Rogas: A Declarative Framework for Network Analytics

Minjian Liu
minjian.liu@anu.edu.au

Qing Wang
qing.wang@anu.edu.au

Research School of Computer Science, The Australian National University
Canberra, ACT 0200, Australia

ABSTRACT

Network analytics has become increasingly popular in recent years. Various graph systems have been developed for analysing networks, while network data is still largely stored and managed in relational database systems in the first place. As two separate systems are often used to manage and analyse network data, it not only increases the difficulty for users to learn and maintain these different systems simultaneously, but also impedes performing more sophisticated analysis on relational and topological properties of network data. Aiming to tackle these issues, we present *Rogas* in this paper, which is a declarative framework that allows the user to formulate analysis queries naturally without thinking about the tedious implementation details of graph algorithms and query processing.

1. INTRODUCTION

Nowadays, more and more large networks become available, such as social networks, biological networks, and bibliographical networks. Analysing these networks to discover and predict patterns is increasingly critical for many enterprises and organisations. In practice, network data is often stored and managed in relational database systems. Nonetheless, relational database systems have limitations to perform network analytics. For example, it is difficult to use SQL to express simple operations such as finding friends of friends in a social network, and SQL programming would require multiple joins and recursion which goes beyond the expressive power of standard SQL queries. Thus, a deluge of graph systems with different concerns have been developed, such as Graph-tool [1], NetworkX [2], SNAP [3], Pregel [11], GraphLab [10], G-SPARQL [13], Gbase [9], and GraphX [15]. Accordingly, the most common scenario for network analytics is: (1) exporting data from relational database systems to text files (e.g. CSV, XML, and TXT), (2) importing those text files into graph systems, (3) running analysis and getting results from graph systems.

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

However, this common scenario severely restricts the capability and flexibility of analysis. As two separate systems are used to manage and analyse network data, it not only increases the difficulty for users to learn and maintain these different systems simultaneously, but also impedes performing more sophisticated analysis on relational and topological properties of network data. To circumvent this issue, several SQL-based graph systems have been recently proposed [6, 8]. Although these SQL-based graph systems have incorporated additional operations into the traditional SQL query processing, they failed to provide a declarative framework that allows the user to easily specify network analytics at a high-level abstraction. Other state-of-the-art graph systems are either procedural in nature for which the user needs to take care of query optimization, data distribution, locking and synchronization by themselves, e.g. Pregel [11] and GraphLab [10], or only focus on certain graph algorithms, for instance, in terms of community detection, SNAP only provides modularity-based algorithms [5, 7] whilst Graph-tool provides stochastic blockmodel algorithms [12].

To tackle the issues above, we present a declarative framework for network analytics, called *Rogas*. Unlike existing works, our intention is to develop a high-level declarative query language that enables the user to formulate analysis queries naturally without thinking about the tedious implementation details of graph algorithms and query processing. To achieve this, we extend SQL in a way that elegantly generalises the standard SQL operations such as `GROUP BY` and `ORDER BY` into the graph primitives such as `CLUSTER` and `RANK`. In doing so, the user only needs to conceptually specify queries with desired graph operations and sends these queries to the query engine behind. Then the query engine is able to automatically select most efficient algorithms for execution. Nonetheless, how the query engine will execute such queries can be flexible, which is a decision of the system developer. For example, a hybrid memory and disk engine as described in [13] may be used for executing queries, which maintains topological structures in memory while the data is stored in a relational database. In addition to performing sophisticated analysis collectively based on both relational and topological properties of network data, *Rogas* also provides several other advantages, such as: to support dynamic analysis of network data which may discover new and evolving knowledge about networks, and to efficiently execute analysis queries through query plans optimized by the query engine. Most importantly, this framework enables us to semantically align and mine the relationships of various analysis queries and govern their semantic integrity [14].

2. FRAMEWORK OVERVIEW

This framework has three main components: a hybrid data model, a SQL-like query language and a query engine. In the data model, network data is stored as relations in one or more relational databases. When performing network analysis, we use *mapper queries* (refer to Section 2.1) to map relations to materialised or temporary graphs. Upon these graphs, the query engine performs network analysis tasks by choosing appropriate algorithms provided by graph systems (refer to Section 2.2). The results of analysing graphs are transformed into relational databases. Now we discuss about the query language and the query engine in details.

2.1 Query Language

We propose a SQL-like query language, called RG-SQL, which extends the standard SQL with graph construction, ranking, clustering and path finding operations, while still preserving the nice closure property of SQL. Unlike the existing query languages of SQL-based graph systems [6, 8], RG-SQL is a high-level declarative query language which is easy-to-use in terms of writing up queries.

As a running example, we use the ACM bibliographical network $\mathfrak{N}_{ACM}$, which has a number of relations stored in a relational database. For example, it has `AUTHOR` that is a relation storing the information of authors (e.g. Aid, Name, Affiliation, Email), `PAPER` storing the information of publications (e.g. Pid, Title, PublicationDate), `WRITES` storing the authors of publications (e.g. Aid, Pid) and `CITES` storing the papers of citation (e.g. Pid, CitedPid). To exemplify the main features of RG-SQL, let us consider the following queries over $\mathfrak{N}_{ACM}$:

Q1 (Top-k influential authors) Find the top k influential authors in terms of their influence of co-authorship.

Q2 (Collaborative communities) Find communities that consist of authors who collaborate with each other to write papers together.

Q3 (Shortest path) Find a shortest path between authors Minjian and Qing so that Minjian is able to know who to contact if he wants to work with Qing.

Graph Construction We use a mapper query to map one or more relations to a graph for network anlytics. Essentially, mapper queries are a special kind of relational algebra queries that only return edge lists. For example, a coauthorship graph `coauthorship` can be created from $\mathfrak{N}_{ACM}$ using the following mapper query.

```
CREATE UNGRAPH coauthorship AS
(
   SELECT w1.Aid AS Aid, w2.Aid AS CoAid
   FROM WRITES AS w1, WRITES AS w2
   WHERE w1.Pid = w2.Pid AND w1.Aid != w2.Aid
);
```

Two different types of graphs can be constructed: `CREATE UNGRAPH` creates an undirected graph, while `CREATE DIGRAPH` creates a directed graph. A mapper query has the form of `SELECT-FROM-WHERE`, which extracts an edge list from relations in the underlying databases for graph construction. All mapper queries follow two rules: (1) The number of attributes in the `SELECT` clause shall be two; (2) These attributes shall also be the primary keys of certain relations.

Ranking Operation In network analytics, we are often interested in vertex centrality that indicates the importance of vertices within a graph, such as Q1. In RQ-SQL, we provide a `RANK` operator to specify the ranking operation for vertices. The syntax is defined as follows:

`RANK`(<graph name>, <measure>)
<measure> := `degree` | `indegree` | `outdegree` | `betweenness` | `closeness` | `pagerank`

A number of measures are available for determining the importance of vertices [4]. One may choose the most suitable measure for a specific query based on the type of the graph and desired properties. For example, in terms of Q1, we can apply the following query to find the top 3 influential authors in terms of their influence of co-authorship using the `closeness` measure. The result of `RANK(coauthorship, closeness)` is stored in a result table that has two predefined attributes: `VertexID` and `Value`. The user may flexibly select attributes from these pre-defined ones, together with attributes from other relational and graph operations.

```
SELECT VertexID, Value
FROM RANK(coauthorship, closeness)
LIMIT 3;
```

Clustering Operation Finding a cluster of vertices over a graph is one of the most common tasks in network analytics, such as Q2. We define a `CLUSTER` operator in RG-SQL using the following syntax:

`CLUSTER`(<graph name>, <algorithm>)
<algorithm> := `CC` | `SCC` | `GN` | `CNM` | `MC`

In the above, `CC` refers to an algorithm of finding connected components, `SCC` an algorithm of finding strongly connected components, and `GN`, `CNM` and `MC` three algorithms for community detection, which respectively correspond to Girvan-Newman algorithm [7], Clauset-Newman-Moore Algorithm [5] and Peixoto's modified Monte Carlo Algorithm [12]. For Q2, we may use the following query to find the collaborative communities of the co-authorship graph and list the result in a descending order based on the community size. The result of `CLUSTER(coauthorship, MC)` is stored in a result table that has three pre-defined attributes: `ClusterID`, `Size` and `Members`.

```
SELECT ClusterID, Size, Members
FROM CLUSTER(coauthorship, MC)
ORDER BY Size DESC;
```

Path Finding Operation Path finding aims to discover paths that connect two or more vertices, such as Q3. For this, we define a `PATH` operator in RG-SQL to specify how to find paths in a graph with the following syntax:

`PATH`(<graph name>, <path expression>)
<path expression> := `.` | `V` |
<path expression>/ <path expression>|
<path expression>// <path expression>

In a path expression, `V` is a vertex expression that imposes certain condition on the vertices of a path, `.` is a *do-not-care* symbol indicating that any vertex is allowed in its position, `/` represents one edge, and `//` represents any number of edges. A path expression is *valid* if it contains a vertex expression in the first and last positions. For Q3, we may use the path expression `V1//V2`, where `V1` represents Minjian and `V2` represents Qing, to find a shortest path between Minjian and Qing in the following query. The result of `PATH(coauthorship, V1//V2)` is stored in a result table with three pre-defined attributes: `PathID`, `Length` and `Path`.

```
SELECT PathID, Length, Path
FROM PATH(coauthorship, V1//V2)
WHERE V1 AS
    ( SELECT Aid FROM AUTHOR WHERE Name = 'Minjian Liu')
AND V2 AS
    ( SELECT Aid FROM AUTHOR WHERE Name = 'Qing Wang')
ORDER BY Length ASC LIMIT 1;
```

2.2 Query Engine

We develop a query engine to process queries written in RG-SQL. In a nutshell, the query engine extends the relational query engine of PostgreSQL by incorporating a component called *operation executor* for handling graph operations that relate to various graph systems used for algorithm support. Figure 1 presents the architecture of the engine.

Similar to traditional query processing, a query written in RG-SQL is processed by following 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. A plan tree may contain two different types of operation nodes: graph operation nodes, i.e. corresponding to `RANK`, `CLUSTER` and `PATH` operations, and relational operation nodes, i.e. corresponding to selection, join, aggregate and other operations in SQL. For each plan tree, the query optimiser firstly passes its graph operation nodes to the operation executor, then enumerates alternative plan trees, estimates their costs, and determines the best execution plan. For each query, the query optimiser chooses the best possible graph system to support the operation executor processing graph operations. Based on the chosen execution plan, the plan executor controls the execution order and processes relational operations (e.g. table scans, nested-loop joins, sorting, and aggregation) while the operation executor processes graph operations (e.g. ranking, clustering, and path finding). When executing graph operations, the operation executor retrieves graph data from the underlying data storage layer, runs graph algorithms provided by different graph systems over the graph data, transforms the results into result tables, and sends back to the data storage layer.

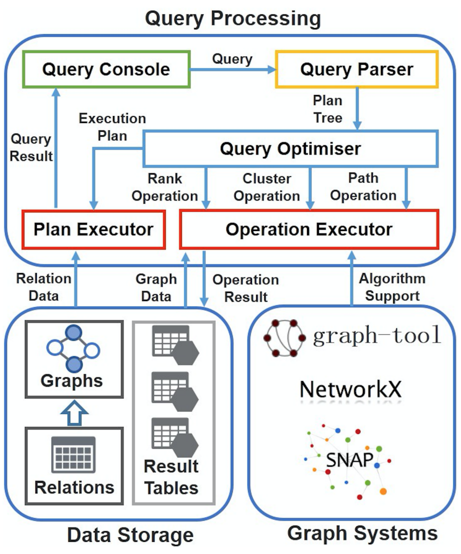


Figure 1: Architecture of the Query Engine

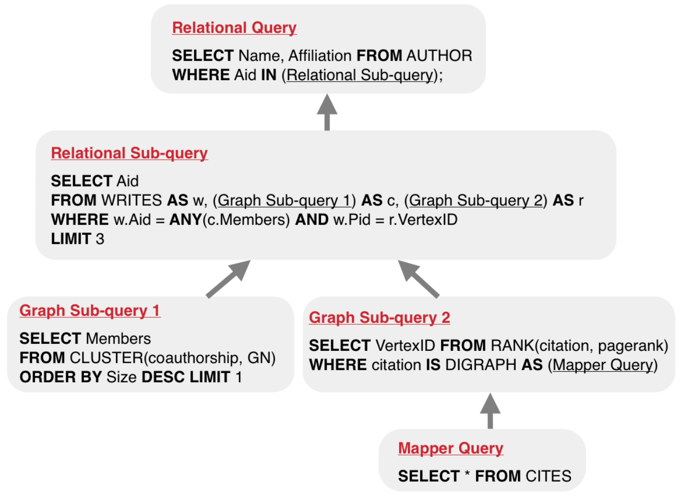

Relational Query
SELECT Name, Affiliation FROM AUTHOR
WHERE Aid IN (Relational Sub-query);
Relational Sub-query
SELECT Aid
FROM WRITES AS w, (Graph Sub-query 1) AS c, (Graph Sub-query 2) AS r
WHERE w.Aid = ANY(c.Members) AND w.Pid = r.VertexID
LIMIT 3
Graph Sub-query 1
SELECT Members
FROM CLUSTER(coauthorship, GN)
ORDER BY Size DESC LIMIT 1
Graph Sub-query 2
SELECT VertexID FROM RANK(citation, pagerank)
WHERE citation IS DIGRAPH AS (Mapper Query)
Mapper Query
SELECT * FROM CITES

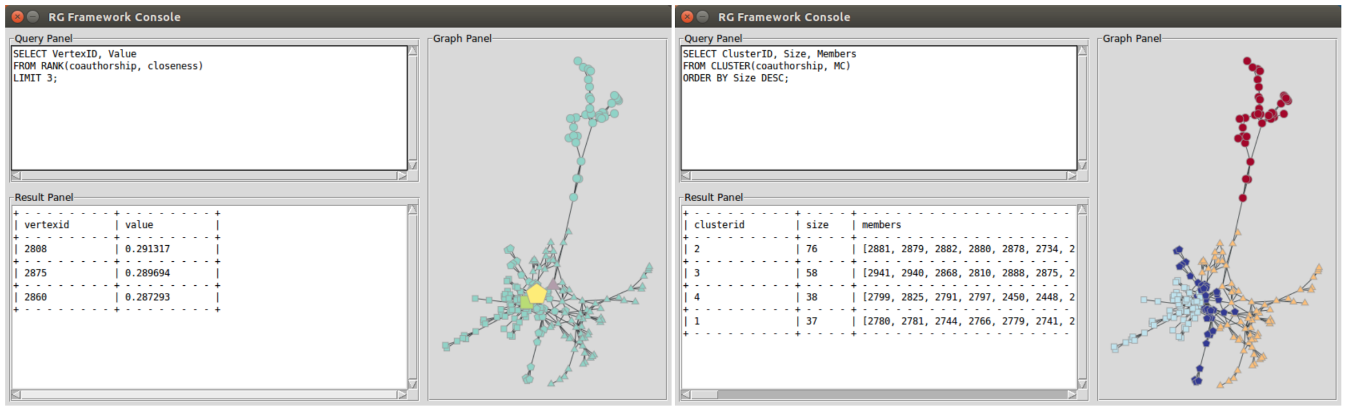
Figure 2: Query Decomposition

3. QUERY OPTIMISATION

The main features of RG-SQL allow users to write sophisticated queries that combine relational analysis and graph analysis. In order to process these sophisticated queries more efficiently, the query parser separates relational sub-queries from graph sub-queries. For example, the following query over $\mathfrak{N}_{ACM}$ can be decomposed into a set of sub-queries including relational sub-queries and graph sub-queries, as shown in Figure 2.

```
SELECT Name, Affiliation FROM AUTHOR
WHERE Aid IN
(
   SELECT Aid FROM WRITES as w,
   (
      SELECT Members
      FROM CLUSTER(coauthorship, GN)
      ORDER BY Size DESC
      LIMIT 1
   ) as c,
   (
      SELECT VertexID
      FROM RANK(citation, pagerank)
      WHERE citation IS DIGRAPH AS
      (
         SELECT * FROM CITES
      )
   ) as r
   WHERE w.Aid = ANY(c.Members) AND w.Pid = r.VertexID
   LIMIT 3
);
```

Thus, after a query has been parsed, a decomposed RG-SQL query Q consists of a set of relational and graph sub-queries $\{q_1, q_2, \ldots, q_n\}$. One main job of the query optimiser is to rewrite these sub-queries into semantically equivalent sub-queries that can be performed more efficiently. For relational sub-queries, we leverage existing query optimisation techniques of PostgreSQL such as the transformation rules based on relational algebraic equivalence, the genetic optimisation algorithms for searching alternative plan trees and so forth. For graph sub-queries, we aim to build a caching pool through hashing query results so as to avoid repeated computation.



```
SELECT VertexID, Value
FROM RANK(coauthorship, closeness)
LIMIT 3;
```

```
+ - - - - - - - - - + - - - - - - - - - +
| vertexid          | value             |
+ - - - - - - - - - + - - - - - - - - - +
| 2808              | 0.291317          |
+ - - - - - - - - - + - - - - - - - - - +
| 2875              | 0.289694          |
+ - - - - - - - - - + - - - - - - - - - +
| 2860              | 0.287293          |
+ - - - - - - - - - + - - - - - - - - - +
```

```
SELECT ClusterID, Size, Members
FROM CLUSTER(coauthorship, MC)
ORDER BY Size DESC;
```

```
+ - - - - - - - - - - + - - - - + - - - - - - - - - - - - - - - - - -
| clusterid           | size    | members
+ - - - - - - - - - - + - - - - + - - - - - - - - - - - - - - - - - -
| 2                   | 76      | [2881, 2879, 2882, 2880, 2878, 2734, 2
+ - - - - - - - - - - + - - - - + - - - - - - - - - - - - - - - - - -
| 3                   | 58      | [2941, 2940, 2868, 2810, 2888, 2875, 2
+ - - - - - - - - - - + - - - - + - - - - - - - - - - - - - - - - - -
| 4                   | 38      | [2799, 2825, 2791, 2797, 2450, 2448, 2
+ - - - - - - - - - - + - - - - + - - - - - - - - - - - - - - - - - -
| 1                   | 37      | [2780, 2781, 2744, 2766, 2779, 2741, 2
+ - - - - - - - - - - + - - - - + - - - - - - - - - - - - - - - - - -
```

Figure 3: Demonstration Graphical Interface

4. DEMONSTRATION

In this section, we will demonstrate a prototype system of this framework. The system is implemented in Python 2.7 and is deployed on an Intel Core i7 3.6GHz, 16GB server with a bibliographical network dataset provided by the ACM Digital Library. The source code of the system can be found in `https://github.com/CornucopiaRG/Rogas.git`.

As shown in Figure 3, the attendees can enter queries in the query panel, get the query results from the result panel, and visualise the query results as graphs in the graph panel. We take Q1 and Q2 mentioned in Section 2 as examples. Figure 3.(a) contains a graph with three highlighted vertices (i.e. the yellow pentagon, the green square and the purple triangle) representing the top three influential authors for Q1. Figure 3.(b) presents a graph with four types of vertices in different colors and shapes representing four collaborative communities for Q2.

In the demonstration, we will provide more queries about the ACM bibliographical network to the attendees, in addition to the queries show in Figure 3. Through our prototype system, the attendees can run relational queries for relational analysis, create mapper queries to construct materialised or temporary graphs, perform various graph operations for network analysis, and conduct a sophisticated analysis based on both relations and graphs.

5. CONCLUSIONS

We have presented a declarative framework for network analytics that includes a relation-graph hybrid data model, a SQL-like query language extending SQL with network analysis operations and a query engine being able to incorporate various network analysis algorithms provided by different graph systems. We have also demonstrated our prototype implementation of this framework.

6. ACKNOWLEDGMENTS

We thank the ACM Digital Library for providing the data set of the ACM bibliographical network.

7. REFERENCES

[1] Graph-tool. `http://graph-tool.skewed.de`.
[2] NetworkX. `http://networkx.github.io`.
[3] SNAP. `http://snap.stanford.edu`.
[4] U. Brandes and T. Erlebach. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.
[5] A. Clauset, M. E. Newman, and C. Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.
[6] J. Fan, A. Gerald, S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, pages 1–10, 2015.
[7] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.
[8] A. Jindal and S. Madden. GRAPHiQL: A graph intuitive query language for relational databases. In *IEEE International Conference on Big Data*, pages 441–450, 2014.
[9] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: an efficient analysis platform for large graphs. *The VLDB Journal*, 21(5):637–650, 2012.
[10] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.
[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, pages 135–146, 2010.
[12] T. P. Peixoto. Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models. *Physical Review E*, 89(1):012804, 2014.
[13] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: a hybrid engine for querying large attributed graphs. In *CIKM*, pages 335–344. ACM, 2012.
[14] Q. Wang. Network Analytics ER Model – Towards a Conceptual View of Network Analytics. In *Conceptual Modeling*, pages 158–171. Springer, 2014.
[15] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.