Query Optimization in the RG Framework

Chong Feng
u4943054

supervised by
Dr. Qing Wang

COMP6470 Special Topics in Computing (6 units)
Research School of Computer Science
Australian National University

May 27, 2016
Abstract

This report proposes a mechanism for query engine optimisation of the Relation-Graph (RG) Framework, which is a unified framework for network analytics developed at the Australian National University. The mechanism is built upon the core implementation of the RG Framework, which is called the RG engine, with the aim to improve time efficiency of query processing. A new query optimiser is incorporated into the query engine which can cache the execution results of queries formulated using the query language of the RG Framework, and reuse the results of queries that have been previously executed when possible. In my experiments, it shows that the proposed query optimiser can improve query handling time as well as overall efficiency of the RG engine.
Contents

1 Introduction .................................................. 3
  1.1 Research Problem ........................................ 4
  1.2 Objectives ................................................. 5
  1.3 Contributions ............................................. 6

2 Background and Related Work ................................. 7
  2.1 The RG Framework ........................................ 7
  2.2 Query Engine Optimisation ................................ 9
    2.2.1 Shared Pool Check .................................... 9
    2.2.2 Query Transformation ................................ 9

3 Query Optimisation ............................................ 11
  3.1 Design Overview .......................................... 11
  3.2 Query Tree Structure .................................... 12
  3.3 Query Parser .............................................. 15
  3.4 Query Optimiser ......................................... 16

4 Evaluation ..................................................... 18
  4.1 Experimental Environment ................................. 18
    4.1.1 Hardware ............................................. 18
    4.1.2 Software ............................................ 18
  4.2 Dataset and Queries ...................................... 19
    4.2.1 Dataset ............................................. 19
    4.2.2 Queries ............................................. 19
  4.3 Experimental Results .................................... 21

5 Conclusion ................................................... 25
Chapter 1

Introduction

Graph theory has been one of the most rapidly growing research areas in mathematics over the last forty year. Its results have been applied to many areas of the computing, social sciences, and natural sciences [2, 8]. For example, networks have been studied extensively in the social sciences. This is due to the fact that typical social network studies use vertices and edges in a graph to represent individuals and interactions, and address issues such as centrality and connectivity in a social network [5].

In 2015, a unified framework (RG framework) for network analytics was developed at the Australian National University. This framework is designed to provide users with an easy-to-use, yet powerful querying mechanism to deal with network analysis tasks as well as relational analysis tasks [4]. The framework consists of the following three main parts:

- Relation-Graph (RG) model
- Relation-graph Structured Query Language (RG-SQL)
- Relation-Graph (RG) engine

The RG model is a hybrid model that contains a relation core in the center with a number of graphical views surrounded. The RG model serves as a basis for network data management and analysis. The RG-SQL is a query language that extends the standard SQL language by integrating with a set of graph operations and algorithms including path finding, ranking and clustering.

The focus of this project is on the RG engine, which is the core implementation of the RG framework. The engine is built upon the PostgreSQL query engine with the capability of handling RG-SQL queries. As illustrated in Figure 1.1, the RG engine is made of a Query Analyser and Rewriter, three graph
operation executors (i.e. Rank Query Executor, Cluster Query Executor, and Path Query Executor) and APIs to external database engines. Users can interact with the RG engine through a graphical query console. When a new query is submitted to proceed from the query console, the Query Analyser and Rewriter will parse the query, extract its sub-queries that include graph operations, and submit the sub-queries to different graph operation executors accordingly. The results of these executions are stored as temporary tables in the underlying database or as text files on local machine. After all of the graph operations are executed, the engine will rewrite the original RG-SQL query by replacing the graph operations in the original RG-SQL query with the intermediate execution results and commit the rewritten query to the PostgreSQL query engine for final execution. After that, the final execution result is returned to the query console [4].

1.1 Research Problem

Currently the RG framework does not have any optimisation mechanisms or strategies being incorporated into the RG engine. Queries are treated as new and isolated individuals each time they come in and this has introduced an issue of inefficiency as some of the query executions could be unnecessary

Figure 1.1: An overview of the RG Engine, taken from [4]
and repetitive. Considering the following motivating example, if we run the query as shown in Example Query 1.1 twice consecutively, the current RG engine will need to repeat the exact same execution procedures as in the first time. This is undesirable as we could reuse some execution results from the first time, i.e. execution result of the sub-query shown in (2) can be reused to improve the performance for the second time as long as the relevant data from the underlying database remains unchanged. This kind of optimisation is particularly important for queries that are performed upon large datasets.

```
SELECT Fname, Lname
FROM author WHERE Auid IN
WHERE twitterPost is UNGRAPH as
(
  SELECT VertexID
  FROM RANK (coauthorship, pagerank)
  LIMIT 20
);
```

Example Query 1.1: Rank Operation

1.2 Objectives

The goal of this project is to design and develop an optimisation mechanism that can be incorporated into the existing RG framework to reduce the cost of executing computation-intensive queries through reusing execution results of repeated queries or sub-queries. Particularly, the following objectives are required:

- Improve the functionality of the RG-SQL query analyser so it can parse the RG-SQL queries and construct execution plans using an effective data structure.
- Develop query optimisation mechanisms that can reuse the results from previous executions to reduce query processing time.
- Evaluate the performance of the optimisation mechanisms and propose future optimisation strategies.
1.3 Contributions

In this project, I have made the following contributions:

- I have designed a new tree structure to serve as the foundation of query parsing and query structure comparing. The tree structure is able to store queries on its nodes and each node can point to multiple child nodes which contain its sub-queries.

- I have optimised the previous Query Analyser and Rewriter so the new parser can split different components of a RG-SQL queries and build an execution tree for each query. The parser will then match the tree structure with the cache to find equivalent tree patterns from previous executions and reuse repeated results.

- I have conducted experiments to evaluate the performance of the new query engine. Four experimental queries as well as a real-world dataset were set up for the experiments, and each query was tested using both the original query engine and the new query engine.
Chapter 2

Background and Related Work

In recent years, a shift in network analysis has been witnessed with the focus moving away from the analysis of small single graphs and the properties of limited number of vertices or edges within this kind of graph to large-scale graphs that may have millions or billions vertices [5]. In real-life applications, network analysis tasks are often carried out upon relational databases. Data needs to be retrieved from underlying databases before applying analytical techniques on it. A key limitation of this model lies in the separation of managing network data and analysing network data as data of large volume needs to be transferred back and forth between databases and analytical tools. Network data is often managed in relational databases while network analyses are usually performed in ad hoc and isolated environments, and this may cause efficiency and integrity issues [7].

2.1 The RG Framework

For this reason, the RG framework was developed with the aim of diminishing the gap between network data management and network analysis. The framework is beneficial as it allows users to: 1) skip the process of data exporting and importing, and 2) gain more valuable insights by combining network analysis and relational analysis together [3].

The RG Framework, as shown in Figure 1.1, contains a query console which is a graphical user interface that allows users to interact with the RG engine using RG-SQL queries. The RG-SQL queries extends traditional SQL queries with three graph operations as follows [4]:

- Rank Operation. The Rank operation finds the ranking of vertices within a graph based on a number of measures, including degree, indegree, outdegree and more.
• Cluster Operation. The Cluster operation specifies a group of vertices by using algorithms for connected components and community detection.

• Path Operation. The Path operation finds the path of different length between two vertices.

For example, the following Example Query 2.1 is a RG-SQL query which contains a “Rank” operator together with the parameters “twitterPost” (the name of the graph) and “indegree” (the measure). It specifies a graph operation that finds the ranking of each node in terms of indegree, and the graph “twitterPost” is generated by the sub-query “SELECT qid, tid FROM labelled_by LIMIT 20”.

Example Query 2.1: Rank Operation

```
SELECT * FROM (1)
RANK(twitterPost, indegree) (2)
WHERE twitterPost is UNGRAPH as
{
    SELECT qid, tid FROM labelled_by LIMIT 20 (3)
};
```

When the query is entered into the console, the parser will first go through the query string to locate graph operators, i.e. “Rank”, “Cluster” and “Path” in order to construct graphs that are required by these graph operators. The RG framework provides two approaches to construct a graph, either by the on-the-fly approach or the materialised approach. On-the-fly means that the graph will be stored in the shared memory instead of databases or local disks, which will significantly reduce the I/O cost of reading and storing data. Otherwise, graphs are constructed and stored persistently in the database as materialised views. The materialised approach is more appropriate when a graph needs to be used for multiple times.

As in the Example Query 2.1, the graph “twitterPost” is defined in the “WHERE” clause, and constructed using the on-the-fly approach which means that the graph will be temporarily stored in the local shared memory. The sub-query (3), being the most inner relational sub-query, will first be extracted out and passed to the PostgreSQL query engine to execute. The execution result will be returned as a text file saved in the shared memory.
Then the sub-query (2), together with the pointer to the returned file, will then be parsed and submitted to the “rank executor” as shown in Figure 1.1. The rank executor is capable of performing ranking algorithms using external packages and the result of sub-query (2) will be saved as a temporary table in the PostgreSQL database. When all graph operations are successfully executed in the original RG-SQL query, they will be replaced by the result table name and the whole query (1) will be submitted to the PostgreSQL query engine for final execution. The final result will be returned to the console.

2.2 Query Engine Optimisation

Query optimisation exists in almost every commercial relational database management systems. Typically the following optimisation mechanisms are used [6].

2.2.1 Shared Pool Check

The database maintain a shared memory space that contains the parse tree and execution plan for a SQL statement, and each SQL statement is assigned a hash value. When a user submits a SQL query, the database engine will search for the hash value of the query in the shared space to see if it has the same parsed statement. If the query already exists in the shared pool and it is reusable, the database engine will skip the process of execution plan generation and query plan generation, and reuse the existing code [6].

2.2.2 Query Transformation

Another traditional query optimisation technique is to rewrite a query into a semantically equivalent query that returns the same result but can be performed more efficiently. The query optimiser can determine and calculate the cost of alternative plans for a given SQL query, and choose the plan with the lowest cost to proceed [1].

There are many forms of query transformations. For example, as shown in Example Query 2.2 a query optimiser can transform a query with an OR operator into a query that uses the UNION ALL operator for efficiency purpose [6].
Example Query 2.2: Query Transformation

```
SELECT * FROM sales
WHERE promo_id=33
OR prod_id=136;

SELECT * FROM sales
WHERE promo_id=33
UNION ALL
SELECT * FROM sales
WHERE prod_id=136
WHERE promo_id=33
```
Chapter 3

Query Optimisation

This chapter presents the data structure and algorithms I developed for query optimisation. Section 3.1 gives an overview of the optimised RG engine. Section 3.2 introduces the data structure served as the foundation of query optimisation for the parser that is developed in my project. Section 3.3 explains the idea behind the new query parser and Section 3.4 presents the idea of optimisation strategies.

3.1 Design Overview

In the original implementation of the RG engine [4], the Query Analyser & Rewriter contains the a query parser as shown in Figure 3.1. This original query parser is capable of parsing the RG-SQL queries and submit parsed queries to different graph operation executors.

Based on this implementation, I developed three new classes in this project and integrated them into the Query Analyser & Rewriter as shown in Figure 3.2:

- **Optimised Query Parser**
  Unlike the previous implementation of the parser which parses a RG-SQL query reversely and only extracts out graph operations [4], the new parser I developed in this project goes through a RG-SQL query sequentially to match and extract out both graph operations and relational sub-queries using regular expressions.

- **Query Tree**
  The Query Tree is a tree structure for the new query parser. The tree stores different parts of a RG-SQL query on each of its nodes, and
Figure 3.1: Original Query Parser and Rewriter

supports tree pattern matching which compares two tree structures to find reusable tree patterns.

• **Query Optimiser**
  The Query Optimiser looks up the cache to see if a query has been executed before. If so, the optimiser will retrieve the previous execution result from the cache and return it to the console. If the query is new, the optimiser will generate a hash value for the query and execute the query, then store the execution result together with the query’s hash value in the cache.

### 3.2 Query Tree Structure

In the original RG engine [4], when a RG-SQL query is submitted, the query parser checks the syntax of the query and locates graph operators reversely and iteratively, i.e. the last graph operator will be located first. Using the reverse order allows the analyser to find the innermost graph operator that is not dependent on any other graph operators. In each iteration, a graph operator together with its relational sub-queries will be extracted and processed, the execution result will be stored in database as temporary tables for further execution.

The problem with this query parser is that it does not provide a high-level view of the query structure. As a RG-SQL query allows to have a complex combination of standard SQL queries and graph operators, it is common that a graph operator, being the sub-query of a relational query or another
graph operator, is dependent on the execution results of its relational sub-queries. To resolve this issue, I developed a tree structure as well as a new query parser to parse and represent the RG-SQL query structure by storing sub-query blocks and query types (relational type or graph type) on to tree nodes.

The tree structure I propose, as shown in Figure 3.3, is not a binary tree as each node can have any number of child nodes, and the child nodes are stored as an array under their parent node. This is due to the fact that a RG-SQL query may have an arbitrary number of sub-queries. The root of the tree corresponds to the original RG-SQL query and each of its child nodes corresponds to one of the original query’s immediate sub-queries. If the level 2 sub-queries also have sub-queries, their sub-queries will be added to the level 3 nodes until there are no more sub-queries.

Each node is given a unique node ID to facilitate searching and indexing. The naming convention of the nodes is defined by the rules below:

- The root’s ID is “0”
- A node other than the root is given the ID “parent ID + the index of
the node in the child node array"

For example, the root in Figure 3.3 is given an ID of “0”, its child nodes are “00”, “01” and “02”, and the child nodes of node “00” are named “000” and “001”.

For each of the nodes in a tree, a Python dictionary variable is created and stored, which has the following structure:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node ID</td>
<td>[Original Query, Rewritten Query]</td>
</tr>
</tbody>
</table>

Table 3.1: Node Dictionary Variable

The key of such a dictionary variable is the node ID, and the value is an array that contains the original query and the rewritten query. In the following section, I will explain how I developed a query parser and query optimiser to improve the performance of the RG engine.
3.3 Query Parser

The new query parser I develop parses the RG-SQL queries sequentially and iteratively to construct a query tree for each RG-SQL query. The parser will first initialise a tree structure and assign the original query to the root of the tree before it starts parsing. In each parsing iteration, the parser goes through the query on each tree node to find parentheses or graph operators that mark the start of sub-queries and assign these sub-queries to child nodes. Considering the following Example Query 3.1, the tree is initialised and the root of the tree contains the original query. In the first iteration, the parser pointer starts parsing from the first level, i.e. the root of the tree. It will first detect a left parenthesis as show on Line 3, then it will continue parsing the query to find the corresponding right parenthesis (on Line 6) and assign the sub-query between Line 3 and Line 6 to the child node of the root. Afterwards, the query will continue parsing from Line 6 to the end of the query. In this case, there are no other sub-queries for the original query. After the first iteration, the parser pointer will move to the second level of the tree and start parsing the sub-queries that are on the second-level nodes. As the sub-query between Line 3 and Line 6 also contains another sub-query, in the second parsing iteration, the “Cluster” graph operator on Line 5 will be detected and it will be added to the level 3 of the tree. The complete parsed tree is shown in Figure 3.4

| SELECT tag_label                                   | (1) |
| FROM tag,                                         | (2) |
| (                                                   | (3) |
| SELECT members                                     | (4) |
| FROM CLUSTER (cotag, CNM) LIMIT 1                 | (5) |
| ) AS c                                             | (6) |
| WHERE tid = ANY(c.members);                        | (7) |

Example Query 3.1: Cluster Operation [4]

When storing queries to tree nodes, both the original queries and rewritten queries will be stored. The approach of rewriting a query is to replace its sub-queries by their sub-queries’ node ID. As shown in Figure 3.4, the original query contains the sub-query “SELECT members From CLUSTER(cotag,CNM) LIMIT 1”, thus this part is replaced by node ID “00”.

15
Saving rewritten queries to tree nodes facilitates query executions. When the new query parser has successfully constructed a query tree for a RG-SQL query, it will call the relevant query processing functions in the original query parser to execute the query tree. Queries are executed bottom-up based on their positions on a query tree, meaning that lower-level queries will be executed first and their execution results will be passed to their parent nodes as execution input. Therefore, using rewritten queries can accelerate the process of replacing sub-queries with their execution results as node ID is easier to locate and index.

3.4 Query Optimiser

The query optimiser has two main functionalities. The first one is to check if a query has been executed before. When a new query comes in, before parsing and other query processing take place, the query optimiser will first generate a hash value for it and check it against the cache to see whether there exists the same hash value. Currently the cache is represented by a python dictionary variable with hash value being the key and execution result being the value. If the hash value does not exist in the cache, which means the query is unseen before, the optimiser will initialise the parsing procedures to process the query and store the execution result together with
the query’s hash value in the cache. If the same query exists in the cache in the first place, the query optimiser will skip the parsing process and retrieve the previous execution result directly from the cache and return it to the console.

Another main functionality of the query optimiser is to store and retrieve tree patterns and their execution results. When a RG-SQL query is executed, the tree nodes that contain graph operations together with the intermediate execution result table names will be stored in the cache. The execution results will also be stored in the database as tables for the purpose of future reuse. Take the query in Figure 3.5 as an example, the query on node 00 is a “Rank” graph operation, thus it will be stored in the cache together with the execution result table name. When another query comes in, the query optimiser will compare the node 00 against the new query tree to check if the pattern exists in the new query tree. If so, the optimiser will first retrieve the execution result table name from the cache and then retrieve the data in the corresponding database table.
Chapter 4

Evaluation

The chapter presents the results of the experiments conducted to evaluate the performance of the new query optimiser developed in this project. In the experiments, I have compared the performance of the RG engine with optimisation and the performance of the original RG engine. Section 4.1 introduces the experimental environment including hardware and software information. Section 4.2 describes the datasets and queries used in the experiments. Section 4.3 presents the outcome of the experiments as well as some insights of the outcome.

4.1 Experimental Environment

4.1.1 Hardware

All experiments were conducted on a 2010 13 inch macbook pro with a 2.4GHz Intel Core 2 Duo CPU (p8600), 6 GB of memory running at 1066MHz, and a 256 GB Samsung solid state drive.

4.1.2 Software

The software information is presented in Table 4.1:
4.2 Dataset and Queries

4.2.1 Dataset

The dataset used to evaluate the performance of the query optimiser is the Stack Overflow network dataset (ST network)\(^1\). The Stack Overflow network dataset contains the following four tables:

- Labelled by table: 13,466,686 records
- Tag table: 38205 records
- Question table: 7,990,787 records
- Answer table: 13,684,117 records

The dataset has the schema as shown in Figure 4.1 [4]:

4.2.2 Queries

As the RG engine supports multiple graph operations. Four experimental queries were set up at different levels of complexity for different graph operations to evaluate the performance. Each query was executed five times and the elapsed time of each execution was recorded. The average time of each query was used for plotting and comparison. Each query was also executed on different sizes of graphs. For example, in Query 4.1, the size of the graph was set to be 100, 1000 and 10000 (LIMIT keyword on line 1). The queries are shown in the following listings from Experimental Query 4.1 to Experimental Query 4.4:

---
\(^1\)Provided by Stanford Network Analysis Platform (http://snap.stanford.edu/proj/snap/icwsm/)
---

19
SELECT *
FROM RANK(Answer, indegree)
WHERE Answer is UNGRAPH AS

( SELECT Aid, Parent_Qid
FROM Answer LIMIT 100
); (1)

Experimental Query 4.1: Rank Operation

CREATE UNGRAPH QuestionOwner AS

( SELECT Qid, Owner_id
FROM Question LIMIT 100
); (1)

SELECT *
FROM CLUSTER(QuestionOwner, CNM);

Experimental Query 4.2: Cluster Operation
SELECT * FROM question,
{
  
  SELECT members
  FROM CLUSTER(questionAnswer, CNM)
  WHERE questionAnswer is UNGRAPH AS
  {
    SELECT aid, Parent_Qid FROM Answer limit 100;  
  }
  as t1
WHERE qid = ANY(t1.members);
}

Experimental Query 4.3: Cluster Operation

SELECT t1.Accepted_Aid
FROM (SELECT * FROM question LIMIT 1000000) AS t1
LEFT JOIN
(SELECT * FROM (SELECT * FROM Answer LIMIT 1000000) AS t2) AS t3
ON t1.Accepted_Aid = t3.Aid
UNION
SELECT VertexID
FROM RANK(Answer, indegree)
WHERE Answer is UNGRAPH AS
{
  SELECT Aid, Parent_Qid FROM Answer LIMIT 100
};

Experimental Query 4.4: Rank Operation

4.3 Experimental Results

The experimental results are shown in Figure 4.2 to Figure 4.5, in which each query was tested using both the original RG engine (without optimisation, represented by an orange line) and the optimised RG engine with the new query optimiser (with optimisation, represented by a blue line). Note that when testing on the optimised RG engine, each query needs to pre-run for at least once in order to store the tree patterns in the cache. The elapsed time measured in the experiments only contains the time used in the necessary query processing procedures such as query parsing, tree constructing, pattern matching, query executing and query pattern retrieving and storing. This is to eliminate the variance generated from other procedures that are irrelevant to the core query processing.
In addition, it is worth noting that in the experiments, I manually turned off the functionality of whole query caching, i.e. caching the whole query and its execution result. This is due to the fact that if a query can reuse the previous execution result of the same query, it will skip all of the processing procedures and simply return the cached result to the console. It is trivial to evaluate the performance of this functionality as we can easily anticipate a huge performance improvement.

As the original RG engine does not have any optimisation mechanisms, each query has to go through the same procedures every time they are submitted to process. Therefore the time needed for query processing scales up with the increase of graph size. This is reflected as blue lines in Figure 4.2 to Figure 4.5.

However, given that the graph operations are already cached in the memory, the new RG engine outperforms the original RG engine by a large margin. This is due to that graph operations are generally more computationally expensive compared with relational query executions. Therefore, if we can skip the computationally expensive parts of RG-SQL query processing, it will save a large amount of time. This is reflected as orange lines in Figure 4.2 to Figure 4.5.

If we increase the computation complexity of relational parts in a RG-SQL query, we see reduction in the effect of optimisation as the optimisation mechanisms proposed in this project only work on graph operations. This is reflected in Figure 4.5 and Experimental Query 4.4 where I increased the computation intensity of the relational sub-queries. Thus, in Figure 4.5, the difference between the orange line and the blue line is not as significant as in other figures.

Therefore, the performance of the new RG engine is closely related to the structure of the queries being submitted to it. If we run a number of totally different queries or the proportion of similar patterns among these queries is low, the optimisation mechanisms proposed in this project will not have any significant effect as these mechanisms rely on reusable query patterns and execution results generated from previous queries. Thus, to fully understand how effective the optimisation mechanisms are, experiments using a large query set from real-life scenario are required.
Figure 4.2: Result for Experimental Query 4.1

Figure 4.3: Result for Experimental Query 4.2
Figure 4.4: Result for Experimental Query 4.3

Figure 4.5: Result for Experimental Query 4.4
Chapter 5

Conclusion

The RG Framework provides a powerful platform for network analytics using data stored in relation databases. It allows users to conduct network analysis tasks directly upon relational databases, thus eliminating the process of transferring data back and forth between databases and network analytical tools. However, as currently there is no optimisation mechanisms incorporated into the RG Framework, in this project I have developed a query optimiser as well as a new query parser and query tree structure to optimise the performance of the RG engine. For each RG-SQL query, the new query parser builds a query tree for it which serves as the foundation of tree pattern matching. The query optimiser is capable of looking up tree patterns in the cache to find reusable execution results and skip the computation of reusable tree patterns to achieve better performance.

Also, it is found that the optimisation effectiveness is largely dependent on the types of queries being adopted for the experiments. To better understand the performance of the query optimiser, it is necessary to experiment using queries from real-life applications.

Therefore, there are a number of directions we may continue to explore as the future work:

- Conduct experiments using queries from real-life scenarios to better understand how effective the proposed mechanism is.

- Explore other optimisation mechanisms to further improve the efficiency of the RG framework.
Appendix
<table>
<thead>
<tr>
<th>Query1 W/O Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>0.412840</td>
<td>0.422674</td>
<td>0.240456</td>
<td>0.240087</td>
<td>0.250705</td>
<td>0.313352</td>
</tr>
<tr>
<td>Size 1000</td>
<td>3.668003</td>
<td>3.686244</td>
<td>3.615030</td>
<td>3.527160</td>
<td>3.730830</td>
<td>3.645453</td>
</tr>
<tr>
<td>Size 10000</td>
<td>34.657651</td>
<td>34.399928</td>
<td>34.188881</td>
<td>34.924377</td>
<td>34.84135</td>
<td>34.602437</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query1 W/ Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>0.001376</td>
<td>0.001739</td>
<td>0.001203</td>
<td>0.001348</td>
<td>0.001413</td>
<td>0.001416</td>
</tr>
<tr>
<td>Size 1000</td>
<td>0.003485</td>
<td>0.002739</td>
<td>0.003160</td>
<td>0.002726</td>
<td>0.003048</td>
<td>0.003032</td>
</tr>
<tr>
<td>Size 10000</td>
<td>0.018336</td>
<td>0.017655</td>
<td>0.016816</td>
<td>0.017684</td>
<td>0.017135</td>
<td>0.017525</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query2 W/O Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>0.196239</td>
<td>0.201575</td>
<td>0.204861</td>
<td>0.193413</td>
<td>0.215315</td>
<td>0.202281</td>
</tr>
<tr>
<td>Size 1000</td>
<td>1.093686</td>
<td>1.108618</td>
<td>1.049560</td>
<td>1.039340</td>
<td>1.058546</td>
<td>1.069950</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query2 W/ Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>0.001505</td>
<td>0.001414</td>
<td>0.002485</td>
<td>0.001594</td>
<td>0.001228</td>
<td>0.001645</td>
</tr>
<tr>
<td>Size 1000</td>
<td>0.002854</td>
<td>0.001767</td>
<td>0.001857</td>
<td>0.002088</td>
<td>0.001673</td>
<td>0.002048</td>
</tr>
<tr>
<td>Size 10000</td>
<td>0.015856</td>
<td>0.007203</td>
<td>0.008014</td>
<td>0.008479</td>
<td>0.007314</td>
<td>0.009373</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query3 W/O Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>0.228490</td>
<td>0.198167</td>
<td>0.215184</td>
<td>0.239184</td>
<td>0.195187</td>
<td>0.215242</td>
</tr>
<tr>
<td>Size 1000</td>
<td>2.035191</td>
<td>1.930862</td>
<td>1.985720</td>
<td>1.961812</td>
<td>2.418132</td>
<td>2.066343</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query3 W/ Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>0.003763</td>
<td>0.007672</td>
<td>0.006342</td>
<td>0.003877</td>
<td>0.005211</td>
<td>0.005373</td>
</tr>
<tr>
<td>Size 1000</td>
<td>0.014950</td>
<td>0.013384</td>
<td>0.012998</td>
<td>0.016585</td>
<td>0.013432</td>
<td>0.014270</td>
</tr>
<tr>
<td>Size 10000</td>
<td>0.087173</td>
<td>0.086586</td>
<td>0.085188</td>
<td>0.870492</td>
<td>0.086696</td>
<td>0.243227</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query4 W/O Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>4.179858</td>
<td>3.682276</td>
<td>3.725647</td>
<td>3.981254</td>
<td>4.025185</td>
<td>3.918844</td>
</tr>
<tr>
<td>Size 1000</td>
<td>7.713140</td>
<td>7.290824</td>
<td>7.384330</td>
<td>7.541205</td>
<td>7.618204</td>
<td>7.509541</td>
</tr>
<tr>
<td>Size 10000</td>
<td>43.32918</td>
<td>43.489130</td>
<td>42.816068</td>
<td>43.968913</td>
<td>43.471952</td>
<td>43.415049</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query4 W/ Optimisation</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Time4</th>
<th>Time5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 100</td>
<td>2.126189</td>
<td>1.342888</td>
<td>1.492585</td>
<td>2.769258</td>
<td>3.568997</td>
<td>2.258263</td>
</tr>
<tr>
<td>Size 1000</td>
<td>2.103221</td>
<td>1.148211</td>
<td>1.914967</td>
<td>3.062455</td>
<td>3.037150</td>
<td>2.253201</td>
</tr>
<tr>
<td>Size 10000</td>
<td>1.163945</td>
<td>1.167765</td>
<td>1.935184</td>
<td>3.020913</td>
<td>3.048567</td>
<td>2.067275</td>
</tr>
</tbody>
</table>
1 Project Title

Query Optimization in the RG Framework

2 Description

The RG Framework is a platform for network analytics which integrates a collection of graph analysis tools and algorithms into a unified framework in order to support various network analysis tasks efficiently and effectively. The query engine in the RG Framework supports an extended SQL query language, called RG-SQL, which extends relational algebra operators with a number of primitive graph constructors.

As network analysis queries are often computationally expensive, an efficient query optimizer is vital for improving the efficiency of processing network analysis queries over graphs, relations, or a mix of them. The goal of this project is to improve the performance of the query engine IN the RG Framework. On the completion of the project, the following learning objectives are expected to achieve:

- Have a good understanding of state-of-the-art query optimization techniques used in database applications;
- Get hands on experience in developing a query engine for processing and optimizing queries over relations and graphs;
- Develop query optimization methods to improve the performance of executing RG-SQL queries, based on the analysis on how relational data and graph data are used and stored in the RG Framework;
- Be able to implement the developed query optimization methods to test how well they are used in real-world applications and be able to analyze the results.
Bibliography


