank, Betweenness, Closeness) on multi-core architectures, which allows parallel computation [4]. However, SNAP and NetworkX do not support multi-core architectures. Therefore, we compare SNAP and NetworkX both with Graph-tool (using 1 core) and Graph-tool (using 4 cores). For the time performance evaluation, we

	Number of Vertices	**Number of Edges**	**Size (KB)**
Graph 1	100	200	1
Graph 2	100	1,000	6
Graph 3	100	5,000	29
Graph 4	500	1,000	8
Graph 5	500	5,000	38
Graph 6	500	25,000	189
Graph 7	2,500	5,000	46
Graph 8	2,500	25,000	228
Graph 9	2,500	125,000	1,100
Graph 10	12,500	25,000	256
Graph 11	12,500	125,000	1,300
Graph 12	12,500	625,000	6,400

Table 5.3: Erdos-Renyi Random Graphs

have run each algorithm five times and taken the average time for plotting. The average time is the sum of graph constructing time and algorithm computation time. For the memory performance evaluation, we also have run each algorithm five times, taken the peak value of each time as the memory consumption, and taken the average memory consumption for plotting.

Figure 5.1 shows the time performance comparison of the graph analysis tools. Note that the value of the Y axis of Figure 5.1.(3) and Figure 5.1.(4) is scaled in logarithm. Based on the plots in Figure 5.1, we have the following observations:

- For the algorithms about degree, page rank, connected components and strongly connected component), SNAP has the better time performance than NetworkX. This is mostly because the core library of SNAP is a C/C++ library and NetworkX is a pure Python implementation, which in general is known to be substantially slower than C/C++ [23] [10].

- However, although Graph-tool use a pure C/C++ library, it requires more time than SNAP when running the algorithms mentioned in the last bullet point. This is because Graph-tool spends more time on constructing graphs (refer to Appendix B for details). When constructing graphs, Graph-tool always creates vertices starting from ID 0. Simply speaking, if Graph-tool constructs a graph that only consists of one vertex with an identifier 100, it will create 101 vertices from ID 0 to ID 100. So when using Graph-tool to construct graphs, we need to create dictionaries that map vertex identifiers with the Graph-tool internal IDs. Because of the dictionary operations, Graph-tool requires more time for constructing graphs than the other two graph analysis tools.

- In terms of algorithms about betweenness and closeness, despite more graph constructing time, Graph-tool (4 Cores) takes advantage of its multi-core architectures to achieve better performance, especially in large graphs.

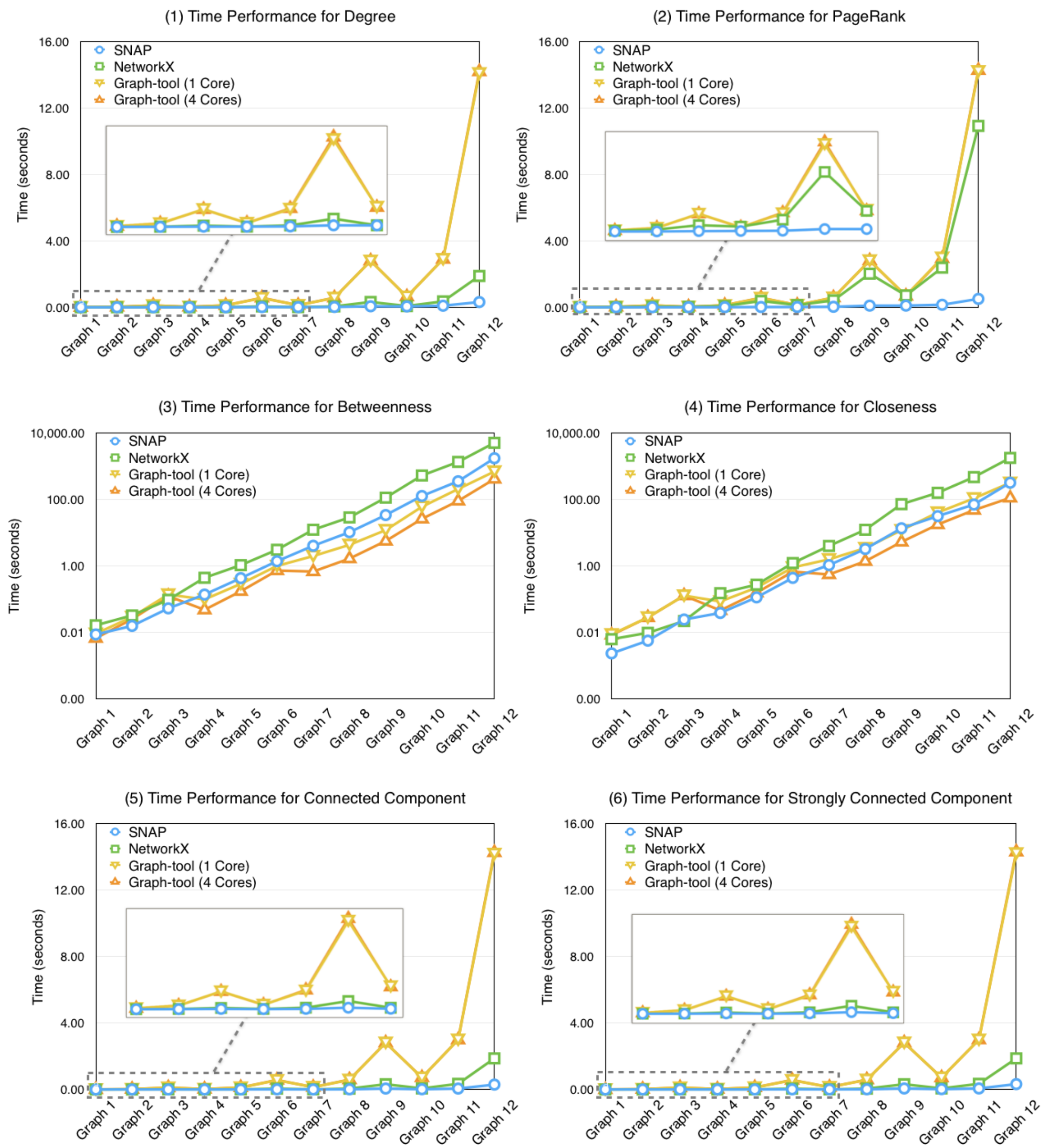


Figure 5.1: Time Performance of the Graph Analysis Tools

Figure 5.2 is about the memory performance comparison of the graph analysis tools. From Figure 5.2.(3) and Figure5.2.(4), we can conclude that Graph-tool's better time performance of betweenness and closeness algorithms comes at the cost of memory required during compilation. Due to different implementations, the memory performance varies among the graph analysis tools. Overall, SNAP has a better memory performance in this experiment.

Based on the experimental result above, we choose SNAP to support algorithms for degree, page rank, connected component, and strongly connected component. We choose Graph-tool to support algorithms for betweenness and closeness.

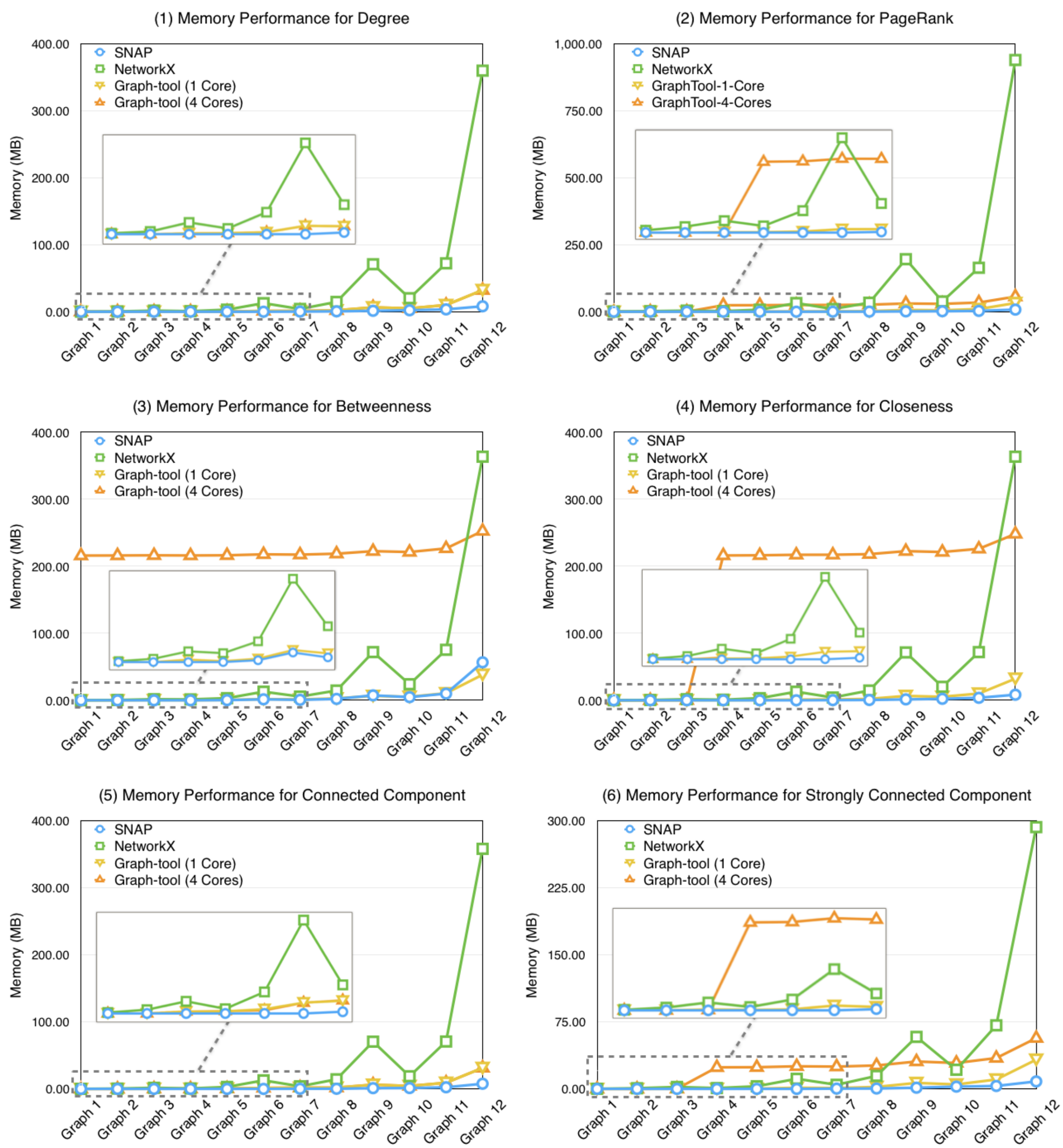


Figure 5.2: Memory Performance of the Graph Analysis Tools

5.3 Performance of the RG Engine

In this experiment, we set up different types of queries over three datasets. Through processing these queries, we compare our RG engine with the query engines of a relational database (PostgreSQL) and a graph database (Neo4j). We first introduce the three datasets used in the experiment in Section 5.3.1. In Section 5.3.2, we describes the queries processed by the query engines. Section 5.3.3 presents the experimental results about processing these queries.

5.3.1 Datasets

In this experiment, we used three datasets: (1) ACM bibliographical network (ACM network)[3] , (2) Stack Overflow network (ST network)[4] and (3) Twitter network (TW network)[5]. The data in these three datasets can be described as follows:

- In the ACM network, each article is written by one or more authors, an article is published in a conference proceeding or a journal, one article may cite a number of other articles, and each journal or conference proceeding is published by a publisher (refer to the ER diagram of Appendix A).
- In the ST network, each question and each answer is posted by one user, an answer is accepted for one question as the accepted answer, one question can have zero or more answers and one question can be labelled by zero or more tags (refer to the ER diagram of Appendix A).
- In the TW network, each tweet is posted by one user, a tweet can be labelled by zero or more tags, a tweet can mention zero or more users and a user can follow zero or more other users (refer to the ER diagram of Appendix A).

The data of the ACM network and the ST network are both in the XML format and the data of the TW network is in the TXT format. We write a Python program (refer to https://gitlab.com/RG_Framework/Data_Import) to transform the data into the PostgreSQL relational database (refer to the relation schemas of Appendix A) and follow the instruction [9] [8] of the Neo4j official website to transform data into the Neo4j. Table 5.4 shows the information about the datasets.

5.3.2 Queries

Based on the three datasets mentioned in the previous subsection, we set up 12 queries that can be divided into three categories. Table 5.5 shows more details about the queries. In Table 5.5, Queries 1 – 3 are relational queries including join operations, sorting operations, aggregate operations and set operations. Queries 4 – 10 are about

[3]Provided by ACM Digital Library (http://dl.acm.org/)
[4]Provided by Stanford Network Analysis Platform (http://snap.stanford.edu/proj/snap-icwsm/)
[5]Provided by Haewoon Kwak (http://an.kaist.ac.kr/traces/WWW2010.html) and Stanford Network Analysis Platform (http://snap.stanford.edu/data/twitter7.html)

	Raw Data Size	Number of Vertices in Neo4j	Number of Edges in Neo4j	Number of Records in PostgreSQL
ACM Network	14.9 GB (XML)	1,128,243	2,488,849	PUBLISHER : 50 JOURNAL : 128 PROCEEDING : 6,421 ARTICLE : 337,006 AUTHOR : 784,638 WRITES : 932,400 CITES : 1,212,894
ST Network	30.6 GB (XML)	21,713,109	31,747,662	QUESTION : 7,990,787 ANSWER : 13,684,117 TAG : 38,205 LABELLED_BY : 13,466,686
TW Network	29.7 GB (TXT)	13,250,196	264,368,797	TWEET : 10,762,104 TAG : 210,121 TW_USER : 2,277,971 FOLLOW : 259,602,970 MENTIONED_IN : 3,108,776 LABELLED_BY : 1,657,051

Table 5.4: Dataset Characteristics

some typical network analytics tasks including pattern matching, triangle counting, pagerank centrality, finding connected components, path finding and community detection. Queries 11 – 12 are advanced queries that combine two different types of network analytics tasks together, in which Query 11 combines pagerank centrality with finding connected components and Query 12 combines pagerank centrality with path finding. In terms of how to write these queries in SQL, RG-SQL and Cypher (Neo4j's query language), please refer to the Appendix C.

5.3.3 Experimental Results

We have evaluated all these experiment queries using 3 query engines. However, as shown in Table 5.6, PostgreSQL cannot process Queries 6 – 12 and Neo4j cannot process Queries 10 – 12. This is due to the limited expressive power of SQL and Cypher: we cannot use SQL to express Queries 6 – 12 and Queries 10 – 12 cannot be expressed using Cypher. One advantage of our work is that all these queries can be expressed in RG-SQL and processed by the RG engine.

To compare the RG engine with PostgreSQL and Neo4j for Queries 1 – 5 and compare the RG engine with Neo4j for Queries 6 – 10, we have conducted an experiment to evaluate their time performance. Note that for Query 6 and Query 7, Neo4j needs to use an extension called Neo4j Mazerunner that extends Neo4j to run network analytics algorithms at scale with Hadoop HDFS and Apache Spark [12]. For the time

Join Operation + Sorting Operation		
Query 1	ST Network	Show the question id, the owner id and the tag label of top 10 questions that have the most view count.
Join Operation + Sorting Operation + Aggregate Operation		
Query 2	ST Network	Show the top 5 answerers and their latest reputation score in an descending order based on the number of their answers that accepted by questions.
Join Operation + Sorting Operation + Aggregate Operation + Set Operation		
Query 3	ACM Network	Show the number of articles of each journal and proceeding along with the journal name and the proceeding title in a descending order.
Pattern Matching		
Query 4	TW Network	Recommend 10 twitter users for Jack who currently does not follow these users but Jack follows somebody who are following them.
Triangle Counting		
Query 5	ACM Network	Count the number of triangles of the co-authorship network.
PageRank Centrality		
Query 6	ACM Network	Find the top 10 influential authors according to the pagerank centrality in the co-authorship network.
Connected Component		
Query 7	ACM Network	Count the number of connected components of the co-authorship network.
Path Finding		
Query 8	ACM Network	Find paths with length less than 2, which connect two author V1 and V2 in the co-authorship network where author V1 is affiliated at ANU and author V2 is affiliated at UNSW.
Shortest Path		
Query 9	ACM Network	Find a shortest paths between two authors Michael Norrish and Kevin Elphinstone in the co-author network.
Community Detection		
Query 10	ST Network	Find a group of tags that they are often used together to label a question.
PageRank Centrality + Connected Component		
Query 11	ACM Network	According to the pagerank centrality, find the top 3 authors of the biggest collaborative community in the co-authorship network.
PageRank Centrality + Path Finding		
Query 12	ACM Network	According to the pagerank centrality, show how the top 2 authors connect with each other in the co-authorship network.

Table 5.5: Queries Used in Our Experiment

	PostgreSQL	**RG Engine**	**Neo4j**
Query 1	$\surd$	$\surd$	$\surd$
Query 2	$\surd$	$\surd$	$\surd$
Query 3	$\surd$	$\surd$	$\surd$
Query 4	$\surd$	$\surd$	$\surd$
Query 5	$\surd$	$\surd$	$\surd$
Query 6	–	$\surd$	$\surd$
Query 7	–	$\surd$	$\surd$
Query 8	–	$\surd$	$\surd$
Query 9	–	$\surd$	$\surd$
Query 10	–	$\surd$	–
Query 11	–	$\surd$	–
Query 12	–	$\surd$	–

Table 5.6: Queries Processed by three Query Engines

performance evaluation, we have run each query 5 times and taken the average time for plotting. For Queries 1 – 5 and Queries 8 – 9, once a query is submitted, we started to record the time until the result of each query was returned. For Query 6 – 7, as Neo4j is required to send an HTTP GET request to the Mazerunner extension to begin a network analytics algorithm, the time of Neo4j for these two queries is the sum of the request processing time and the query processing time. As shown in Figure 5.3, for Queries 1 – 5, the RG engine has nearly the same time performance with PostgreSQL since our query engine is developed with the official PostgreSQL library – libpq (refer to Section 4.2). The RG engine can achieve better performance for most queries except Queries 4, 8 and 9. This is mostly due to the following reasons.

- For Queries 1 – 3, which are the relational queries, the RG engine achieves better performance by taking advantage of the query optimisation techniques from relational databases. For Query 5, triangle counting, it has already been proved that relation databases can perform the triangle counting task very efficiently through expressing a three-way self-join [36].

- For Queries 6 – 7, as Neo4j needs to rely on the Mazerunner extension, it requires more time on sending the algorithm requests and waiting for the request completion.

- Query 4 is about pattern matching. Queries 8 – 9 are about finding path. These two types of tasks are required to navigate hyper-connectivity on graphs. Neo4j is completely optimised for these kinds of tasks [13] [47]. We, however, have not yet implemented the query optimisation strategies (refer to Section 4.3) for the RG engine.

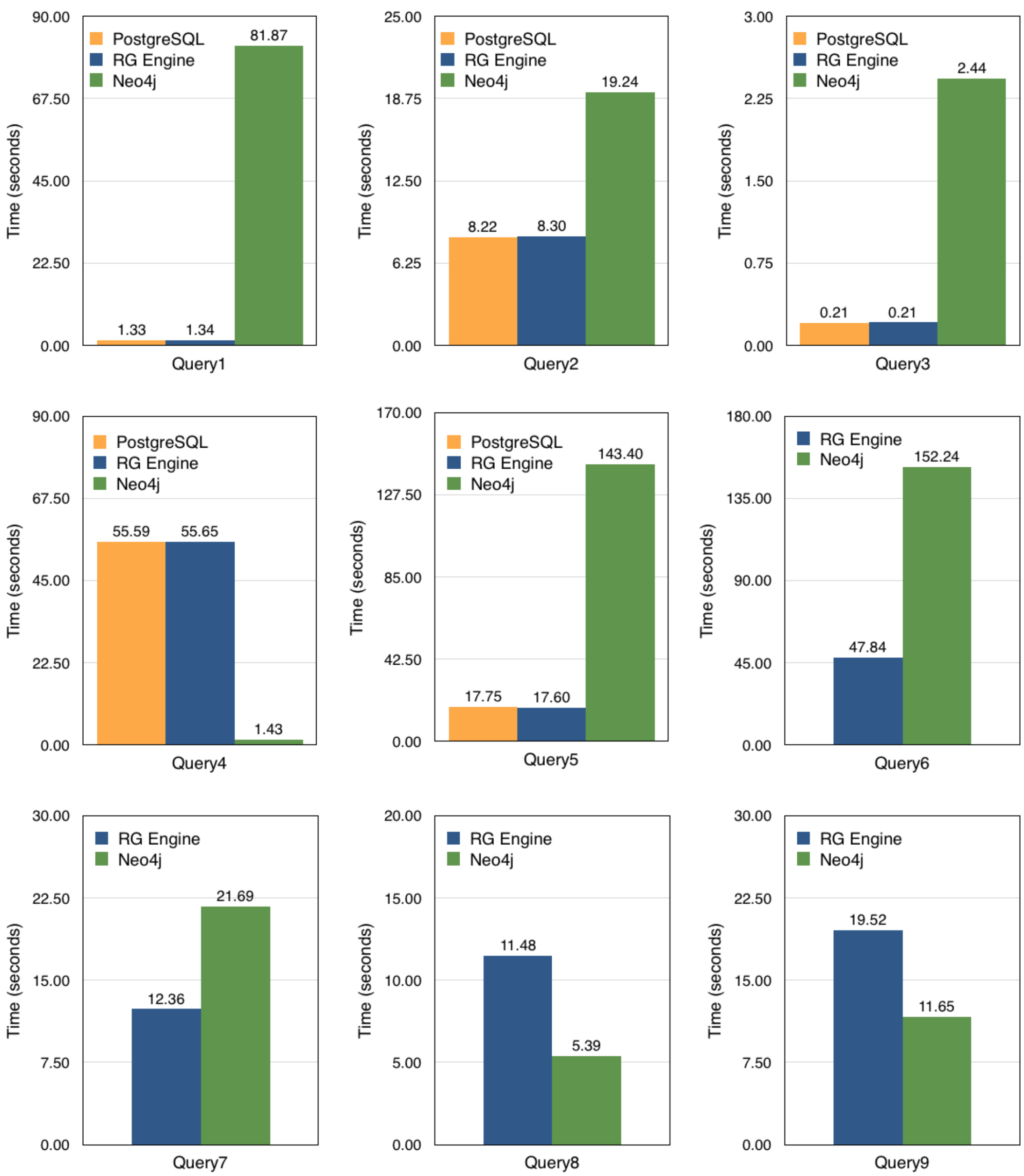


Figure 5.3: Time Performance of three Query Engines

In addition to all the queries mentioned above, we have also run queries about closeness centrality over the Twitter network using the RG engine and Neo4j. The RG engine can successfully process the queries. However, Neo4j failed to process these queries and the system reported the ”OutOfMemory” error. We suspect the reason is that the number of edges in Twitter network is too large, which exceeds the memory limitation of Neo4j.

5.4 Summary

In this chapter, we have conducted two experiments. One experiment is to evaluate three graph analysis tools (as algorithm support for the RG engine) with their time performance and memory performance. Another experiment is to compare the RG engine with other two query engines (one is PostgreSQL, another is Neo4j) in terms of query processing. According to the experiment results, the RG engine is able to process more types of queries and achieve better performance for most queries. However, for pattern matching and path finding queries, the RG engine is not efficient as Neo4j. In the future, We attempt to implement some query optimisation techniques for RG engine to improve its efficiency.

Chapter 6

Conclusion

Network analytics is one of the popular fields in computer science and its effect has already been augmented since the ”Big Data” era approaches. Nowadays, relational databases are still widely used by enterprises and organisations to process and manage their data. However, because of the rigid data model of relational databases, most of network analytics tasks do not fit well with relational databases. Therefore, the main purpose of this thesis is to describe a unified framework for network analytics via using data stored in relational databases. This unified framework includes a data model, a query language and a query engine.

Our data model is called RG model, which is a hybrid model of relations and graphs. Using the RG model, we are able to flexibly manage data in relations or in graphs. Correspondingly, we present a novel query language, called RG-SQL, which extends SQL with graph operators and graph construction features. RG-SQL aims to enable users to flexibly manipulate data from relations and graphs, supporting interactive data analysis between relational analysis and network analysis.

In terms of query processing, we leverage some components of an open-source relational database (PostgreSQL) to develop a query engine called RG engine. The main differences between the RG engine and traditional query engines of relational databases are: (1) the query parser of the RG engine is required to validate the syntax of graph sub-queries and differentiate between graph sub-queries and relational sub-queries; (2) the RG engine contains three additional executors for graph operations; (3) the query optimiser of the RG engine may incorporate some query optimisation strategies that are specially designed for RG-SQL queries. In addition to these, the experiments for performance evaluation demonstrate that RG engine is able to process various types of queries and achieve better performance in most cases. However, our experiments also expose some limitations of the RG engine when coping with pattern matching and path finding. The real advantage of the RG engine is the capability to combine different types of network analytics tasks with relational analysis.

There are a number of directions we may continue to explore as the **future work**, including:

- To incorporate query optimisation strategies into our query engine such as query equivalence and query caching.
- To support more network analytics tasks, such as sub-graph matching, K-core finding, link prediction and so forth.
- To support more graph types including weighted graphs and hyper graphs.
- To apply this unified framework on a distributed relational database architecture.

In conclusion, this thesis develops a unified framework which extends relational databases with network analytics capability. This unified framework is still in its fledgeling stage and we have a pile of ideas to enrich and maturate it. We hope, in the future, this unified framework would become full of vigour and vitality.

Appendices

Appendix A

ER Diagrams and Relation Schemas

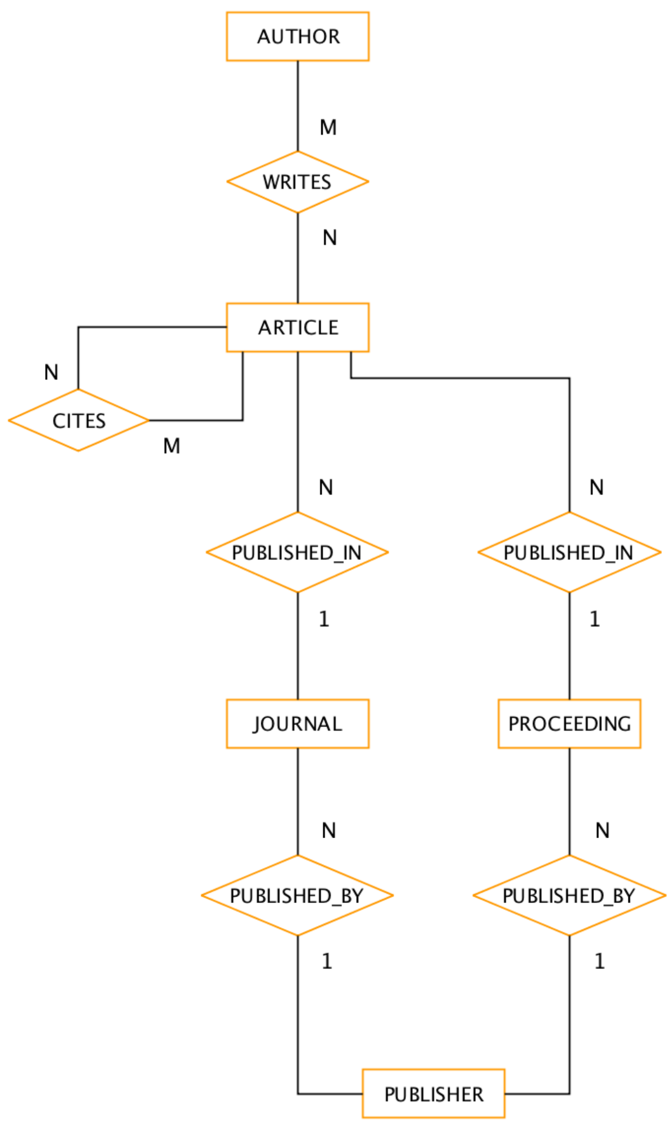


Figure A.1: The Entity-Relationship Diagram of ACM Bibliographical Network

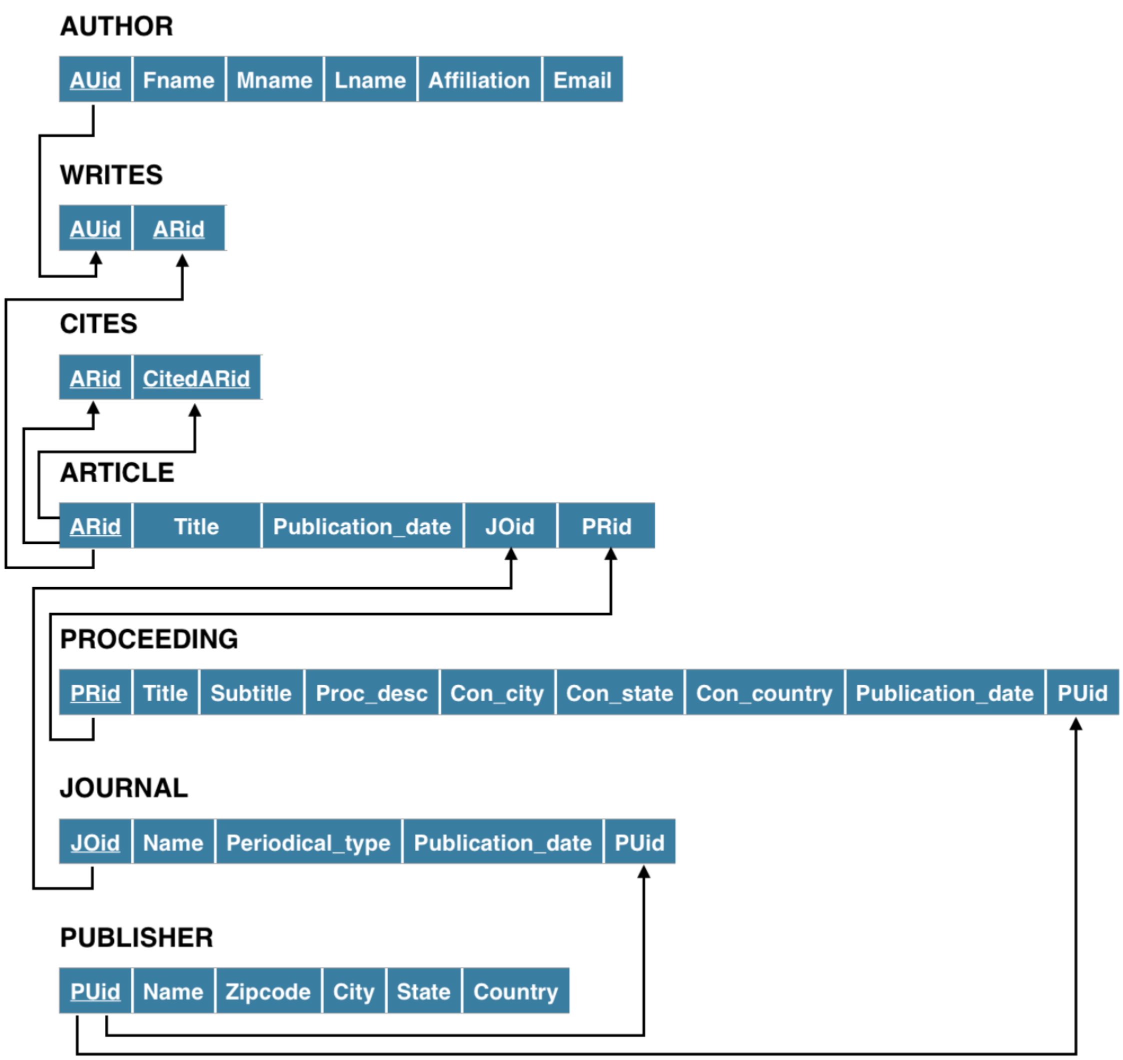


Figure A.2: The Relation Schema of ACM Bibliographical Network

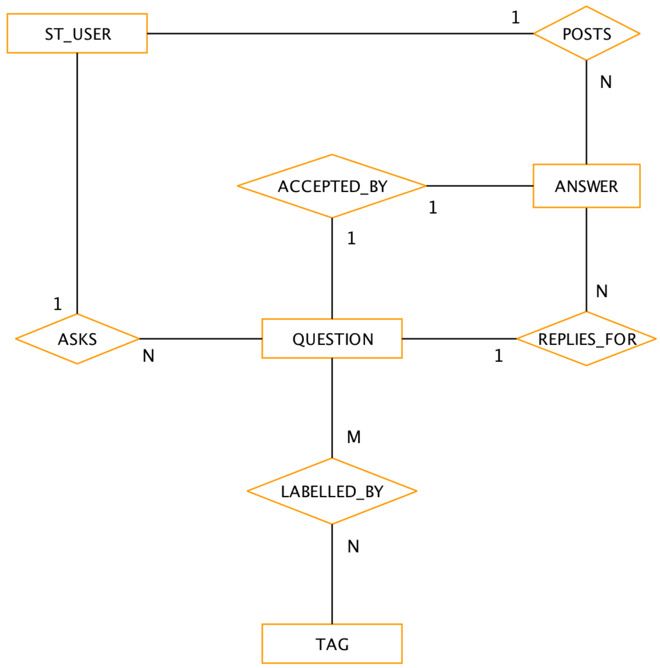


Figure A.3: The Entity-Relationship Diagram of Stack Overflow Network

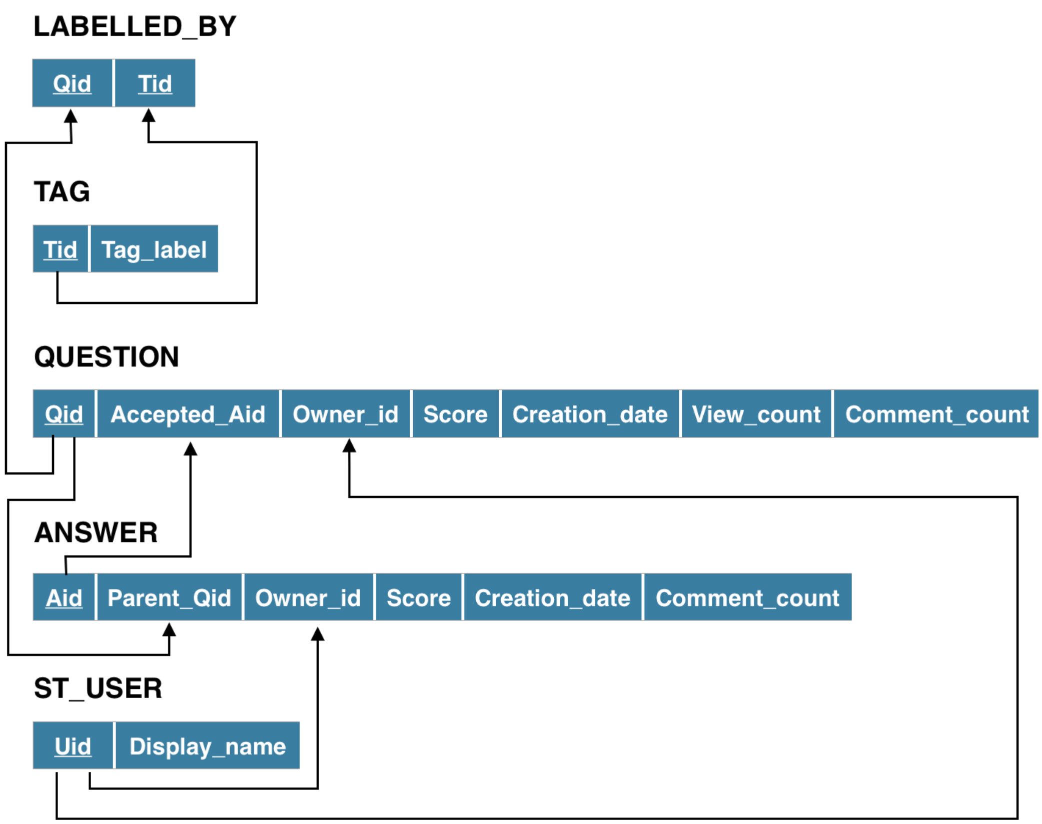


Figure A.4: The Relation Schema of Stack Overflow Network

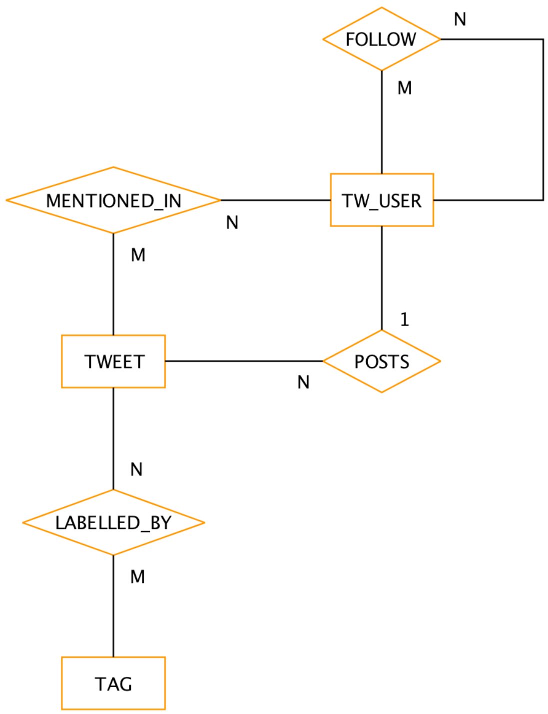


Figure A.5: The Entity-Relationship Diagram of Twitter Network

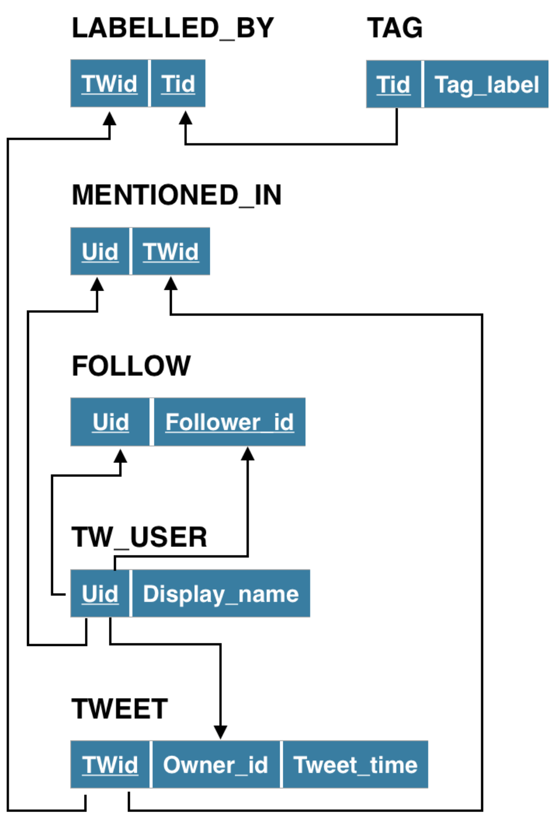


Figure A.6: The Relation Schema of Twitter Network

Appendix B

Experimental Data

Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9
PostgreSQL Time (s)	PostgreSQL Time (s)	PostgreSQL Time (s)	PostgreSQL Time (s)	PostgreSQL Time (s)	PostgreSQL Time (s)	PostgreSQL Time (s)	PostgreSQL Time (s)	PostgreSQL Time (s)
1.316	8.198	0.216	57.239	17.375	—	—	—	—
1.406	8.280	0.170	54.426	17.783	—	—	—	—
1.316	8.310	0.219	52.085	17.540	—	—	—	—
1.319	8.213	0.214	58.010	17.875	—	—	—	—
1.313	8.117	0.212	56.198	18.172	—	—	—	—
RG Engine Time (s)	RG Engine Time (s)	RG Engine Time (s)	RG Engine Time (s)	RG Engine Time (s)	RG Engine Time (s)	RG Engine Time (s)	RG Engine Time (s)	RG Engine Time (s)
1.403	8.223	0.189	54.123	17.948	48.313	11.332	11.345	19.354
1.315	8.135	0.212	57.235	17.394	47.195	12.346	11.590	19.347
1.313	8.235	0.212	56.235	18.219	48.124	13.104	11.437	20.125
1.323	8.575	0.216	58.225	17.346	47.591	12.124	11.345	19.458
1.369	8.345	0.218	52.453	17.084	47.987	12.898	11.679	19.335
Neo4j Time (s)	Neo4j Time (s)	Neo4j Time (s)	Neo4j Time (s)	Neo4j Time (s)	Neo4j Time (s)	Neo4j Time (s)	Neo4j Time (s)	Neo4j Time (s)
81.592	19.186	2.593	1.112	154.521	152.661	21.782	5.713	11.406
81.359	19.287	2.387	1.284	140.896	152.377	22.013	5.455	11.126
81.692	18.289	2.129	2.122	141.836	151.837	21.485	5.123	11.841
82.531	19.297	2.936	0.993	140.467	151.478	21.198	5.178	12.003
82.181	20.116	2.137	1.654	139.268	152.862	21.973	5.468	11.853

Figure B.1: Time Performance Data of Query Engines

Graph1 Constructing Time (s)	Graph2 Constructing Time (s)	Graph3 Constructing Time (s)	Graph4 Constructing Time (s)	Graph5 Constructing Time (s)	Graph6 Constructing Time (s)	Graph7 Constructing Time (s)	Graph8 Constructing Time (s)	Graph9 Constructing Time (s)	Graph10 Constructing Time (s)	Graph11 Constructing Time (s)	Graph12 Constructing Time (s)
0.00013	0.00039	0.00168	0.00047	0.00176	0.00853	0.00233	0.00953	0.05014	0.01201	0.04792	0.26010
0.00013	0.00039	0.00168	0.00048	0.00177	0.00867	0.00236	0.00959	0.05138	0.01218	0.04880	0.26663
0.00013	0.00039	0.00171	0.00048	0.00181	0.00879	0.00238	0.00993	0.05204	0.01245	0.05056	0.27433
0.00014	0.00040	0.00172	0.00049	0.00191	0.00997	0.00264	0.01009	0.05268	0.01290	0.05205	0.27877
0.00022	0.00047	0.00179	0.00056	0.00197	0.01084	0.00270	0.01027	0.05454	0.01314	0.05261	0.28383
Graph1 Degree Time (s)	Graph2 Degree Time (s)	Graph3 Degree Time (s)	Graph4 Degree Time (s)	Graph5 Degree Time (s)	Graph6 Degree Time (s)	Graph7 Degree Time (s)	Graph8 Degree Time (s)	Graph9 Degree Time (s)	Graph10 Degree Time (s)	Graph11 Degree Time (s)	Graph12 Degree Time (s)
0.00036	0.00046	0.00036	0.00158	0.00164	0.00182	0.00762	0.00790	0.00790	0.03815	0.04162	0.03969
0.00037	0.00046	0.00037	0.00169	0.00164	0.00185	0.00779	0.00822	0.00803	0.03979	0.04222	0.04064
0.00042	0.00056	0.00044	0.00179	0.00171	0.00193	0.00782	0.00849	0.00820	0.04011	0.04495	0.04109
0.00026	0.00036	0.00026	0.00159	0.00154	0.00174	0.00768	0.00812	0.00792	0.03969	0.04212	0.04053
0.00033	0.00047	0.00035	0.00169	0.00162	0.00183	0.00772	0.00840	0.00810	0.04002	0.04486	0.04100
Graph1 PageRank Time (s)	Graph2 PageRank Time (s)	Graph3 PageRank Time (s)	Graph4 PageRank Time (s)	Graph5 PageRank Time (s)	Graph6 PageRank Time (s)	Graph7 PageRank Time (s)	Graph8 PageRank Time (s)	Graph9 PageRank Time (s)	Graph10 PageRank Time (s)	Graph11 PageRank Time (s)	Graph12 PageRank Time (s)
0.00051	0.00072	0.00135	0.00349	0.00366	0.00762	0.01383	0.01915	0.04630	0.08516	0.09470	0.22941
0.00053	0.00072	0.00136	0.00353	0.00373	0.00779	0.01387	0.01928	0.04638	0.08656	0.10011	0.23045
0.00057	0.00079	0.00141	0.00363	0.00382	0.00795	0.01405	0.01929	0.04865	0.08802	0.10015	0.23189
0.00043	0.00062	0.00126	0.00343	0.00363	0.00769	0.01377	0.01918	0.04627	0.08646	0.10001	0.23035
0.00047	0.00069	0.00131	0.00353	0.00371	0.00785	0.01395	0.01919	0.04855	0.08792	0.10005	0.23179
Graph1 Betweenness Time (s)	Graph2 Betweenness Time (s)	Graph3 Betweenness Time (s)	Graph4 Betweenness Time (s)	Graph5 Betweenness Time (s)	Graph6 Betweenness Time (s)	Graph7 Betweenness Time (s)	Graph8 Betweenness Time (s)	Graph9 Betweenness Time (s)	Graph10 Betweenness Time (s)	Graph11 Betweenness Time (s)	Graph12 Betweenness Time (s)
0.00804	0.01463	0.04797	0.13231	0.41395	1.31810	3.91431	10.02800	32.80723	123.36672	340.78084	1665.38647
0.00827	0.01480	0.04949	0.13283	0.41449	1.33705	3.91861	10.03061	32.91175	123.42373	342.41570	1688.01751
0.02193	0.01572	0.04964	0.13299	0.41867	1.34146	3.92747	10.04740	33.52034	123.53229	354.89726	1707.17448
0.00817	0.01470	0.04939	0.13273	0.41439	1.33695	3.91851	10.03051	32.91165	123.42363	342.41560	1688.01740
0.02183	0.01562	0.04954	0.13289	0.41857	1.34136	3.92737	10.04729	33.52024	123.53219	354.89716	1707.17438

Figure B.2: Time Performance Data of SNAP – Part 1

Graph1 Closeness Time (s)	Graph2 Closeness Time (s)	Graph3 Closeness Time (s)	Graph4 Closeness Time (s)	Graph5 Closeness Time (s)	Graph6 Closeness Time (s)	Graph7 Closeness Time (s)	Graph8 Closeness Time (s)	Graph9 Closeness Time (s)	Graph10 Closeness Time (s)	Graph11 Closeness Time (s)	Graph12 Closeness Time (s)
0.00202	0.00505	0.02041	0.03652	0.10554	0.41348	1.01044	3.11579	13.12567	30.56022	68.22805	308.59938
0.00213	0.00507	0.02197	0.03681	0.10799	0.41451	1.02363	3.12701	13.12973	30.62040	68.23408	309.27073
0.00217	0.00515	0.02654	0.03720	0.10880	0.41465	1.02712	3.14832	13.13133	30.67966	68.37704	310.69287
0.00203	0.00497	0.02187	0.03670	0.10788	0.41441	1.02353	3.12690	13.12962	30.62030	68.23398	309.27062
0.00207	0.00505	0.02644	0.03710	0.10869	0.41455	1.02702	3.14822	13.13123	30.67956	68.37694	310.69277
Graph1 Connected Component Time (s)	Graph2 Connected Component Time (s)	Graph3 Connected Component Time (s)	Graph4 Connected Component Time (s)	Graph5 Connected Component Time (s)	Graph6 Connected Component Time (s)	Graph7 Connected Component Time (s)	Graph8 Connected Component Time (s)	Graph9 Connected Component Time (s)	Graph10 Connected Component Time (s)	Graph11 Connected Component Time (s)	Graph12 Connected Component Time (s)
0.00006	0.00009	0.00018	0.00013	0.00027	0.00088	0.00057	0.00146	0.00534	0.00322	0.00567	0.02131
0.00008	0.00010	0.00018	0.00018	0.00028	0.00090	0.00059	0.00151	0.00539	0.00323	0.00569	0.02148
0.00670	0.00016	0.00023	0.00026	0.00033	0.00094	0.00064	0.00152	0.00555	0.00600	0.00574	0.02397
0.00008	0.00009	0.00018	0.00018	0.00028	0.00090	0.00058	0.00151	0.00539	0.00323	0.00568	0.02148
0.00660	0.00005	0.00012	0.00016	0.00022	0.00083	0.00054	0.00142	0.00544	0.00589	0.00564	0.02386
Graph1 Strongly Connected Component Time (s)	Graph2 Strongly Connected Component Time (s)	Graph3 Strongly Connected Component Time (s)	Graph4 Strongly Connected Component Time (s)	Graph5 Strongly Connected Component Time (s)	Graph6 Strongly Connected Component Time (s)	Graph7 Strongly Connected Component Time (s)	Graph8 Strongly Connected Component Time (s)	Graph9 Strongly Connected Component Time (s)	Graph10 Strongly Connected Component Time (s)	Graph11 Strongly Connected Component Time (s)	Graph12 Strongly Connected Component Time (s)
0.00013	0.00013	0.00036	0.00028	0.00049	0.00180	0.00142	0.00247	0.00923	0.00800	0.01266	0.05099
0.00016	0.00014	0.00039	0.00031	0.00049	0.00187	0.00143	0.00255	0.00938	0.00909	0.01294	0.05129
0.00016	0.00019	0.00049	0.00033	0.00054	0.00218	0.00146	0.00255	0.00942	0.01184	0.01553	0.05371
0.00015	0.00014	0.00039	0.00031	0.00049	0.00187	0.00142	0.00255	0.00938	0.00909	0.01294	0.05128
0.00016	0.00019	0.00049	0.00033	0.00054	0.00218	0.00146	0.00255	0.00941	0.01184	0.01553	0.05371

Figure B.3: Time Performance Data of SNAP – Part 2

Graph1 Constructing Memory (MB)	Graph2 Constructing Memory (MB)	Graph3 Constructing Memory (MB)	Graph4 Constructing Memory (MB)	Graph5 Constructing Memory (MB)	Graph6 Constructing Memory (MB)	Graph7 Constructing Memory (MB)	Graph8 Constructing Memory (MB)	Graph9 Constructing Memory (MB)	Graph10 Constructing Memory (MB)	Graph11 Constructing Memory (MB)	Graph12 Constructing Memory (MB)
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.16797	1.24609	2.08594	7.48828
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.26563	1.24609	1.99609	7.48828
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.16797	1.24609	1.99609	7.48828
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.26563	1.24609	1.99609	7.48828
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.26563	1.24609	1.99609	7.48828
Graph1 Degree Memory (MB)	Graph2 Degree Memory (MB)	Graph3 Degree Memory (MB)	Graph4 Degree Memory (MB)	Graph5 Degree Memory (MB)	Graph6 Degree Memory (MB)	Graph7 Degree Memory (MB)	Graph8 Degree Memory (MB)	Graph9 Degree Memory (MB)	Graph10 Degree Memory (MB)	Graph11 Degree Memory (MB)	Graph12 Degree Memory (MB)
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.41797	1.13672	1.82422	1.93750	7.35156
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.41797	1.35547	1.82422	3.22656	8.12891
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.55078	1.35547	2.17188	3.45703	8.12500
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.55078	1.35547	2.17188	3.45703	8.12500
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.55078	1.35547	2.17188	3.45703	8.12500
Graph1 PageRank Memory (MB)	Graph2 PageRank Memory (MB)	Graph3 PageRank Memory (MB)	Graph4 PageRank Memory (MB)	Graph5 PageRank Memory (MB)	Graph6 PageRank Memory (MB)	Graph7 PageRank Memory (MB)	Graph8 PageRank Memory (MB)	Graph9 PageRank Memory (MB)	Graph10 PageRank Memory (MB)	Graph11 PageRank Memory (MB)	Graph12 PageRank Memory (MB)
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.13672	1.78516	3.04688	7.35156
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.36328	1.26563	1.78516	3.04688	7.86328
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.36328	1.26563	1.78516	3.04688	7.86328
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.36328	1.26563	1.78516	3.04688	7.86328
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.36328	1.26563	1.78516	3.04688	7.86328
Graph1 Betweenness Memory (MB)	Graph2 Betweenness Memory (MB)	Graph3 Betweenness Memory (MB)	Graph4 Betweenness Memory (MB)	Graph5 Betweenness Memory (MB)	Graph6 Betweenness Memory (MB)	Graph7 Betweenness Memory (MB)	Graph8 Betweenness Memory (MB)	Graph9 Betweenness Memory (MB)	Graph10 Betweenness Memory (MB)	Graph11 Betweenness Memory (MB)	Graph12 Betweenness Memory (MB)
0.00000	0.00000	0.00000	0.00000	0.28125	1.08203	0.59766	1.60938	5.86328	3.77344	8.12891	39.65625
0.00000	0.00000	0.00000	0.00000	0.28125	1.08203	0.59766	1.60938	5.86328	3.77344	8.12891	39.65625
0.00000	0.00000	0.00000	0.00000	0.28125	1.47656	0.72656	2.08984	7.39063	4.76172	10.01563	56.70703
0.00000	0.00000	0.00000	0.00000	0.28125	1.47656	0.67969	2.08984	7.39063	4.76172	10.01563	56.70703
0.00000	0.00000	0.00000	0.00000	0.28125	1.47656	0.67969	2.08984	7.39063	4.76172	10.01563	56.70703

Figure B.4: Memory Performance Data of SNAP – Part 1

Graph1 Closeness Memory (MB)	Graph2 Closeness Memory (MB)	Graph3 Closeness Memory (MB)	Graph4 Closeness Memory (MB)	Graph5 Closeness Memory (MB)	Graph6 Closeness Memory (MB)	Graph7 Closeness Memory (MB)	Graph8 Closeness Memory (MB)	Graph9 Closeness Memory (MB)	Graph10 Closeness Memory (MB)	Graph11 Closeness Memory (MB)	Graph12 Closeness Memory (MB)
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.36328	1.39453	1.87891	3.14063	7.35156
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.36328	1.39453	1.87891	3.14063	8.36328
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.49609	1.39453	2.26563	3.74219	8.36328
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.49609	1.39453	2.26563	3.74219	8.36328
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.49609	1.39453	2.26563	3.74219	8.36328
Graph1 Connected Component Memory (MB)	Graph2 Connected Component Memory (MB)	Graph3 Connected Component Memory (MB)	Graph4 Connected Component Memory (MB)	Graph5 Connected Component Memory (MB)	Graph6 Connected Component Memory (MB)	Graph7 Connected Component Memory (MB)	Graph8 Connected Component Memory (MB)	Graph9 Connected Component Memory (MB)	Graph10 Connected Component Memory (MB)	Graph11 Connected Component Memory (MB)	Graph12 Connected Component Memory (MB)
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.13672	0.86328	1.93750	7.35156
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.13672	1.30078	2.44141	7.72266
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.13672	1.30078	2.44141	7.72266
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.13672	1.30078	2.44141	7.72266
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23438	0.23438	1.13672	1.30078	2.44141	7.72266
Graph1 Strongly Connected Component Memory (MB)	Graph2 Strongly Connected Component Memory (MB)	Graph3 Strongly Connected Component Memory (MB)	Graph4 Strongly Connected Component Memory (MB)	Graph5 Strongly Connected Component Memory (MB)	Graph6 Strongly Connected Component Memory (MB)	Graph7 Strongly Connected Component Memory (MB)	Graph8 Strongly Connected Component Memory (MB)	Graph9 Strongly Connected Component Memory (MB)	Graph10 Strongly Connected Component Memory (MB)	Graph11 Strongly Connected Component Memory (MB)	Graph12 Strongly Connected Component Memory (MB)
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.32813	0.32813	1.23047	1.59375	2.37891	7.79297
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.32813	0.46094	1.36328	2.55859	3.29688	8.37891
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.32813	0.46094	1.36328	2.55859	3.29688	8.37891
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.32813	0.46094	1.36328	2.55859	3.29688	8.37891
0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.32813	0.46094	1.36328	1.75000	3.29688	8.37891

Figure B.5: Memory Performance Data of SNAP – Part 2

Graph1 Constructing Time (s)	Graph2 Constructing Time (s)	Graph3 Constructing Time (s)	Graph4 Constructing Time (s)	Graph5 Constructing Time (s)	Graph6 Constructing Time (s)	Graph7 Constructing Time (s)	Graph8 Constructing Time (s)	Graph9 Constructing Time (s)	Graph10 Constructing Time (s)	Graph11 Constructing Time (s)	Graph12 Constructing Time (s)
0.00038	0.00170	0.00800	0.00190	0.00896	0.05115	0.00926	0.05191	0.30018	0.05935	0.31305	1.63864
0.00039	0.00175	0.00804	0.00193	0.00933	0.05157	0.00934	0.05224	0.31601	0.06402	0.34149	1.86331
0.00040	0.00182	0.00823	0.00206	0.00967	0.05177	0.00946	0.05251	0.31767	0.06532	0.34313	1.87423
0.00043	0.00184	0.00827	0.00207	0.01124	0.05249	0.00968	0.05435	0.31939	0.06582	0.34830	1.88199
0.00047	0.00200	0.00832	0.00208	0.01148	0.05377	0.01123	0.05796	0.32468	0.06678	0.34872	1.88429
Graph1 Degree Time (s)	Graph2 Degree Time (s)	Graph3 Degree Time (s)	Graph4 Degree Time (s)	Graph5 Degree Time (s)	Graph6 Degree Time (s)	Graph7 Degree Time (s)	Graph8 Degree Time (s)	Graph9 Degree Time (s)	Graph10 Degree Time (s)	Graph11 Degree Time (s)	Graph12 Degree Time (s)
0.00009	0.00009	0.00009	0.00029	0.00028	0.00030	0.00120	0.00138	0.00132	0.00742	0.00991	0.00883
0.00011	0.00009	0.00009	0.00029	0.00028	0.00035	0.00120	0.00143	0.00136	0.00817	0.01232	0.00925
0.00013	0.00011	0.00018	0.00033	0.00034	0.00036	0.00129	0.00159	0.00185	0.00866	0.01295	0.01050
0.00022	0.00020	0.00020	0.00040	0.00039	0.00046	0.00131	0.00154	0.00147	0.00827	0.01243	0.00935
0.00023	0.00021	0.00028	0.00043	0.00044	0.00046	0.00139	0.00169	0.00195	0.00876	0.01305	0.01060
Graph1 PageRank Time (s)	Graph2 PageRank Time (s)	Graph3 PageRank Time (s)	Graph4 PageRank Time (s)	Graph5 PageRank Time (s)	Graph6 PageRank Time (s)	Graph7 PageRank Time (s)	Graph8 PageRank Time (s)	Graph9 PageRank Time (s)	Graph10 PageRank Time (s)	Graph11 PageRank Time (s)	Graph12 PageRank Time (s)
0.00547	0.01147	0.03156	0.02802	0.06606	0.32413	0.12218	0.35611	1.59563	0.63170	1.93717	8.62340
0.00555	0.01151	0.03290	0.02999	0.06634	0.32886	0.12278	0.36158	1.70335	0.64680	2.04210	9.04262
0.00579	0.01194	0.03507	0.03262	0.06862	0.35479	0.12492	0.36770	1.71899	0.66461	2.06531	9.13207
0.00546	0.01142	0.03281	0.02990	0.06625	0.32876	0.12269	0.36149	1.70326	0.64670	2.04201	9.04252
0.00566	0.01181	0.03494	0.03249	0.06850	0.35467	0.12480	0.36757	1.71886	0.66449	2.06518	9.13194
Graph1 Betweenness Time (s)	Graph2 Betweenness Time (s)	Graph3 Betweenness Time (s)	Graph4 Betweenness Time (s)	Graph5 Betweenness Time (s)	Graph6 Betweenness Time (s)	Graph7 Betweenness Time (s)	Graph8 Betweenness Time (s)	Graph9 Betweenness Time (s)	Graph10 Betweenness Time (s)	Graph11 Betweenness Time (s)	Graph12 Betweenness Time (s)
0.01540	0.02933	0.08385	0.42321	1.01315	2.91795	11.87792	27.78613	106.05446	508.97276	1266.95199	4862.05763
0.01547	0.02961	0.08446	0.42470	1.01499	2.92062	11.90781	27.91953	109.56798	510.80694	1310.42698	4918.00273
0.01585	0.03096	0.08564	0.42762	1.02239	2.94118	11.94836	28.01099	111.02169	518.16466	1387.63835	5085.82583
0.01540	0.02954	0.08438	0.42463	1.01492	2.92055	11.90773	27.91946	109.56791	510.80686	1310.42691	4918.00266
0.01572	0.03083	0.08551	0.42749	1.02226	2.94105	11.94823	28.01086	111.02156	518.16454	1387.63822	5085.82570

Figure B.6: Time Performance Data of NetworkX – Part 1

Graph1 Closeness Time (s)	Graph2 Closeness Time (s)	Graph3 Closeness Time (s)	Graph4 Closeness Time (s)	Graph5 Closeness Time (s)	Graph6 Closeness Time (s)	Graph7 Closeness Time (s)	Graph8 Closeness Time (s)	Graph9 Closeness Time (s)	Graph10 Closeness Time (s)	Graph11 Closeness Time (s)	Graph12 Closeness Time (s)
0.00559	0.00756	0.01285	0.14292	0.25327	1.12646	3.86073	11.79448	69.60303	153.94321	448.37476	1723.67105
0.00566	0.00759	0.01307	0.14484	0.25538	1.13065	3.86309	11.82508	69.63367	154.19643	456.96698	1724.40261
0.00569	0.00760	0.01478	0.14544	0.25758	1.14876	3.88376	11.92476	69.63925	154.35797	494.20209	2005.97665
0.00523	0.00716	0.01265	0.14441	0.25495	1.13022	3.86267	11.82466	69.63324	154.19600	456.96655	1724.40218
0.00559	0.00750	0.01469	0.14535	0.25749	1.14867	3.88367	11.92466	69.63916	154.35788	494.20199	2005.97656
Graph1 Connected Component Time (s)	Graph2 Connected Component Time (s)	Graph3 Connected Component Time (s)	Graph4 Connected Component Time (s)	Graph5 Connected Component Time (s)	Graph6 Connected Component Time (s)	Graph7 Connected Component Time (s)	Graph8 Connected Component Time (s)	Graph9 Connected Component Time (s)	Graph10 Connected Component Time (s)	Graph11 Connected Component Time (s)	Graph12 Connected Component Time (s)
0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004
0.00004	0.00005	0.00004	0.00004	0.00004	0.00005	0.00004	0.00004	0.00006	0.00005	0.00004	0.00005
0.00006	0.00007	0.00006	0.00006	0.00006	0.00006	0.00006	0.00007	0.00006	0.00007	0.00006	0.00007
0.00003	0.00005	0.00003	0.00003	0.00003	0.00004	0.00003	0.00004	0.00005	0.00004	0.00004	0.00004
0.00006	0.00007	0.00006	0.00006	0.00005	0.00006	0.00006	0.00007	0.00006	0.00006	0.00006	0.00007
Graph1 Strongly Connected Component Time (s)	Graph2 Strongly Connected Component Time (s)	Graph3 Strongly Connected Component Time (s)	Graph4 Strongly Connected Component Time (s)	Graph5 Strongly Connected Component Time (s)	Graph6 Strongly Connected Component Time (s)	Graph7 Strongly Connected Component Time (s)	Graph8 Strongly Connected Component Time (s)	Graph9 Strongly Connected Component Time (s)	Graph10 Strongly Connected Component Time (s)	Graph11 Strongly Connected Component Time (s)	Graph12 Strongly Connected Component Time (s)
0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004
0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00006	0.00004
0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00007	0.00007
0.00003	0.00003	0.00003	0.00003	0.00003	0.00003	0.00003	0.00003	0.00003	0.00003	0.00005	0.00003
0.00006	0.00005	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006	0.00006

Figure B.7: Time Performance Data of NetworkX – Part 2

Graph1 Constructing Memory (MB)	Graph2 Constructing Memory (MB)	Graph3 Constructing Memory (MB)	Graph4 Constructing Memory (MB)	Graph5 Constructing Memory (MB)	Graph6 Constructing Memory (MB)	Graph7 Constructing Memory (MB)	Graph8 Constructing Memory (MB)	Graph9 Constructing Memory (MB)	Graph10 Constructing Memory (MB)	Graph11 Constructing Memory (MB)	Graph12 Constructing Memory (MB)
0.00391	0.38281	1.67578	0.65625	2.92969	13.98438	3.87891	15.60156	79.77344	20.12891	80.01172	406.70313
0.00391	0.38281	1.59766	0.65625	2.98047	14.10547	3.87891	15.58984	79.75391	20.17188	80.01172	406.72266
0.00391	0.38281	1.59766	0.65625	2.98047	14.10547	3.87891	15.58984	79.75391	20.17188	80.01172	406.72266
0.00391	0.38281	1.59766	0.65625	2.98047	14.10547	3.87891	15.58984	79.75391	20.17188	80.01172	406.72266
0.00391	0.38281	1.59766	0.65625	2.98047	14.10547	3.79688	15.58984	79.75391	20.17188	80.01172	406.72266
Graph1 Degree Memory (MB)	Graph2 Degree Memory (MB)	Graph3 Degree Memory (MB)	Graph4 Degree Memory (MB)	Graph5 Degree Memory (MB)	Graph6 Degree Memory (MB)	Graph7 Degree Memory (MB)	Graph8 Degree Memory (MB)	Graph9 Degree Memory (MB)	Graph10 Degree Memory (MB)	Graph11 Degree Memory (MB)	Graph12 Degree Memory (MB)
0.13672	0.37891	1.64063	0.66406	2.91016	12.74219	3.75000	14.34766	70.51172	18.92188	70.76172	358.28125
0.13672	0.37891	1.64063	0.79688	3.06641	12.74219	4.11328	14.61719	70.81250	20.03125	72.18750	359.95313
0.13672	0.37891	1.64063	0.79688	3.06641	12.74219	4.11328	14.61719	70.81250	20.15625	72.18750	359.70313
0.13672	0.37891	1.64063	0.79688	3.06641	12.74219	4.11328	14.61719	70.81250	20.15625	72.18750	359.70313
0.13672	0.37891	1.64063	0.79688	3.06641	12.74219	4.11328	14.61719	70.81250	20.15625	72.18750	359.70313
Graph1 PageRank Memory (MB)	Graph2 PageRank Memory (MB)	Graph3 PageRank Memory (MB)	Graph4 PageRank Memory (MB)	Graph5 PageRank Memory (MB)	Graph6 PageRank Memory (MB)	Graph7 PageRank Memory (MB)	Graph8 PageRank Memory (MB)	Graph9 PageRank Memory (MB)	Graph10 PageRank Memory (MB)	Graph11 PageRank Memory (MB)	Graph12 PageRank Memory (MB)
0.78516	2.02344	3.80859	2.26172	7.21875	32.45313	9.04688	33.81250	186.86328	35.98828	161.11719	939.04688
0.78516	2.02344	3.80859	2.26172	7.21875	32.45313	9.04688	33.81250	186.86328	35.98828	161.11719	941.61328
0.78516	2.02344	4.05859	2.26172	7.46875	32.31250	9.92188	33.56250	195.90625	38.25781	163.98047	939.04688
0.78516	2.02344	4.30859	2.26172	7.21875	32.56250	10.67188	33.81250	196.40625	38.50781	164.23047	939.29688
0.78516	2.02344	4.30859	2.26172	7.21875	32.56250	10.67188	33.81250	187.55859	38.50781	164.23047	939.04688
Graph1 Betweenness Memory (MB)	Graph2 Betweenness Memory (MB)	Graph3 Betweenness Memory (MB)	Graph4 Betweenness Memory (MB)	Graph5 Betweenness Memory (MB)	Graph6 Betweenness Memory (MB)	Graph7 Betweenness Memory (MB)	Graph8 Betweenness Memory (MB)	Graph9 Betweenness Memory (MB)	Graph10 Betweenness Memory (MB)	Graph11 Betweenness Memory (MB)	Graph12 Betweenness Memory (MB)
0.14453	0.75781	1.66406	1.37891	2.90625	12.73438	3.88281	14.35547	70.50391	18.87500	70.75000	358.27734
0.14453	0.75781	1.66406	1.37891	3.20703	12.88672	5.54297	14.62500	71.94531	24.78125	72.23438	363.54297
0.14453	0.50781	1.66406	1.37891	3.20703	12.88672	5.54297	14.62500	71.94531	24.02734	75.39844	363.54297
0.14453	0.75781	1.66406	1.37891	3.20703	12.88672	5.54297	14.62500	71.94531	24.02734	75.39844	363.54297
0.14453	0.75781	1.66406	1.37891	3.20703	12.88672	5.54297	14.62500	71.94531	24.02734	75.39844	363.54297

Figure B.8: Memory Performance Data of NetworkX – Part 1

Graph1 Closeness Memory (MB)	Graph2 Closeness Memory (MB)	Graph3 Closeness Memory (MB)	Graph4 Closeness Memory (MB)	Graph5 Closeness Memory (MB)	Graph6 Closeness Memory (MB)	Graph7 Closeness Memory (MB)	Graph8 Closeness Memory (MB)	Graph9 Closeness Memory (MB)	Graph10 Closeness Memory (MB)	Graph11 Closeness Memory (MB)	Graph12 Closeness Memory (MB)
0.12891	0.37891	1.65625	0.66016	2.92578	12.71875	3.76172	14.33203	70.53516	18.87500	70.74609	358.27344
0.12891	0.50781	1.65625	0.93359	3.19922	12.87109	4.11328	14.62500	71.45703	19.92188	71.93750	363.52734
0.12891	0.50781	1.65625	0.93359	3.19922	12.87109	4.16016	14.62500	71.45703	20.15625	71.93750	363.52734
0.12891	0.50781	1.65625	0.93359	3.19922	12.87109	4.16016	14.62500	71.45703	20.15625	71.93750	363.52734
0.12891	0.50781	1.65625	0.93359	3.19922	12.87109	4.16016	14.62500	71.45703	20.15625	71.93750	363.52734
Graph1 Connected Component Memory (MB)	Graph2 Connected Component Memory (MB)	Graph3 Connected Component Memory (MB)	Graph4 Connected Component Memory (MB)	Graph5 Connected Component Memory (MB)	Graph6 Connected Component Memory (MB)	Graph7 Connected Component Memory (MB)	Graph8 Connected Component Memory (MB)	Graph9 Connected Component Memory (MB)	Graph10 Connected Component Memory (MB)	Graph11 Connected Component Memory (MB)	Graph12 Connected Component Memory (MB)
0.12891	0.37891	1.66016	0.64063	2.94141	12.76563	3.75000	14.34375	70.53125	18.91016	70.74609	358.26953
0.12891	0.51172	1.66016	0.64063	2.94141	12.76563	3.91016	14.34375	70.53125	18.91016	70.74609	358.26953
0.12891	0.51172	1.66016	0.64063	2.94141	12.76563	3.91016	14.34375	70.53125	18.91016	70.74609	358.26953
0.12891	0.51172	1.66016	0.64063	2.94141	12.76563	3.91016	14.34375	70.53125	18.91016	70.74609	358.26953
0.12891	0.51172	1.66016	0.64063	2.94141	12.76563	3.91016	14.34375	70.53125	18.91016	70.74609	358.26953
Graph1 Strongly Connected Component Memory (MB)	Graph2 Strongly Connected Component Memory (MB)	Graph3 Strongly Connected Component Memory (MB)	Graph4 Strongly Connected Component Memory (MB)	Graph5 Strongly Connected Component Memory (MB)	Graph6 Strongly Connected Component Memory (MB)	Graph7 Strongly Connected Component Memory (MB)	Graph8 Strongly Connected Component Memory (MB)	Graph9 Strongly Connected Component Memory (MB)	Graph10 Strongly Connected Component Memory (MB)	Graph11 Strongly Connected Component Memory (MB)	Graph12 Strongly Connected Component Memory (MB)
0.15234	0.62891	2.17188	0.92188	2.82031	11.33594	4.61719	14.55859	58.08203	21.45313	71.00391	293.00000
0.15234	0.76563	2.17188	0.92188	2.97656	11.33594	4.61719	14.55859	58.23047	21.45313	71.15625	293.00000
0.15234	0.76563	2.17188	0.92188	2.97656	11.33594	4.61719	14.55859	58.23047	21.45313	71.15625	293.00000
0.15234	0.76563	2.17188	0.92188	2.97656	11.33594	4.61719	14.55859	58.23047	21.45313	71.15625	293.00000
0.15234	0.76563	2.17188	0.92188	2.97656	11.33594	4.61719	14.55859	58.23047	21.45313	71.15625	293.00000

Figure B.9: Memory Performance Data of NetworkX – Part 2

Graph1 Constructing Time (s)	Graph2 Constructing Time (s)	Graph3 Constructing Time (s)	Graph4 Constructing Time (s)	Graph5 Constructing Time (s)	Graph6 Constructing Time (s)	Graph7 Constructing Time (s)	Graph8 Constructing Time (s)	Graph9 Constructing Time (s)	Graph10 Constructing Time (s)	Graph11 Constructing Time (s)	Graph12 Constructing Time (s)
0.00441	0.02156	0.10688	0.02206	0.11092	0.54928	0.11372	0.54924	2.77284	0.56906	2.81328	13.98697
0.00471	0.02174	0.10773	0.02222	0.11112	0.54991	0.11418	0.55016	2.78290	0.56907	2.82220	13.99511
0.00483	0.02190	0.11112	0.02263	0.11123	0.55140	0.11421	0.55018	2.78331	0.57445	2.83239	14.01760
0.00484	0.02301	0.11143	0.02302	0.11128	0.55301	0.11465	0.55503	2.78652	0.57717	2.83720	14.07886
0.00486	0.02481	0.11397	0.02352	0.11163	0.56006	0.11603	0.55855	2.79884	0.58007	2.84844	14.10519
Graph1 Degree Time (s)	Graph2 Degree Time (s)	Graph3 Degree Time (s)	Graph4 Degree Time (s)	Graph5 Degree Time (s)	Graph6 Degree Time (s)	Graph7 Degree Time (s)	Graph8 Degree Time (s)	Graph9 Degree Time (s)	Graph10 Degree Time (s)	Graph11 Degree Time (s)	Graph12 Degree Time (s)
0.00052	0.00053	0.00052	0.00233	0.00233	0.00243	0.01153	0.01162	0.01164	0.05802	0.05839	0.05995
0.00053	0.00053	0.00053	0.00244	0.00237	0.00245	0.01163	0.01164	0.01170	0.05890	0.06186	0.06032
0.00068	0.00064	0.00064	0.00254	0.00251	0.00257	0.01184	0.01173	0.01204	0.05952	0.06333	0.06149
0.00069	0.00069	0.00069	0.00259	0.00253	0.00261	0.01179	0.01180	0.01186	0.05906	0.06202	0.06047
0.00078	0.00073	0.00074	0.00264	0.00261	0.00267	0.01193	0.01183	0.01214	0.05962	0.06342	0.06159
Graph1 PageRank Time (s)	Graph2 PageRank Time (s)	Graph3 PageRank Time (s)	Graph4 PageRank Time (s)	Graph5 PageRank Time (s)	Graph6 PageRank Time (s)	Graph7 PageRank Time (s)	Graph8 PageRank Time (s)	Graph9 PageRank Time (s)	Graph10 PageRank Time (s)	Graph11 PageRank Time (s)	Graph12 PageRank Time (s)
0.00323	0.00226	0.00261	0.00591	0.00543	0.00647	0.02442	0.02383	0.03153	0.12208	0.14475	0.18382
0.00342	0.00244	0.00280	0.00609	0.00561	0.00665	0.02461	0.02401	0.03171	0.12227	0.14494	0.18400
0.00329	0.00227	0.00262	0.00591	0.00552	0.00653	0.02459	0.02460	0.03201	0.12281	0.14591	0.18404
0.05459	0.00263	0.00340	0.00601	0.00620	0.00703	0.02554	0.02462	0.03233	0.12576	0.15332	0.18789
0.05450	0.00253	0.00330	0.00592	0.00610	0.00693	0.02545	0.02452	0.03224	0.12567	0.15323	0.18779
Graph1 Betweenness Time (s)	Graph2 Betweenness Time (s)	Graph3 Betweenness Time (s)	Graph4 Betweenness Time (s)	Graph5 Betweenness Time (s)	Graph6 Betweenness Time (s)	Graph7 Betweenness Time (s)	Graph8 Betweenness Time (s)	Graph9 Betweenness Time (s)	Graph10 Betweenness Time (s)	Graph11 Betweenness Time (s)	Graph12 Betweenness Time (s)
0.00586	0.00830	0.02457	0.07442	0.17033	0.42078	1.81236	3.63100	8.80652	57.91824	197.51308	658.33331
0.00381	0.00646	0.02218	0.07200	0.16843	0.41815	1.79374	3.62867	8.66594	57.81469	197.02182	642.13140
0.00402	0.00646	0.02273	0.07259	0.16850	0.41895	1.81052	3.62916	8.80468	57.91640	197.51124	658.33147
0.00636	0.00668	0.02295	0.07285	0.16878	0.42400	1.81446	3.64010	8.90698	58.32829	197.65374	692.75695
0.00542	0.00575	0.02201	0.07191	0.16784	0.42307	1.81352	3.63916	8.90605	58.32735	197.65280	692.75601

Figure B.10: Time Performance Data of Graph-tool (1 Core) – Part 1

Graph1 Closeness Time (s)	Graph2 Closeness Time (s)	Graph3 Closeness Time (s)	Graph4 Closeness Time (s)	Graph5 Closeness Time (s)	Graph6 Closeness Time (s)	Graph7 Closeness Time (s)	Graph8 Closeness Time (s)	Graph9 Closeness Time (s)	Graph10 Closeness Time (s)	Graph11 Closeness Time (s)	Graph12 Closeness Time (s)
0.00450	0.00619	0.01401	0.06066	0.10511	0.31676	1.41745	2.78043	8.71445	38.64033	101.83290	294.04677
0.00336	0.00506	0.01295	0.05988	0.10507	0.32108	1.41992	2.78398	8.88734	39.36973	102.46277	295.45471
0.00357	0.00526	0.01308	0.05973	0.10418	0.31583	1.41652	2.77951	8.71352	38.63940	101.83197	294.04584
0.00364	0.00536	0.01314	0.05997	0.10438	0.31986	1.41803	2.78027	8.74582	38.66958	101.95037	294.49955
0.00375	0.00545	0.01333	0.06027	0.10545	0.32147	1.42030	2.78436	8.88773	39.37012	102.46316	295.45510
Graph1 Connected Component Time (s)	Graph2 Connected Component Time (s)	Graph3 Connected Component Time (s)	Graph4 Connected Component Time (s)	Graph5 Connected Component Time (s)	Graph6 Connected Component Time (s)	Graph7 Connected Component Time (s)	Graph8 Connected Component Time (s)	Graph9 Connected Component Time (s)	Graph10 Connected Component Time (s)	Graph11 Connected Component Time (s)	Graph12 Connected Component Time (s)
0.00156	0.00156	0.00156	0.00617	0.00641	0.00594	0.03003	0.02901	0.02911	0.13822	0.15954	0.15872
0.00128	0.00127	0.00127	0.00589	0.00612	0.00566	0.02974	0.02872	0.02883	0.13793	0.15925	0.15843
0.01097	0.00210	0.00204	0.00701	0.00692	0.00671	0.03358	0.03038	0.03035	0.14039	0.16603	0.16132
0.00138	0.00134	0.00138	0.00601	0.00618	0.00600	0.03031	0.02873	0.02923	0.13841	0.16356	0.16031
0.01038	0.00151	0.00145	0.00642	0.00633	0.00612	0.03299	0.02980	0.02976	0.13980	0.16544	0.16073
Graph1 Strongly Connected Component Time (s)	Graph2 Strongly Connected Component Time (s)	Graph3 Strongly Connected Component Time (s)	Graph4 Strongly Connected Component Time (s)	Graph5 Strongly Connected Component Time (s)	Graph6 Strongly Connected Component Time (s)	Graph7 Strongly Connected Component Time (s)	Graph8 Strongly Connected Component Time (s)	Graph9 Strongly Connected Component Time (s)	Graph10 Strongly Connected Component Time (s)	Graph11 Strongly Connected Component Time (s)	Graph12 Strongly Connected Component Time (s)
0.00222	0.00246	0.00250	0.00812	0.00739	0.00790	0.02789	0.03238	0.03382	0.14029	0.16263	0.19317
0.00120	0.00144	0.00152	0.00545	0.00639	0.00678	0.02690	0.03133	0.03177	0.13896	0.16033	0.19190
0.00124	0.00147	0.00152	0.00714	0.00641	0.00692	0.02691	0.03139	0.03284	0.13930	0.16164	0.19218
0.00116	0.00143	0.00150	0.00697	0.00845	0.00695	0.02935	0.03316	0.03597	0.14540	0.16547	0.19533
0.00134	0.00162	0.00168	0.00716	0.00864	0.00714	0.02954	0.03335	0.03615	0.14558	0.16565	0.19551

Figure B.11: Time Performance Data of Graph-tool (1 Core) – Part 2

Graph1 Constructing Memory (MB)	Graph2 Constructing Memory (MB)	Graph3 Constructing Memory (MB)	Graph4 Constructing Memory (MB)	Graph5 Constructing Memory (MB)	Graph6 Constructing Memory (MB)	Graph7 Constructing Memory (MB)	Graph8 Constructing Memory (MB)	Graph9 Constructing Memory (MB)	Graph10 Constructing Memory (MB)	Graph11 Constructing Memory (MB)	Graph12 Constructing Memory (MB)
0.00000	0.00000	0.12891	0.13281	0.26953	1.18750	1.10156	1.91406	6.27344	4.76953	9.05078	31.16797
0.00000	0.00000	0.12891	0.13281	0.26953	1.18750	1.10156	1.91406	6.23047	4.76953	9.09375	31.18750
0.00000	0.00000	0.12891	0.13281	0.26953	1.18750	1.10156	1.91406	6.27344	4.76953	9.01563	31.18750
0.00000	0.00000	0.12891	0.13281	0.26953	1.18750	1.10156	1.91406	6.24609	4.76953	9.03516	31.16797
0.00000	0.00000	0.12891	0.13281	0.26953	1.18750	1.10156	1.91406	6.24609	4.76953	9.03516	31.16797
Graph1 Degree Memory (MB)	Graph2 Degree Memory (MB)	Graph3 Degree Memory (MB)	Graph4 Degree Memory (MB)	Graph5 Degree Memory (MB)	Graph6 Degree Memory (MB)	Graph7 Degree Memory (MB)	Graph8 Degree Memory (MB)	Graph9 Degree Memory (MB)	Graph10 Degree Memory (MB)	Graph11 Degree Memory (MB)	Graph12 Degree Memory (MB)
0.00000	0.00000	0.12891	0.14063	0.27344	1.17188	0.80859	1.73047	6.26563	3.97656	8.91406	31.26563
0.00000	0.00000	0.12891	0.14063	0.27344	1.17188	1.10547	2.10156	6.55078	5.02344	9.99219	32.33594
0.00000	0.00000	0.12891	0.14063	0.27344	1.17188	1.10547	2.10156	6.55078	5.02344	9.99219	32.33594
0.00000	0.00000	0.12891	0.14063	0.27344	1.17188	1.10547	2.10156	6.55078	5.02344	9.99219	32.33594
0.00000	0.00000	0.12891	0.14063	0.27344	1.17188	1.10547	2.10156	6.55078	5.02344	9.99219	32.33594
Graph1 PageRank Memory (MB)	Graph2 PageRank Memory (MB)	Graph3 PageRank Memory (MB)	Graph4 PageRank Memory (MB)	Graph5 PageRank Memory (MB)	Graph6 PageRank Memory (MB)	Graph7 PageRank Memory (MB)	Graph8 PageRank Memory (MB)	Graph9 PageRank Memory (MB)	Graph10 PageRank Memory (MB)	Graph11 PageRank Memory (MB)	Graph12 PageRank Memory (MB)
0.00000	0.00000	0.12891	0.15234	0.39844	1.17188	0.87500	1.73047	6.38672	3.98047	8.90625	31.16016
0.00000	0.00000	0.12891	0.15234	0.39844	1.17188	1.18359	2.12109	6.68359	5.28516	10.08594	32.32813
0.00000	0.00000	0.12891	0.15234	0.39844	1.17188	1.18359	2.12109	6.68359	5.28516	10.08594	32.32813
0.00000	0.00000	0.12891	0.15234	0.39844	1.17188	1.18359	2.12109	6.68359	5.28516	10.08594	32.32813
0.00000	0.00000	0.12891	0.15234	0.39844	1.17188	1.18359	2.12109	6.68359	5.28516	10.08594	32.32813
Graph1 Betweenness Memory (MB)	Graph2 Betweenness Memory (MB)	Graph3 Betweenness Memory (MB)	Graph4 Betweenness Memory (MB)	Graph5 Betweenness Memory (MB)	Graph6 Betweenness Memory (MB)	Graph7 Betweenness Memory (MB)	Graph8 Betweenness Memory (MB)	Graph9 Betweenness Memory (MB)	Graph10 Betweenness Memory (MB)	Graph11 Betweenness Memory (MB)	Graph12 Betweenness Memory (MB)
0.00000	0.00000	0.12891	0.14453	0.27344	1.19141	0.87891	1.84766	6.32422	3.98047	8.90234	31.16406
0.00000	0.00000	0.38672	0.14453	0.53125	1.85938	1.34375	2.89063	7.32031	5.28516	10.08984	32.33984
0.00000	0.00000	0.38672	0.14453	0.53125	1.85938	1.34375	2.89063	6.59766	5.28516	10.61719	38.17578
0.00000	0.00000	0.38672	0.14453	0.53125	1.85938	1.34375	2.89063	6.59766	5.28516	10.61719	38.17578
0.00000	0.00000	0.38672	0.14453	0.53125	1.85938	1.34375	2.89063	6.59766	5.28516	10.61719	38.17578

Figure B.12: Memory Performance Data of Graph-tool (1 Core) – Part 1

Graph1 Closeness Memory (MB)	Graph2 Closeness Memory (MB)	Graph3 Closeness Memory (MB)	Graph4 Closeness Memory (MB)	Graph5 Closeness Memory (MB)	Graph6 Closeness Memory (MB)	Graph7 Closeness Memory (MB)	Graph8 Closeness Memory (MB)	Graph9 Closeness Memory (MB)	Graph10 Closeness Memory (MB)	Graph11 Closeness Memory (MB)	Graph12 Closeness Memory (MB)
0.00000	0.00000	0.12891	0.14453	0.27344	1.16797	0.87500	1.84766	6.26563	3.98047	8.91016	31.07813
0.00000	0.00000	0.12891	0.14453	0.41797	1.16797	1.26172	2.12891	6.55469	5.28516	10.10156	32.28516
0.00000	0.00000	0.12891	0.14453	0.41797	1.16797	1.26172	2.12891	6.55469	5.28516	10.10156	32.28516
0.00000	0.00000	0.12891	0.14453	0.41797	1.16797	1.26172	2.12891	6.55469	5.28516	10.10156	32.28516
0.00000	0.00000	0.12891	0.14453	0.41797	1.16797	1.26172	2.12891	6.55469	5.28516	10.10156	32.28516
Graph1 Connected Component Memory (MB)	Graph2 Connected Component Memory (MB)	Graph3 Connected Component Memory (MB)	Graph4 Connected Component Memory (MB)	Graph5 Connected Component Memory (MB)	Graph6 Connected Component Memory (MB)	Graph7 Connected Component Memory (MB)	Graph8 Connected Component Memory (MB)	Graph9 Connected Component Memory (MB)	Graph10 Connected Component Memory (MB)	Graph11 Connected Component Memory (MB)	Graph12 Connected Component Memory (MB)
0.00000	0.00000	0.12891	0.14844	0.27734	1.18359	0.82031	1.86719	6.25000	3.97656	8.96484	31.15625
0.00000	0.00000	0.25781	0.28906	0.58984	1.48047	1.73047	2.72266	7.15625	4.97266	9.99219	32.15625
0.00000	0.00000	0.25781	0.28906	0.58984	1.48047	1.73047	1.89063	6.31641	4.41406	9.20313	31.33984
0.00000	0.00000	0.25781	0.28906	0.58984	1.48047	1.73047	1.89063	6.31641	4.41406	9.20313	31.33984
0.00000	0.00000	0.25781	0.28906	0.58984	1.48047	1.73047	1.89063	6.31641	4.41406	9.20313	31.33984
Graph1 Strongly Connected Component Memory (MB)	Graph2 Strongly Connected Component Memory (MB)	Graph3 Strongly Connected Component Memory (MB)	Graph4 Strongly Connected Component Memory (MB)	Graph5 Strongly Connected Component Memory (MB)	Graph6 Strongly Connected Component Memory (MB)	Graph7 Strongly Connected Component Memory (MB)	Graph8 Strongly Connected Component Memory (MB)	Graph9 Strongly Connected Component Memory (MB)	Graph10 Strongly Connected Component Memory (MB)	Graph11 Strongly Connected Component Memory (MB)	Graph12 Strongly Connected Component Memory (MB)
0.00000	0.00000	0.25781	0.14063	0.27344	1.17969	0.80078	1.86328	6.32031	3.96484	8.92969	31.07813
0.00000	0.00000	0.25781	0.14063	0.27344	1.31250	0.94531	2.23828	6.71094	4.88672	10.53906	32.75000
0.00000	0.00000	0.25781	0.14063	0.27344	1.31250	0.94531	2.23828	6.71094	5.01563	10.53906	32.75000
0.00000	0.00000	0.25781	0.14063	0.27344	1.31250	0.94531	2.23828	6.71094	5.01563	10.53906	32.75000
0.00000	0.00000	0.25781	0.14063	0.27344	1.31250	0.94531	2.23828	6.71094	5.01563	9.10547	31.28125

Figure B.13: Memory Performance Data of Graph-tool (1 Core) – Part 2

Graph1 Constructing Time (s)	Graph2 Constructing Time (s)	Graph3 Constructing Time (s)	Graph4 Constructing Time (s)	Graph5 Constructing Time (s)	Graph6 Constructing Time (s)	Graph7 Constructing Time (s)	Graph8 Constructing Time (s)	Graph9 Constructing Time (s)	Graph10 Constructing Time (s)	Graph11 Constructing Time (s)	Graph12 Constructing Time (s)
0.00449	0.02133	0.10822	0.02206	0.11502	0.55617	0.11166	0.55564	2.74063	0.56380	2.83075	14.11520
0.00455	0.02150	0.10941	0.02219	0.11513	0.56458	0.11296	0.55581	2.77451	0.57308	2.83257	14.12287
0.00460	0.02160	0.11080	0.02227	0.11524	0.56608	0.11303	0.56026	2.78462	0.57929	2.83932	14.12723
0.00470	0.02166	0.11123	0.02236	0.11559	0.56710	0.11551	0.56982	2.79509	0.58407	2.84431	14.14305
0.00483	0.02476	0.11213	0.02373	0.11595	0.57108	0.11647	0.57114	2.79620	0.58551	2.84441	14.21341
Graph1 Degree Time (s)	Graph2 Degree Time (s)	Graph3 Degree Time (s)	Graph4 Degree Time (s)	Graph5 Degree Time (s)	Graph6 Degree Time (s)	Graph7 Degree Time (s)	Graph8 Degree Time (s)	Graph9 Degree Time (s)	Graph10 Degree Time (s)	Graph11 Degree Time (s)	Graph12 Degree Time (s)
0.00068	0.00071	0.00068	0.00255	0.00249	0.00258	0.01173	0.01196	0.01181	0.05805	0.05953	0.05909
0.00052	0.00056	0.00052	0.00239	0.00234	0.00242	0.01157	0.01180	0.01166	0.05789	0.05937	0.05893
0.00053	0.00063	0.00053	0.00246	0.00243	0.00249	0.01160	0.01221	0.01170	0.05882	0.06081	0.06024
0.00071	0.00080	0.00070	0.00264	0.00261	0.00267	0.01177	0.01238	0.01187	0.05899	0.06099	0.06042
0.00063	0.00064	0.00063	0.00247	0.00244	0.00254	0.01175	0.01242	0.01216	0.05965	0.06249	0.06092
Graph1 PageRank Time (s)	Graph2 PageRank Time (s)	Graph3 PageRank Time (s)	Graph4 PageRank Time (s)	Graph5 PageRank Time (s)	Graph6 PageRank Time (s)	Graph7 PageRank Time (s)	Graph8 PageRank Time (s)	Graph9 PageRank Time (s)	Graph10 PageRank Time (s)	Graph11 PageRank Time (s)	Graph12 PageRank Time (s)
0.00257	0.00292	0.00260	0.00535	0.00558	0.00598	0.01983	0.02154	0.02403	0.10141	0.10476	0.14000
0.00218	0.00231	0.00227	0.00504	0.00526	0.00559	0.01943	0.02091	0.02368	0.10082	0.10194	0.13653
0.00228	0.00264	0.00232	0.00507	0.00530	0.00569	0.01954	0.02126	0.02375	0.10113	0.10448	0.13972
0.00260	0.00316	0.00254	0.00521	0.00534	0.00605	0.01985	0.02119	0.02389	0.10121	0.10826	0.14024
0.00279	0.00336	0.00273	0.00540	0.00553	0.00624	0.02004	0.02138	0.02409	0.10140	0.10845	0.14044
Graph1 Betweenness Time (s)	Graph2 Betweenness Time (s)	Graph3 Betweenness Time (s)	Graph4 Betweenness Time (s)	Graph5 Betweenness Time (s)	Graph6 Betweenness Time (s)	Graph7 Betweenness Time (s)	Graph8 Betweenness Time (s)	Graph9 Betweenness Time (s)	Graph10 Betweenness Time (s)	Graph11 Betweenness Time (s)	Graph12 Betweenness Time (s)
0.00214	0.00299	0.00897	0.02391	0.05260	0.13387	0.53945	1.04017	2.61873	21.49198	83.94183	383.31474
0.00185	0.00270	0.00868	0.02361	0.05231	0.13358	0.53915	1.03988	2.61844	21.49168	83.94154	383.31445
0.00188	0.00279	0.00873	0.02413	0.05239	0.13828	0.54454	1.05019	2.66995	24.70982	84.92146	386.42614
0.00200	0.00292	0.00886	0.02425	0.05252	0.13841	0.54466	1.05032	2.67008	24.70995	84.92159	386.42627
0.00224	0.00330	0.00905	0.02487	0.05278	0.13841	0.54616	1.05979	2.67171	25.57759	86.45102	389.50689

Figure B.14: Time Performance Data of Graph-tool (4 Cores) – Part 1

Graph1 Closeness Time (s)	Graph2 Closeness Time (s)	Graph3 Closeness Time (s)	Graph4 Closeness Time (s)	Graph5 Closeness Time (s)	Graph6 Closeness Time (s)	Graph7 Closeness Time (s)	Graph8 Closeness Time (s)	Graph9 Closeness Time (s)	Graph10 Closeness Time (s)	Graph11 Closeness Time (s)	Graph12 Closeness Time (s)
0.00334	0.00548	0.01323	0.02193	0.03480	0.08744	0.42207	0.77891	2.32441	13.00743	43.53132	93.66137
0.00347	0.00561	0.01336	0.02205	0.03493	0.08756	0.42220	0.77904	2.32454	13.00756	43.53144	93.66150
0.00350	0.00568	0.01340	0.02215	0.03517	0.08779	0.42252	0.77959	2.32924	16.48818	43.59580	94.22835
0.00442	0.00649	0.01436	0.02314	0.03802	0.08951	0.42744	0.78319	2.33875	16.51264	43.62174	94.86880
0.00366	0.00573	0.01359	0.02237	0.03726	0.08874	0.42668	0.78242	2.33798	16.51187	43.62097	94.86804
Graph1 Connected Component Time (s)	Graph2 Connected Component Time (s)	Graph3 Connected Component Time (s)	Graph4 Connected Component Time (s)	Graph5 Connected Component Time (s)	Graph6 Connected Component Time (s)	Graph7 Connected Component Time (s)	Graph8 Connected Component Time (s)	Graph9 Connected Component Time (s)	Graph10 Connected Component Time (s)	Graph11 Connected Component Time (s)	Graph12 Connected Component Time (s)
0.00130	0.00138	0.00127	0.00565	0.00571	0.00615	0.02709	0.02917	0.03013	0.14334	0.15695	0.14407
0.00130	0.00146	0.00128	0.00568	0.00630	0.00633	0.02829	0.03627	0.03058	0.14434	0.15736	0.14427
0.00144	0.00171	0.00149	0.00640	0.00856	0.00637	0.02920	0.04215	0.03065	0.14696	0.15834	0.15174
0.00158	0.00174	0.00156	0.00596	0.00658	0.00661	0.02858	0.03655	0.03086	0.14462	0.15764	0.14455
0.00156	0.00184	0.00162	0.00653	0.00869	0.00650	0.02933	0.04228	0.03077	0.14709	0.15847	0.15187
Graph1 Strongly Connected Component Time (s)	Graph2 Strongly Connected Component Time (s)	Graph3 Strongly Connected Component Time (s)	Graph4 Strongly Connected Component Time (s)	Graph5 Strongly Connected Component Time (s)	Graph6 Strongly Connected Component Time (s)	Graph7 Strongly Connected Component Time (s)	Graph8 Strongly Connected Component Time (s)	Graph9 Strongly Connected Component Time (s)	Graph10 Strongly Connected Component Time (s)	Graph11 Strongly Connected Component Time (s)	Graph12 Strongly Connected Component Time (s)
0.00145	0.00161	0.00166	0.00612	0.00641	0.00731	0.02635	0.03096	0.03172	0.12931	0.16264	0.17864
0.00126	0.00143	0.00148	0.00593	0.00623	0.00713	0.02617	0.03077	0.03154	0.12913	0.16246	0.17845
0.00127	0.00145	0.00159	0.00639	0.00626	0.00731	0.02636	0.03104	0.03209	0.13475	0.16356	0.18062
0.00146	0.00162	0.00166	0.00815	0.00657	0.00880	0.02952	0.03150	0.03226	0.13568	0.16693	0.18359
0.00126	0.00143	0.00147	0.00795	0.00638	0.00861	0.02932	0.03130	0.03207	0.13549	0.16674	0.18340

Figure B.15: Time Performance Data of Graph-tool (4 Cores) – Part 2

Graph1 Constructing Memory (MB)	Graph2 Constructing Memory (MB)	Graph3 Constructing Memory (MB)	Graph4 Constructing Memory (MB)	Graph5 Constructing Memory (MB)	Graph6 Constructing Memory (MB)	Graph7 Constructing Memory (MB)	Graph8 Constructing Memory (MB)	Graph9 Constructing Memory (MB)	Graph10 Constructing Memory (MB)	Graph11 Constructing Memory (MB)	Graph12 Constructing Memory (MB)
0.00000	0.00000	0.25781	0.13672	0.27344	1.17188	1.04688	1.86719	6.25391	4.82813	9.02734	31.05469
0.00000	0.00000	0.25781	0.13672	0.27344	1.17188	1.04688	1.86719	6.21484	4.82813	8.94922	31.05469
0.00000	0.00000	0.25781	0.13672	0.27344	1.17188	1.04688	1.86719	6.21484	4.82813	8.94922	31.05469
0.00000	0.00000	0.25781	0.13672	0.27344	1.17188	1.04688	1.86719	6.22656	4.82813	9.05469	31.19922
0.00000	0.00000	0.25781	0.13672	0.27344	1.17188	1.04688	1.86719	6.22656	4.82813	6.24609	31.19922
Graph1 Degree Memory (MB)	Graph2 Degree Memory (MB)	Graph3 Degree Memory (MB)	Graph4 Degree Memory (MB)	Graph5 Degree Memory (MB)	Graph6 Degree Memory (MB)	Graph7 Degree Memory (MB)	Graph8 Degree Memory (MB)	Graph9 Degree Memory (MB)	Graph10 Degree Memory (MB)	Graph11 Degree Memory (MB)	Graph12 Degree Memory (MB)
0.00000	0.00000	0.12891	0.15234	0.28125	1.16406	0.82422	1.86328	6.27344	3.98047	8.92188	31.07813
0.00000	0.00000	0.12891	0.15234	0.28125	1.16406	1.10938	2.10938	6.54297	5.02344	9.98438	32.17188
0.00000	0.00000	0.12891	0.15234	0.28125	1.16406	1.10938	2.10938	6.54297	5.02344	9.98438	32.17188
0.00000	0.00000	0.12891	0.15234	0.28125	1.16406	1.10938	2.10938	6.54297	5.02344	9.98438	32.17188
0.00000	0.00000	0.12891	0.15234	0.28125	1.16406	1.10938	2.10938	6.54297	5.02344	9.98438	32.17188
Graph1 PageRank Memory (MB)	Graph2 PageRank Memory (MB)	Graph3 PageRank Memory (MB)	Graph4 PageRank Memory (MB)	Graph5 PageRank Memory (MB)	Graph6 PageRank Memory (MB)	Graph7 PageRank Memory (MB)	Graph8 PageRank Memory (MB)	Graph9 PageRank Memory (MB)	Graph10 PageRank Memory (MB)	Graph11 PageRank Memory (MB)	Graph12 PageRank Memory (MB)
0.00000	0.00000	0.12891	24.14844	24.28516	25.17969	25.14453	26.26172	30.56250	29.29688	34.09766	56.38672
0.00000	0.00000	0.12891	24.14844	24.28516	25.17969	25.14453	26.26172	30.56250	29.29688	34.09766	56.38672
0.00000	0.00000	0.12891	24.14844	24.28516	25.17969	25.14453	26.26172	30.56250	29.29688	34.09766	56.38672
0.00000	0.00000	0.12891	24.14844	24.28516	25.17969	25.14453	26.26172	30.56250	29.29688	34.09766	56.38672
0.00000	0.00000	0.12891	24.14844	24.28516	25.17969	25.14453	26.26172	30.56250	29.29688	34.09766	56.38672
Graph1 Betweenness Memory (MB)	Graph2 Betweenness Memory (MB)	Graph3 Betweenness Memory (MB)	Graph4 Betweenness Memory (MB)	Graph5 Betweenness Memory (MB)	Graph6 Betweenness Memory (MB)	Graph7 Betweenness Memory (MB)	Graph8 Betweenness Memory (MB)	Graph9 Betweenness Memory (MB)	Graph10 Betweenness Memory (MB)	Graph11 Betweenness Memory (MB)	Graph12 Betweenness Memory (MB)
216.01172	216.14453	216.39844	216.14844	216.41797	217.85547	217.39844	218.78906	223.18750	221.29688	226.16016	248.28906
216.01172	216.14453	216.39844	216.14844	216.41797	217.85547	217.39844	218.78906	223.18750	221.29688	226.16016	248.28906
216.01172	216.14453	216.39844	216.14844	216.41797	217.85547	217.39844	218.78906	222.56641	221.29688	226.68750	252.63281
216.01172	216.14453	216.39844	216.14844	216.41797	217.85547	217.39844	218.91797	222.56641	221.29688	226.68750	253.81641
216.01172	216.14453	216.39844	216.14844	216.41797	217.85547	217.39844	218.91797	222.56641	221.29688	226.68750	253.81641

Figure B.16: Memory Performance Data of Graph-tool (4 Cores) – Part 1

Graph1 Closeness Memory (MB)	Graph2 Closeness Memory (MB)	Graph3 Closeness Memory (MB)	Graph4 Closeness Memory (MB)	Graph5 Closeness Memory (MB)	Graph6 Closeness Memory (MB)	Graph7 Closeness Memory (MB)	Graph8 Closeness Memory (MB)	Graph9 Closeness Memory (MB)	Graph10 Closeness Memory (MB)	Graph11 Closeness Memory (MB)	Graph12 Closeness Memory (MB)
0.00000	0.00000	0.25781	216.15234	216.43750	217.17969	217.23047	218.14063	222.59375	221.30078	226.10156	248.30078
0.00000	0.00000	0.25781	216.15234	216.43750	217.17969	217.23047	218.14063	222.59375	221.30078	226.10156	248.30078
0.00000	0.00000	0.25781	216.15234	216.43750	217.17969	217.23047	218.14063	222.59375	221.30078	226.10156	248.30078
0.00000	0.00000	0.25781	216.15234	216.43750	217.17969	217.23047	218.14063	222.59375	221.30078	226.10156	248.30078
0.00000	0.00000	0.25781	216.15234	216.43750	217.17969	217.23047	218.14063	222.59375	221.30078	226.10156	248.30078
Graph1 Connected Component Memory (MB)	Graph2 Connected Component Memory (MB)	Graph3 Connected Component Memory (MB)	Graph4 Connected Component Memory (MB)	Graph5 Connected Component Memory (MB)	Graph6 Connected Component Memory (MB)	Graph7 Connected Component Memory (MB)	Graph8 Connected Component Memory (MB)	Graph9 Connected Component Memory (MB)	Graph10 Connected Component Memory (MB)	Graph11 Connected Component Memory (MB)	Graph12 Connected Component Memory (MB)
0.00000	0.00000	0.12891	0.14063	0.27344	1.17188	0.88281	1.86719	6.34766	3.97656	8.90234	31.20703
0.00000	0.00000	0.26563	0.32422	0.43359	1.46875	1.79297	2.72266	7.23047	4.97656	9.94922	32.19922
0.00000	0.00000	0.26563	0.32422	0.43359	1.46875	1.79297	1.89453	6.39063	4.41406	9.15625	31.38281
0.00000	0.00000	0.26563	0.32422	0.43359	1.46875	1.79297	1.89453	6.39063	4.41406	9.15625	31.38281
0.00000	0.00000	0.26563	0.32422	0.43359	1.46875	1.79297	1.89453	6.39063	4.41406	9.15625	31.38281
Graph1 Strongly Connected Component Memory (MB)	Graph2 Strongly Connected Component Memory (MB)	Graph3 Strongly Connected Component Memory (MB)	Graph4 Strongly Connected Component Memory (MB)	Graph5 Strongly Connected Component Memory (MB)	Graph6 Strongly Connected Component Memory (MB)	Graph7 Strongly Connected Component Memory (MB)	Graph8 Strongly Connected Component Memory (MB)	Graph9 Strongly Connected Component Memory (MB)	Graph10 Strongly Connected Component Memory (MB)	Graph11 Strongly Connected Component Memory (MB)	Graph12 Strongly Connected Component Memory (MB)
0.00000	0.00000	0.12891	24.16406	24.28516	25.32422	24.96484	26.25000	30.66797	28.91406	34.53125	56.85156
0.00000	0.00000	0.12891	24.16406	24.28516	25.32422	24.96484	26.25000	30.66797	28.91406	34.53125	56.85156
0.00000	0.00000	0.12891	24.16406	24.28516	25.32422	24.96484	26.25000	30.66797	29.04297	34.55078	56.85156
0.00000	0.00000	0.12891	24.16406	24.28516	25.32422	24.96484	26.25000	30.66797	29.04297	34.55078	56.85156
0.00000	0.00000	0.12891	24.16406	24.28516	25.32422	24.96484	26.25000	30.66797	29.04297	33.09766	55.38281

Figure B.17: Memory Performance Data of Graph-tool (4 Cores) – Part 2

Appendix C

Experimental Queries

Query 1 (Join Operation + Sorting Operation): Show the question id, the owner id and the tag label of top 10 questions that have the most view count.

For PostgreSQL and RG engine:

```
SELECT topQ.Qid, topQ.Owner_id, tag.Tag_label
FROM LABELLED_BY AS lb, TAG,
(
    SELECT Qid, Owner_id, View_count
    FROM QUESTION
    ORDER BY View_count DESC
    LIMIT 10
) AS topQ
WHERE lb.Qid = topQ.Qid AND lb.Tid = tag.Tid;
```

For Neo4j:

```
MATCH (t:Tag) -[:LABELS]-> (q:Question)
RETURN q.Qid, q.Owner_id, t.Tag_label
ORDER BY q.View_count DESC
LIMIT 10;
```

Query 2 (Join Operation + Sorting Operation + Aggregate Operation): Show the top 5 answerers and their latest reputation score in an descending order based on the number of their answers that accepted by questions.

For PostgreSQL and RG engine:

```
SELECT Owner_id, max(Score) AS score
FROM ANSWER
WHERE Owner_id IN
(
    SELECT a.Owner_id
    FROM ANSWER AS a, QUESTION AS q
    WHERE q.Accepted_aid = a.Aid AND a.Owner_id != 0
    GROUP BY a.Owner_id
    ORDER BY count(a.Aid) DESC
    LIMIT 5
)
GROUP BY Owner_id
ORDER BY score DESC;
```

For Neo4j:

```
MATCH (q:Question) −[r:ACCEPTS_USER]−>(user:User)
RETURN user.Uid, user.Score, count(r)
ORDER BY COUNT(r) DESC
LIMIT 5;
```

Query 3 (Join Operation + Sorting Operation + Aggregate Operation + Set Operation): Show the number of articles of each journal and proceeding along with the journal name and the proceeding title in a descending order.

For PostgreSQL and RG engine:

```
SELECT jo.Name AS name, jo.Publication_date, arcount.count
FROM JOURNAL AS jo,
(
    SELECT ar.JOid, count(ar.ARid)
    FROM ARTICLE AS ar
    GROUP BY ar.JOid
) AS arcount
WHERE jo.JOid = arcount.JOid AND jo.Name != ''
UNION
SELECT pr.Title AS name, pr.Publication_date, arcount.count
FROM PROCEEDING AS pr,
(
    SELECT ar.PRid, count(ar.ARid)
    FROM ARTICLE AS ar
    GROUP BY ar.PRid
) AS arcount
WHERE pr.PRid = arcount.PRid AND pr.Title != ''
ORDER BY count DESC;
```

For Neo4j:

```
MATCH (ar:Article) −[r:PUBLISHED_IN]−> (jo:Journal)
RETURN jo.Name AS name, jo.Publication_date AS date, count(r) AS count
ORDER BY count DESC
UNION
MATCH (ar:Article) −[r:PUBLISHED_IN]−> (pr:Proceeding)
RETURN pr.Title AS name, pr.Publication_date AS date, count(r) AS count
ORDER BY count DESC;
```

Query 4 (Pattern Matching): Recommend 10 twitter users for Jack who currently does not follow these users but Jack follows somebody who are following them.

For PostgreSQL and RG engine:

```
SELECT Uid, Display_name FROM TW_USER
WHERE Display_name != 'jack' AND Uid IN
(
    SELECT f1.Uid
    FROM FOLLOW AS f1, FOLLOW AS f2
    WHERE f1.Follower_id = f2.Uid AND f1.Uid NOT IN
    (
        SELECT Uid FROM FOLLOW WHERE Follower_id IN
        (SELECT Uid FROM TW_USER WHERE Display_name = 'jack')
    )
)
LIMIT 10;
```

For Neo4j:

```
MATCH (jack:User {Display_name: 'jack'}) -[:FOLLOWS]-> ( ),
          ( )-[:FOLLOWS]-> (other:TW_user)
WHERE NOT ((jack) -[:FOLLOWS]-> (other))
RETURN other.Uid, other.Display_name
LIMIT 10;
```

Query 5 (Triangle Counting): Count the number of triangles of the co-authorship network.

For PostgreSQL and RG engine:

```
SELECT count(*)
FROM coauthorship AS c1
JOIN coauthorship AS c2 ON c1.CoAUid = c2.AUid AND c1.AUid < c2.AUid
JOIN coauthorship AS c3 ON
    c2.CoAUid = c3.AUid AND c3.CoAuid = c1.AUid AND c2.AUid < c3.AUid;
```

For Neo4j:

```
:GET /service/mazerunner/analysis/triangle_count/COAUTHOR

MATCH (au1:Author) -[r1:COAUTHOR]-> (au2:Author),
        (au2:Author)-[r2:COAUTHOR]-> (au3:Author),
        (au3:Author)-[r3:COAUTHOR]-> (au1:Author)
WHERE au2.AUid <> au1.AUid AND au3.AUid <> au2.AUid
        AND au3.AUid <> au1.AUid
RETURN count(*);
```

Query 6 (PageRank Centrality): Find the top 10 influential authors according to the pagerank centrality in the co-authorship network.

For RG engine:

```
SELECT Fname, Mname, Lname
FROM author WHERE AUid IN
(
    SELECT VertexID
    FROM RANK (coauthorship, pagerank)
    LIMIT 10
);
```

For Neo4j:

```
:GET /service/mazerunner/analysis/pagerank/COAUTHOR

MATCH (au:Author) WHERE has(au.pagerank)
RETURN au.Fname, au.Mname, au.Lname, au.pagerank AS pagerank
ORDER BY pagerank DESC
LIMIT 10;
```

Query 7 (Connected Component): Count the number of connected components of the co-authorship network.

For RG engine:

```
SELECT count(ClusterID) FROM CLUSTER (coauthorship, CC)
```

For Neo4j:

```
:GET /service/mazerunner/analysis/connected_components/COAUTHOR

MATCH (au:Author) WHERE has(au.connected_components)
RETURN count(DISTINCT au.connected_components)
```

Query 8 (Path Finding): Find paths with length less than 2, which connect two author V1 and V2 in the co-authorship network where author V1 is affiliated at ANU and author V2 is affiliated at UNSW.

For RG engine:

```
SELECT *
FROM PATH (coauthorship, V1/./V2)
WHERE V1 AS
(
  SELECT AUid FROM AUTHOR WHERE affiliation like '%ANU%'
) AND V2 AS
(
  SELECT AUid FROM AUTHOR WHERE affiliation like '%UNSW%'
);
```

For Neo4j:

```
MATCH p=((n1:Author) −[r:COAUTHOR*1..2]− (n2:Author))
WHERE n1.affiliation =~ '.* ANU.*' AND n2.affiliation =~ '.* UNSW.*'
RETURN [ n IN nodes(p) | n.AUid]
```

Query 9 (Shortest Path): Find a shortest paths between two authors Michael Norrish and Kevin Elphinstone in the co-author network.

For RG engine:

```
SELECT *
FROM PATH (coauthorship, V1//V2)
WHERE V1 AS
(
  SELECT AUid FROM AUTHOR
  WHERE Fname = 'Michael' AND Lname = 'Norrish'
) AND V2 AS
(
  SELECT AUid FROM AUTHOR
  WHERE Fname = 'Kevin' AND Lname = 'Elphinstone'
)
ORDER BY Length ASC;
```

For Neo4j:

```
MATCH p=shortestPath((n1:Author) −[r:COAUTHOR*]− (n2:Author))
WHERE n1.Fname='Michael' AND n1.Lname = 'Norrish' AND n2.Fname = 'Kevin'
            AND n2.Lname = 'Elphinstone'
RETURN [ n IN nodes(p) | n.AUid]
```

Query 10 (Community Detection): Find a group of tags that they are often used together to label a question.

For RG engine:

```
CREATE UNGRAPH cotag AS
(
  SELECT lb1.Tid as Tid, lb2.Tid AS CoTid
  FROM LABELLED_BY AS lb1, LABELLED_BY AS lb2
  WHERE lb1.Qid = lb2.Qid AND lb1.Tid != lb2.Tid
);

SELECT Tag_label
FROM TAG,
(
  SELECT Members
  FROM CLUSTER (cotag, CNM)
  LIMIT 1
) AS c
WHERE Tid = ANY(c.Members);
```

Query 11 (PageRank Centrality + Connected Component): According to the pagerank centrality, find the top 3 authors of the biggest collaborative community in the co-authorship network

For RG engine:

```
SELECT VertexID, Value
FROM RANK (coauthorship, pagerank) AS r,
(
  SELECT Members
  FROM CLUSTER (coauthorship, CC)
  ORDER BY Size DESC
  LIMIT 1
) AS c
WHERE r.VertexID = ANY(c.Members)
LIMIT 3;
```

Query 12 (PageRank Centrality + Path Finding): According to the pagerank centrality, show how the top 2 authors connect with each other in the co-authorship network.

For RG engine:

```
SELECT PathID, Length, Path
FROM PATH (coauthorship, V//V)
WHERE V AS
(
  SELECT VertexID
  FROM RANK (coauthorship, pagerank)
  LIMIT 2
);
```

Bibliography

[1] A Tour of PostgreSQL Internals. http://www.postgresql.org/files/developer/tour.pdf.

[2] AllegroGraph RDFStore Web 3.0's Database. http://franz.com/agraph/allegrograph/.

[3] Apache Jena - Home. https://jena.apache.org.

[4] Centrality measures - graph-tool 2.10 documentation. http://graph-tool.skewed.de/static/doc/centrality.html.

[5] GenRndGnm - Snap.py 1.2 documentation. http://snap.stanford.edu/snappy/doc/reference/GenRndGnm.html.

[6] Giraph - Welcome To Apache Giraph! http://giraph.apache.org.

[7] Graph-tool: Efficient network analysis. http://graph-tool.skewed.de.

[8] Import Data Into Neo4j - Neo4j Graph Database. http://neo4j.com/developer/guide-importing-data-and-etl/.

[9] Importing CSV Data into Neo4j - Neo4j Graph Database. http://neo4j.com/developer/guide-import-csv/.

[10] Julia Benchmarks. http://julialang.org/benchmarks/.

[11] libpq - C Library. http://www.postgresql.org/docs/current/static/libpq.html.

[12] Neo4j and Apache Spark - Neo4j Graph Database. http://neo4j.com/developer/apache-spark/#mazerunner.

[13] Neo4j Graph Database. http://neo4j.com/product/.

[14] Neo4j, the World's Leading Graph Database. http://neo4j.com.

[15] OrientDB - OrientDB Multi-Model NoSQL DatabaseOrientDB Multi-Model NoSQL Database. http://orientdb.com/orientdb/.

[16] Overview - NetworkX. http://networkx.github.io.

[17] PostgreSQL + Python — Psycopg. http://initd.org/psycopg/.

[18] PostgreSQL 9.4.5 Documentation. http://www.postgresql.org/docs/9.4/interactive/index.html.

[19] PostgreSQL: The world's most advanced open source database. http://neo4j.com.

[20] Relational Algebraic Equivalence Transformation Rules. http://www.postgresql.org/message-id/attachment/32513/EquivalenceRules.pdf.

[21] Stanford Network Analysis Project. http://snap.stanford.edu.

[22] Stardog: Enterprise Graph Database. http://stardog.com.

[23] The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org.

[24] Titan: Distributed Graph Database. http://thinkaurelius.github.io/titan/.

[25] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[26] Ulrik Brandes and Thomas Erlebach. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

[27] Chungmin Melvin Chen and Nicholas Roussopoulos. *The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching*. Springer, 1994.

[28] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98. ACM, 2012.

[29] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[30] Emil Eifrem. The New Way to Access Super Fast Social Data. http://mashable.com/2012/09/26/graph-databases/.

[31] Paul Erdős and Alfréd Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Hungarica*, 12(1-2):261–267, 1961.

[32] Jing Fan, Adalbert Gerald, Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. 2015.

[33] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[34] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[35] Jiewen Huang, Kartik Venkatraman, and Daniel J Abadi. Query optimization of distributed pattern matching. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 64–75. IEEE, 2014.

[36] Abhishek Jindal and Steve Madden. GRAPHiQL: A graph intuitive query language for relational databases. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 441–450. IEEE, 2014.

[37] Alekh Jindal, Samuel Madden, Malu Castellanos, and Meichun Hsu. Graph Analytics using the Vertica Relational Database. *arXiv preprint arXiv:1412.5263*, 2014.

[38] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[39] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[40] Yuhanna Noel, Owens Leslie, and Elizabeth Cullen. Market Overview: Graph Databases. https://www.forrester.com/Market+Overview+Graph+Databases/fulltext/-/E-res121473.

[41] Lassila Ora and Swick Ralph. Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[42] Tiago P Peixoto. Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models. *Physical Review E*, 89(1):012804, 2014.

[43] Luis L Perez and Christopher M Jermaine. History-aware query optimization with materialized intermediate views. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 520–531. IEEE, 2014.

[44] Abdul Quamar, Amol Deshpande, and Jimmy Lin. NScale: neighborhood-centric analytics on large graphs. *Proceedings of the VLDB Endowment*, 7(13):1673–1676, 2014.

[45] Ramakrishnan Raghu and Gehrke Johannes. *Database Management Systems, 3rd Edition*. McGraw-Hill Education, 2003.

[46] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013.

[47] Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-SPARQL: a hybrid engine for querying large attributed graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 335–344. ACM, 2012.

[48] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.

[49] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[50] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*, 2013.

[51] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2013.